HAMRAZ: Resilient Partitioning and Replication

Xiao Li Farzin Houshmand Mohsen Lesani
University of California, Riverside

***Abstract*—Inter-organizational systems where subsystems with partial trust need to cooperate are common in healthcare, finance and military. In the face of malicious Byzantine attacks, the ultimate goal is to assure end-to-end policies for the three aspects of trustworthiness: confidentiality, integrity and availability. In contrast to confidentiality and integrity, provision and validation of availability has been often sidestepped. This paper guarantees end-to-end policies simultaneously for all the three aspects of trustworthiness. It presents a security-typed object-based language, a partitioning transformation, an operational semantics, and an information flow type inference system for partitioned and replicated classes. The type system provably guarantees that well-typed methods enjoy noninterference for the three properties, and that their types quantify their resilience to Byzantine attacks. Given a class and the specification of its end-to-end policies, the HAMRAZ tool applies type inference to automatically place and replicate the fields and methods of the class on Byzantine quorum systems, and synthesize trustworthy-by-construction distributed systems. The experiments show the resiliency of the resulting systems; they can gracefully tolerate attacks that are as strong as the specified policies.**

I. INTRODUCTION

Building trustworthy systems has been the holy grail of computing. The three desired properties, *confidentiality, integrity, and availability*, also known as the CIA triad, guarantee that sensitive data does not leak, computed results are correct, and the system remains accessible in the face of failures and attacks. Assurance of these properties is particularly needed when multiple principals with partial mutual trust cooperate. Inter-organizational systems are common: business-to-business procurement systems, medical information systems that integrate care-provider institutions, and joint military information systems. The distrust between the components of these systems leads to *distribution of data and computation* across administrative boundaries. However, building distributed systems that are resilient to both benign (crash) and malign (Byzantine) failures is notoriously complicated.

Given an integrated system, the ultimate question is whether it complies with system-wide trustworthiness policies. Furthermore, given the *end-to-end trustworthiness policies*, how can trustworthy-by-construction systems be automatically constructed? *Information flow control* [21], [47], [51], [55] can enforce end-to-end policies. To preserve confidentiality, it restricts the flow of information from the secret to the public domain. Further, to preserve integrity, a common technique is to compare multiple copies of data or computation against each other, and information flow analysis can check that enough copies are compared [58], [59]. However, this method can reduce availability as all the copies need to be available.

Of the three major aspects of trustworthiness, *availability* has been often dissociated from the others, and unfortunately, sidestepped. Confidentiality and integrity are safety properties but availability is a liveness property. In contrast to safety properties, simply monitoring the system and denying the violating actions cannot provide liveness. A few pioneering works consider information flow control for availability. However, they assume availability of the computation platform [60] or require the user to explicitly program the quorums [61].

Providing availability in the face of Byzantine failures [32] requires sophisticated *Byzantine quorum replication* protocols [12], [38]. A quorum system is a set of quorums such that each is an adequate set of hosts to perform operations. Quorum systems stay available even if only one of their quorums is not compromised. However, Byzantine replication has been largely regarded as a separate discipline. Further, existing protocols often provide guarantees only for a monolithic system based on assumptions on the Byzantine fraction of the processes. How can Byzantine replication be applied to general computation on integrated systems? Since hosts and polices of an integrated system are often heterogeneous, the deployed quorum systems should vary as the information flows from one computation and storage to another.

This paper *enforces end-to-end policies simultaneously for the three aspects of trustworthiness*, especially the *resiliency of availability*, in the face of *Byzantine attacks*. Further, given the end-to-end policies, it *automatically synthesizes* trustworthy-by-construction distributed systems that guarantee the specified policies. To this end, it presents a *security-typed object-based language*, a *partitioning transformation*, an *operational semantics*, and an *information flow type inference system* for partitioned and replicated classes.

We present a security-typed object-based language to describe classes that can encapsulate multiple field objects to implement their methods. The field objects abstract subsystems and the methods capture their interaction. The language allows the user to specify trustworthiness policies as type annotations. A *security type* consists of three components for the three trustworthiness properties. The space of types for each property is elegantly modeled as a lattice. The confidentiality type of a value is the set of hosts that are trusted to observe or store that value. We represent a failure or attack scenario as a set of principals, i.e., the Byzantine principals. Our novel representation of the *integrity* (and similarly *availability*) *type* of a value is a set of attack scenarios that the integrity (and availability) of the value is *resilient* to.

The language permits a class to be described as a centralized

definition with no distribution details. We present a high-level *sequential operational semantics* that model central executions. However, confidentiality types restrict the placement of objects and methods. In particular, a single principal may not be able to host the whole body of a method. Therefore, a method may need to be partitioned and hosted by multiple principals. We present a CPS (continuation-passing style) *transformation to partition* methods. Further, integrity and availability types require *replication* of fields and methods on Byzantine quorum systems. We present a *distributed operational semantics* that model the executions of partitioned and replicated classes. The semantics is parameterized for the placement and replication of the fields and methods.

We present a *type inference system* to enforce policies for the three trustworthiness properties: confidentiality, integrity and availability. It performs dependency analysis and rejects classes that violate the type specifications. In particular, if a method depends on results from another method, then the former can be at most as available as the latter. The type system provably guarantees that *well-typed methods have noninterference* for the three properties. For example, an expression does not access objects of higher confidentiality, lower integrity, or lower availability than its type. Further, *well-typed methods do not go wrong: they are resilient against Byzantine attacks that are as strong as their types*. In particular, if the Byzantine attack is no stronger than the integrity and availability type of a method, then any distributed execution of the method matches its sequential execution.

Given the placement and replication of field objects and methods, the type system can check their adequacy for the type specifications. More importantly, given the type specifications, the type inference system can derive constraints for the placement and replication. Transforming and solving these constraints yields the Byzantine quorum systems that host the fields and and methods of the class. This leads to *trustworthy-by-construction distributed systems*: the user describes the class with type annotations specifying the trustworthiness policies, and our tool, HAMRAZ, automatically synthesizes a distributed system that assures the policies. HAMRAZ can automatically construct hosting Byzantine quorum systems, and adjust them to the resiliency strength of type specifications. Experimental results on a cluster of nodes show that HAMRAZ generates resilient systems; the resulting systems can gracefully tolerate attacks that are as strong as the specifications.

We will start with an overview in § II. We see the programming model, the lattices of the security types, and quorum systems in § III. Then, we present the partitioning transformations in § IV. Next, we see the operational semantics, the type inference system, and the security guarantee theorems in § V, § VI, and § VII. Then, we consider constraint solving for type inference in § VIII. We describe the implementation and report experimental results in § IX. Finally, we discuss the related works in § X before we conclude in § XI.

II. OVERVIEW

In this section, we see a glimpse of security types, partitioning, and the inference of placement and replication.

One-time Transfer. We illustrate these concepts by a simple use-case: *one-time transfer*: Alice manages the set of servers (or principals) $A = \{p_{A1}, p_{A2}, .., p_{A7}\}$ and a register object r_1. Similarly, Bob manages the set of principals $B = \{p_{B1}, p_{B2}, .., p_{B4}\}$ and a register object r_2. The problem is to program a *transfer* method that lets a client principal p_0 (not Alice or Bob principals) choose to see *one of the two* registers, and reveals the value of that register to p_0 *only once*. The system should keep Alice's register *confidential* from Bob and vice versa. Further, there are *resiliency specifications for the integrity and availability* of the system. The system should maintain the integrity of the $transfer$ method to return the correct value even if two of Alice's principals and one of Bob's principals are Byzantine. It should also stay available even if one of Alice's principals and one of Bob's principals are Byzantine. Byzantine principals are compromised by the adversary and may behave arbitrarily. (A variant where Alice and Bob are not authorized to view the client's choice is called oblivious transfer and is a use-case in our experiments.)

Centralized Definition. The one-time transfer class $OneTimeTrans$ is shown in Fig. 1.(a). It is a *high-level centralized definition* with no extra details for distribution. The class has three field objects: Alice's register r_1, Bob's register r_2, and the register r that stores whether the client has already read a value. The $transfer$ method takes the client's choice as a boolean parameter x. It first checks the value of the register r. If it is true, the client has already read a value, and $transfer$ simply returns 0. Otherwise, it sets r to true, and depending on the parity of x, it reads and returns either the value of the register r_1 or r_2.

Security Types. The user can also annotate her program with security types. A type is the triple $\langle c, i, a\rangle$ of confidentiality c, integrity i and availability a types. (In order to focus the type system on trustworthiness, types do not capture the classical representation types such as Int.) The confidentiality type c of a value is the set of principals that are trusted to access the value. An integrity (and similarly availability) type represents resiliency against certain attacks. A *resiliency* value $\{\bar{b}\}$ characterizes all the attack scenarios of the system: for every execution of the system, there is a set b that contains all the Byzantine principals in that execution. The integrity type i of a value is defined as a resiliency $\{\bar{b}\}$ such that the value is correct even in the face of each Byzantine set b. Similarly, the availability type a of a value is a resiliency $\{\bar{b}\}$ such that the value is accessible even in the face of each b. For two resiliency values B and B', let their join $B \times_\cup B'$ be $\{b \cup b' \mid \langle b, b'\rangle \in B \times B'\}$. Types form a lattice with the weakest and strongest types $\bot$ and $\top$. In particular, a resiliency value B is stronger than (or can flow to) another B', written as $B \sqsubseteq B'$, if for any attack in B', there is a stronger attack in B. Let $\mathcal{P}_n(S)$ represent the set of subsets of the set S of cardinality n. For example, $\mathcal{P}_2(A)$ represents the attack

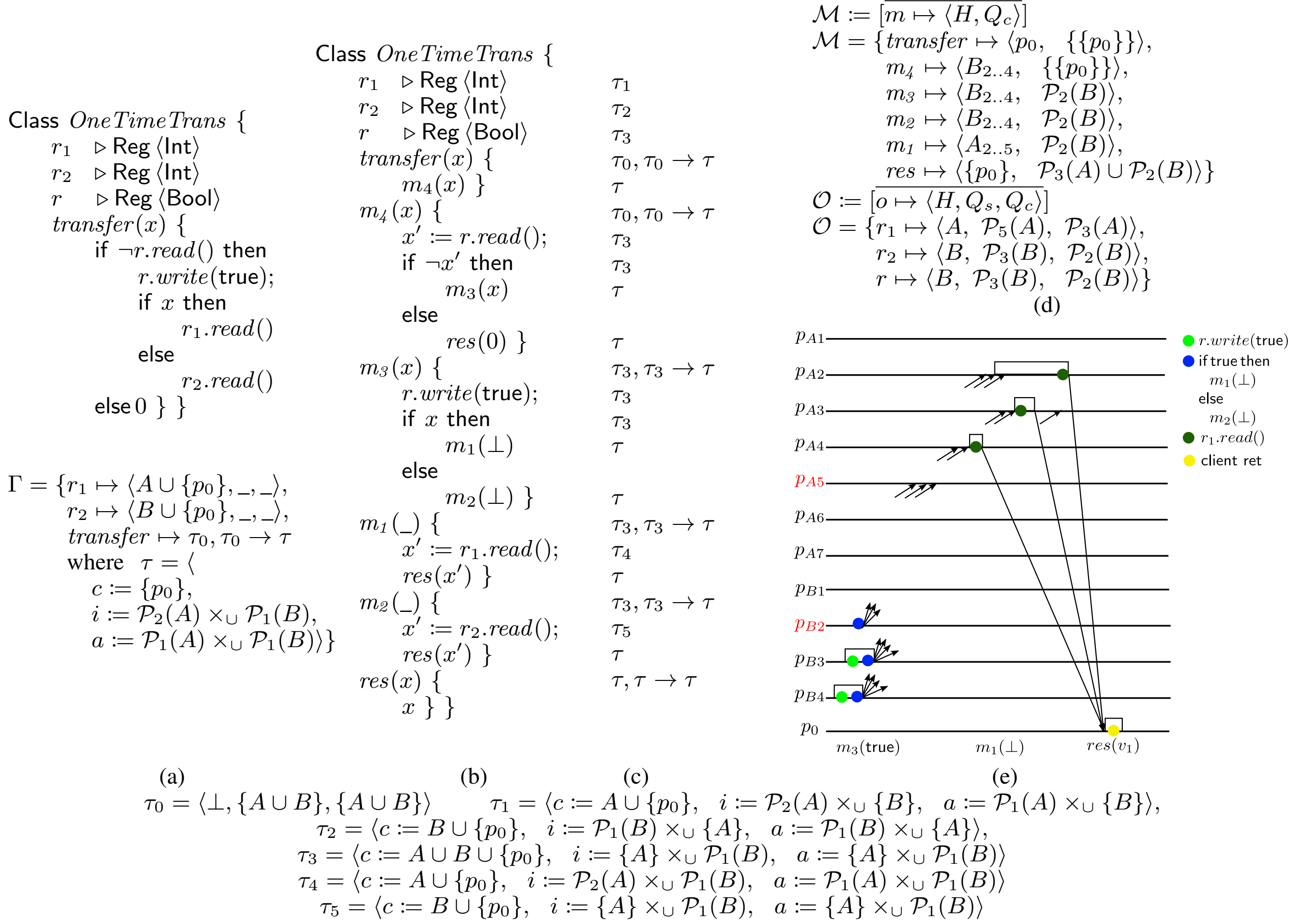

Fig. 1: One-Time Transfer. (a) User specification, (b) Partitioned class, (c) Typing, (d) Placement, (e) Execution. $A = \{p_{A1}, .., p_{A7}\}$, $B = \{p_{B1}, .., p_{B4}\}$. The set $\mathcal{P}_n(S)$ is the set of subsets of S of cardinality n. The set $S_{i..j}$ denotes $\{p_{Si}, .., p_{Sj}\}$.

scenario where two principals of the set A can be Byzantine.

Trustworthiness Policies. The user specifies the trustworthiness policies as type annotations. The type environment Γ in Fig. 1.(a) represents the *user type annotations* for the $\mathit{OneTimeTrans}$ class. The register r_1 should be confidential and accessible to only the set of principals A and the client p_0. Similarly, r_2 should be confidential for B and p_0. (The unspecified integrity and availability types are left for type inference as _.) The type of the *transfer* method is a function type $\tau_\mathsf{x}, \tau \rightarrow \tau'$ from the context type τ_x and the parameter type τ to the return type τ'. (As we will see later in § VI, the context type represents the trustworthiness of the calling context. Here, it is simply the type τ_0 as any context. Similarly, the parameter x is not confidential, has complete integrity and availability at the client p_0; thus, its type is simply τ_0.) The return type τ of *transfer* is more interesting. The confidentiality type c is $\{p_0\}$; only the client p_0 should be able to call the method. The integrity type i is $\mathcal{P}_2(A) \times_\cup \mathcal{P}_1(B)$; it requires the integrity of the return value to be resilient to two A and one B Byzantine principals. Similarly, the availability type a is $\mathcal{P}_1(A) \times_\cup \mathcal{P}_1(B)$; it requires the availability to be resilient to one A and one B Byzantine principals. The goal is to automatically partition and replicate the field objects and methods so that the above specifications are satisfied.

Partitioning. The method *transfer* calls methods on both registers r_1 and r_2. However, there is no principal in the sets A and B that is authorized to see the values of both registers. Therefore, in § IV, we *adapt the CPS transformation* to partition the methods of a class to smaller methods such that each makes at most one call to an object. The result of partitioning the *transfer* method of Fig. 1.(a) is presented in Fig. 1.(b). The *transfer* method is partitioned into six methods. The earlier methods call later ones as tail-calls. The initial method *transfer* and the response method *res* are both hosted at the client p_0 to invoke the call and to later receive the return value respectively. (For uniformity, methods with no parameters take a dummy parameter _.) It is critical that the two calls on the two registers r_1 and r_2 are partitioned into the two separate methods m_1 and m_2; the type inference places them on separate sets of A and B hosts.

Replication. To satisfy the resiliency specifications, the field objects and the methods of the class should be

sufficiently replicated. We apply Byzantine quorum systems for replication. A *quorum system* Q is a set of quorums; a quorum q is a set of principals that is adequate to properly perform operations. We use two types of quorum systems: *communication quorum systems* and *storage quorum systems*. We use the former to communicate and validate method call requests, and the latter to replicate field objects. The placement $\mathcal{M}(m)$ of each method m is a pair $\langle H, Q_c \rangle$: the method m is replicated on the set of hosts H that each execute a call on m if they receive the same call from the communication quorum system Q_c. The placement $\mathcal{O}(o)$ of each field object o is a triple $\langle H, Q_s, Q_c \rangle$: the object o is replicated on the set of hosts H with the the storage quorum system Q_s that executes a method call on o if it receives the same call from the communication quorum system Q_c.

Consider a quorum system $Q = \{\overline{q}\}$ and a set of Byzantine principals b. As we will see in § III, for the *availability of* Q, at least one q should not intersect with b. For the *integrity of* Q *as a storage system*, the intersection of every pair of quorums q_1 and q_2 should not be contained in b, and for the *integrity of* Q *as a communication system*, no q should be contained in b. The placements $\mathcal{O}$ and $\mathcal{M}$ for the $OneTimeTrans$ class are shown in Fig. 1.(d). (We will see below how these placements can be inferred from the specified policies.) The notation $S_{i..j}$ denotes the set $\{p_{Si}, .., p_{Sj}\}$. As an example, assume that we expect resiliency to two Byzantine principals in A. The storage quorum system Q_s for the register r_1 is $\mathcal{P}_5(A)$, subsets of A of size 5. The set A has 7 principals. Thus, there is always a quorum (a subset of A with size $7 - 2 = 5$) of non-Byzantine principals. Therefore, Q_s preserves its availability. Further, any pair of quorums intersect in at least $2 \times 5 - 7 = 3$ principals. Therefore, there is at least $3 - 2 = 1$ non-Byzantine principal in the intersection, and Q_s preserves its integrity. The communication quorum system Q_c for r_1 is $\mathcal{P}_3(A)$, subsets of A of size 3. Therefore, there is at least $3 - 2 = 1$ non-Byzantine principal in every quorum. Thus, every call request received from a quorum is valid, and Q_c preserves its integrity.

We now consider a distributed execution with the given placements, and then consider placement inference.

Replication Semantics. An example execution fragment from the method call $m_3(\mathsf{true})$ to m_1 and finally res is shown in Fig. 1.(e). The two principals p_{A5} and p_{B2} are Byzantine. The method m_3 is hosted on $B_{2..4}$. The non-Byzantine hosting principals of m_3 (i.e., p_{B3} and p_{B4}) execute $r.write(\mathsf{true})$ and the then branch of the subsequent if expression to call m_1. They send request messages to call m_1 to the hosting principals of m_1 that are $A_{2..5}$. A call to m_1 is executed only when the request is received from a quorum in $\mathcal{P}_2(B)$ (that is two B principals). Since there are enough non-Byzantine hosts for m_3, enough requests are received at hosts of m_1, and its non-Byzantine hosts (i.e., p_{A2}, p_{A3} and p_{A4}) execute m_1. The method m_1 reads the register r_1 and calls the method res with the read value. A call to res is executed only when the request is received from a quorum in $\mathcal{P}_3(A) \cup \mathcal{P}_2(B)$. In this case, three A principals make a quorum, and res is finally executed at the client p_0 with the value of the first register. We note that the quorums $\mathcal{P}_2(B)$ in the communication quorum system of res are used when the method m_2 calls res.

Type and Placement Inference. The user-specified type for the return value of $transfer$ is τ. Therefore, the return type and the parameter of the response method res are expected to be of type τ. Given the type specification of the method res, and the user type annotations in Γ, the type inference system can *infer the types* of the other methods. Further, it can *infer the placement* for the field objects and the methods. We will see the type inference system in § VI. The inferred type of each method and expression in Fig. 1.(b) is written in front of it in Fig. 1.(c). The inferred placements $\mathcal{O}$ and $\mathcal{M}$ are shown in Fig. 1.(d). We look at a few steps that infer the storage quorum system of the object r_1, the communication quorum system of the method res, and the hosts of the method m_1.

Storage Quorum Inference. The type of the parameter x of the method res is τ, The integrity and availability components of τ are $i = \mathcal{P}_2(A) \times_\cup \mathcal{P}_1(B)$, and $a = \mathcal{P}_1(A) \times_\cup \mathcal{P}_1(B)$. The method m_1 calls res with the argument x'. The integrity $i_{x'}$ of the argument x' should be stronger than the integrity i of the parameter x, i.e., $i_{x'} \sqsubseteq i$. The variable x' in m_1 binds the return value of a call to the object r_1. Therefore, the integrity i_{r_1} of r_1 should be stronger than the integrity of $i_{x'}$ of x', i.e., $i_{r_1} \sqsubseteq i_{x'}$. The integrity of r_1 is determined by the quorum system Q_s that stores it. By the transitivity of the above relations, the integrity i_{Q_s} of Q_s should be stronger than the integrity i above, i.e., $i_{Q_s} \sqsubseteq i$. A similar argument for the availability a_{Q_s} of Q_s yields $a_{Q_s} \sqsubseteq a$. Further, according to the confidentiality of r_1, Q_s should be stored on only the A principals. Therefore, the integrity of Q_s should be resilient to 2 Byzantine principals, and the availability of Q_s should be resilient to 1 Byzantine principal. What is the size s of the subset of A that hosts Q_s? Further, what is the quorum size n for Q_s? The quorum system Q_s will be $\mathcal{P}_n(A_{1..s})$. For integrity, the quorums should have a non-Byzantine intersection. Thus, we should have $2 \times n - s > 2$. For availability, there should be non-Byzantine quorum. Thus, we should have $s - 1 \geq n$. A solution to these constraints is $s = 7$ and $n = 5$. As Fig. 1.(d) shows, the storage quorum system Q_s for r_1 is $\mathcal{P}_5(A)$ (as $A_{1..7}$ is obviously equal to A).

Communication Quorum Inference. The integrity type of the parameter x of res is $i = \mathcal{P}_2(A) \times_\cup \mathcal{P}_1(B)$. The quorum system Q_c that receives calls to res should have stronger integrity than the parameter. Thus, its integrity should be resilient to 2 A and 1 B Byzantine principals. Therefore, the quorums in Q_c should have at least $2 + 1 = 3$ principals from A, or $1 + 1 = 2$ principals from B. Therefore, as the placement $\mathcal{M}$ in Fig. 1.(d) shows, Q_c is $\mathcal{P}_3(A) \cup \mathcal{P}_2(B)$.

Host Inference. Now, let us consider the hosting principals of m_1. Since m_1 calls a method on r_1, and r_1 is confidential for A, the method m_1 can be hosted on only A principals. The method m_1 calls res. As we just saw, the quorum system Q_c of res receives a call from A subsets of size 3. The type of the parameter x of res is τ, and the availability component of τ is $a = \mathcal{P}_1(A) \times_\cup \mathcal{P}_1(B)$. The parameter should be available if one principal in A is Byzantine. To satisfy the availability of

C	$:=$	$\langle \overline{o}, \overline{d} \rangle$	Class
o			Field Object
d	$:=$	$m(x) := e$	Method Definition
e	$:=$	$v \mid \bot \mid x \mid e \oplus e \mid x := e; e$	Expression
	$\mid$	$\text{if } e \text{ then } e \text{ else } e \mid m(e) \mid o.m(e)$	
p, h	$:$	P	Principal or Host
q, b, H	$:=$	$\mathcal{P}(P)$	Quorum, Byzantine set, Hosts
Q, B	$:=$	$\mathcal{P}(\mathcal{P}(P))$	Quorum System, Resiliency
$\mathcal{M}$	$:=$	$\overline{[m \mapsto \langle H, Q_c \rangle]}$	Method Placement
$\mathcal{O}$	$:=$	$\overline{[o \mapsto \langle H, Q_s, Q_c \rangle]}$	Object Placement
τ	$:=$	$\langle c, i, a \rangle$	Type
c	$:$	H	Confidentiality
i	$:$	B	Integrity
a	$:$	B	Availability

Fig. 2: Syntax. $\mathcal{P}(S) = 2^S$ is the power set of S.

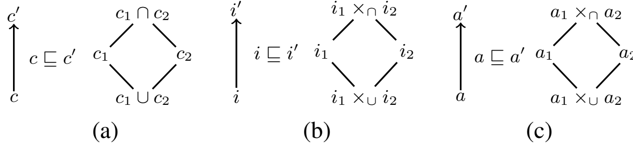

Fig. 3: Lattices for (a) Confidentiality (b) Integrity and (c) Availability. Arrows show the correct flow direction: more confidentiality, less integrity, and less availability.

the parameter of res, the method m_1 should send the argument from at least $3+1=4$ principals in A. Therefore, the hosting principals of m_1 are a subset of A of size 4. The placement shown in Fig. 1.(d) chose the set $A_{2..5}$.

III. Classes and Security Types

Class Definition. As shown in Fig. 2, a class $C = \langle \overline{o}, \overline{d} \rangle$ is described as a set of field objects o and method definitions d. A method definition $m(x) := e$ defines a method m with parameter x and the body e. An expression e is either an integer value v or none value $\bot$, a variable x, an operation $\oplus$ on expressions, a sequence expression $x := e; e'$ that evaluates e and bind its value to x for e', a conditional expression if, a this-method call $m(e)$ (a method call on the current object this of the enclosing class), or an object-method call $o.m(e)$. The sequence expression $e; e'$ is a syntactic sugar for a sequence whose bound variable is not free in e'. The language achieves Turing-completeness through recursive this-method calls.

Principal Sets. A principal (process or host) denoted as p (or h) from the universe of principals P can host both objects and methods. The identity of principals can be authenticated. A quorum q, Byzantine set b, or hosts H is a set of principals. A quorum system Q, or resiliency value B is a set of subsets of principals (such that none of the subsets is contained in another). A quorum system $\{\overline{q}\}$ is a set of quorums q such that each is adequate to perform operations. A resiliency value $\{\overline{b}\}$ characterizes all the attack scenarios of the system: for every execution, at least one of the b sets contains all the Byzantine principals in that execution. Byzantine principals are controlled by the adversary; they may not follow the user-defined programs and system protocols. (In the literature, a set $\{\overline{b}\}$ is called a failure-prone system [38] as well.)

We define the basic operators $\sqsubseteq$, $\times_\cup$ and $\times_\cap$ on quorum systems and resiliency values. A resiliency value B is stronger than (or can flow to) another B', written as $B \sqsubseteq B'$ iff for every set b' in B', there is a set b in B such that $b' \subseteq b$. We say that a Byzantine attack b is subsumed by a resiliency value B iff $B \sqsubseteq \{b\}$. Finally, $B \times_\cup B' = \{b \cup b' \mid \langle b, b' \rangle \in B \times B'\}$ and similarly, $B \times_\cap B' = \{b \cap b' \mid \langle b, b' \rangle \in B \times B'\}$.

Method and Object Placement. A method placement $\mathcal{M}$ is a mapping from each method m to a pair $\langle H, Q_c \rangle$ where H is the set of principals that host m, and Q_c is the communication quorum system for requests to call m. A host h in H executes a call to m only if it receives the call with the same argument from a quorum q in Q_c. An object placement $\mathcal{O}$ is a mapping from each object o to a triple $\langle H, Q_s, Q_c \rangle$ where H is the set of principals that hosts o, Q_s is the storage quorum system that serves calls to o, and Q_c is the communication quorum system to request calls on o. A method call on o is executed by a quorum in Q_s only if they receive the same call from a quorum in Q_c. (We will see more details of the operational semantics in § V.)

Security Types. A type τ is a tuple $\langle c, i, a \rangle$ where c is the confidentiality, i is the integrity, and a is the availability type.

Confidentiality. The confidentiality type c of a value is the set of principals that are trusted to access the value. A confidentiality type c is less than (or can flow to) another c', written as $c \sqsubseteq c'$, iff $c' \subseteq c$. As Fig. 3.(a) shows, information can flow from low confidentiality c to high confidentiality c'. Assume that the confidentiality type of x is $c_x = \{p_1, p_2\}$ and the confidentiality type of x' is $c_{x'} = \{p_1\}$. $c_x \sqsubseteq c_{x'}$. The flow from x to x' leaks no information, but the flow from x' to x can leak a secret in x' to p_2. Confidentiality types form a lattice where join $\sqcup$ is $\cap$, meet $\sqcap$ is $\cup$, $\bot$ is P and $\top$ in $\emptyset$.

Integrity. The integrity type i of a value is defined as a resiliency $\{\overline{b}\}$ such that the value is correct even in the face of each Byzantine attack b. We say that an integrity type i is stronger than (or can flow to) another i' if $i \sqsubseteq i'$. (We saw the definition of $\sqsubseteq$ on resiliency values above.) Intuitively, larger Byzantine sets b represent more integrity. As Fig. 3.(b) shows, information can flow from high integrity i to low integrity i'. Assume that the integrity type of x is $i_x = \{\{p_1\}, \{p_2\}\}$ and the integrity type of x' is $i_{x'} = \{\{p_1, p_2, p_3\}\}$. The variable x preserves its integrity even if p_1 or p_2 are Byzantine. The variable x' preserves its integrity even if p_1, p_2 and p_3 are Byzantine. Thus, $i_{x'} \sqsubseteq i_x$. The flow from x' to x preserves the integrity of x. However, the flow from x to x' can violate the integrity of x' if both p_1 and p_2 are Byzantine, or p_3 is Byzantine. Integrity types form a lattice where join $\sqcup$ is $\times_\cap$, meet $\sqcap$ is $\times_\cup$, $\bot$ is $\{P\}$ and $\top$ in $\{\emptyset\}$.

Availability. Similar to integrity, the availability type a of a value is defined as a resiliency $\{\overline{b}\}$ such that the value is accessible even in the face of each Byzantine attack b. We say that an availability type a is stronger than (or can flow to) another a' if $a \sqsubseteq a'$. As Fig. 3.(c) shows, information can flow from high availability a to low availability a'. Similar to

integrity, if we have $a_{x'} \sqsubseteq a_x$, the flow from x' to x preserves the availability of x but not vice versa. Availability types form a lattice where join $\sqcup$ is $\times_\cap$, meet $\sqcap$ is $\times_\cup$, $\bot$ is $\{P\}$ and $\top$ in $\{\emptyset\}$. We note that no resiliency is represented as $\{\emptyset\}$. An integrity or availability type is expected to be non-empty.

We represent and analyze integrity and availability types separately. However, an available value is often usable only if it has integrity. To assure availability of a correct value, the Byzantine set should be subsumed by both the integrity and availability types. Therefore, as Fig. 1.(a) shows, the integrity type is often stronger than the availability type.

Type. A type $\tau = \langle c, i, a \rangle$ is a subtype of another type $\tau' = \langle c', i', a' \rangle$, written as $\tau \sqsubseteq \tau'$, iff $c \sqsubseteq c'$, $i \sqsubseteq i'$ and $a \sqsubseteq a'$. We note that with the lattice (and flow) direction defined for integrity and availability above, all the three type components are co-variant. Intuitively, the super-type has more confidentiality, less integrity and less availability. A type can be implicitly up-cast to a super-type. If $\tau_x \sqsubseteq \tau_{x'}$ then it is safe for the data from x to flow into x'. Types form a lattice with the expected point-wise definitions for $\sqcup$, $\sqcap$, $\bot$ and $\top$ on the lattices of their three components.

Resiliency of Quorum Systems. First, we consider the integrity of communication and storage in turn. Then, we consider the availability of quorum systems.

Integrity of Communication Quorum Systems. Communication quorum systems are used to deliver a message to a target principal. Sender principals echo the message, and the target principal delivers it only if it receives the same message from a quorum. Thanks to the redundancy in the messages, the delivered message has integrity even if only one of the senders is a non-Byzantine principal. The integrity of a communication quorum system $Q = \{\overline{q}\}$, written as $\mathit{CIntegrity}(Q)$, is the resiliency $B = \{\overline{b}\}$ where the Byzantine sets b are the maximal subsets of the set of principals P such that for each b, there is no quorum q that is a subset of b.

Integrity of Storage Quorum Systems. Storage quorum systems are used to store and retrieve objects. To store a value for the object, at least a quorum should store it, and to retrieve its value, at least a quorum should retrieve the same value. In order to retrieve the latest stored value, it is crucial that the two quorums have a non-Byzantine principal in their intersection. The integrity of a storage quorum system $Q = \{\overline{q}\}$, written as $\mathit{SIntegrity}(Q)$, is the resiliency $B = \{\overline{b}\}$ where the Byzantine sets b are the maximal subsets of the set of principals P such that for each b, the intersection of every pair of quorums q_1 and q_2 is not a subset of b.

Availability of Quorum Systems. Given a set of hosts H for a quorum system $Q = \{\overline{q}\}$ and a set of Byzantine principals b, consider a quorum q that is tasked with the execution of an operation. The quorum q can perform the operation if all of its members are in H and none of them are in b. Therefore, Q is available if at least one of its quorums q is a subset of H and doesn't intersect b. The availability of a set of hosts H for a quorum system $Q = \{\overline{q}\}$, written as $\mathit{Availability}(Q, H)$, is the resiliency $B = \{\overline{b}\}$ where the Byzantine sets b are the maximal subsets of the set of principals P such that for each b, there is at least one quorum q that is a subset of H and doesn't intersect b.

We note that as a quorum system is a set of quorums, the classical labels that represent one set of principals are not enough to capture its integrity and availability.

IV. Partitioning

A method of a class can execute multiple calls on its field objects. A principal that hosts a method should be more confidential than all the objects that it accesses. Such a principal might not exist. For example, in our running example, the one-time transfer class $\mathit{OneTimeTrans}$ in Fig. 1.(a), the method $\mathit{transfer}$ calls methods on both objects r_1 and r_2. However, there is no principal (except the client) that is confidential enough to access both objects. Further, placing more methods on more confidential principals can cause imbalance for the computation load across principals. Therefore, as the first step for resilient replication, we partition the methods of the class into smaller methods such that each method makes at most one object-method call. When methods make at most one object-method call, they provide maximum flexibility for their placement and replication during the type inference.

We perform partitioning in two steps: we first factor the object-method calls by an adaptation of the CPS transformation, and then split the methods.

Factoring Object-method Calls. We first transform expressions e (that we saw in Fig. 2) to factored expressions c as defined in Fig. 4.(a). In a factored expression c, object-method calls are lifted as explicit call expressions $\mathsf{call}\, x := o.m(f)$ where the expressions f are free of object-method calls. As an example, Fig. 4.(b) shows the factored representation of the body of the $\mathit{transfer}$ method that we saw in Fig. 1.(a). The object-method call $r.\mathit{read}()$ is lifted to the beginning of the body. Similarly, $r_1.\mathit{read}()$ and $r_2.\mathit{read}()$ are lifted to the beginning of their enclosing branches of the if statement. Factoring is performed in two steps: First, we perform a CPS transformation to make object-method calls and their evaluation order explicit. Object-method calls are translated to explicit call expressions, and their call-by-value order of evaluation is made explicit as nested call expressions. Second, once the call expressions are captured, we apply β-reduction to remove redundant lambda abstractions.

CPS Transformation. The CPS transformation is presented in Fig. 4.(c). The rules for values v and $\bot$, variables x, and operations $\oplus$ are straightforward. The rule for the if expression applies the transformation to the condition and the branches in order. The rule for the sequence expressions $x := e_1; e_2$ first evaluates the first expression e_1, and then passes its result as x to the second expression e_2 and evaluates it. The rule for this-method calls $m(e)$ evaluates the argument e before calling the method m. Similarly, the rule for object-method calls $o.m(e)$ evaluates the argument e first. More importantly, it converts $o.m(e)$ to an explicit call expression $\mathsf{call}\ x_2 := o.m(x_1)\ \mathsf{in}\ k\, x_2$ instead of directly passing it to the continuation k as $k\, o.m(x_1)$. The explicit call expressions preserve object-method calls and their order after β-reductions.

$$
\begin{array}{rcll}
f & := & v \mid \bot & \text{Integer Literal} \\
 & \mid & x & \text{Variable} \\
 & \mid & f \oplus f & \text{Operation} \\
 & \mid & \mathsf{if}\ f\ \mathsf{then}\ f\ \mathsf{else}\ f & \text{Conditional} \\
 & \mid & m(f) & \text{This call} \\
c & := & \mathsf{call}\ x := o.m(f)\ \mathsf{in}\ c & \text{Method Call} \\
 & \mid & \mathsf{if}\ f\ \mathsf{then}\ c\ \mathsf{else}\ c & \text{Conditional} \\
 & \mid & f & \text{Expression}
\end{array}
$$

(a)

$$
\begin{array}{l}
\mathsf{call}\ x_1 := r.read()\ \mathsf{in} \\
\mathsf{if}\ \neg x_1\ \mathsf{then} \\
\quad \mathsf{call}\ _ := r.write(\mathsf{true})\ \mathsf{in} \\
\quad \mathsf{if}\ x\ \mathsf{then} \\
\quad\quad \mathsf{call}\ x_2 := r_1.read()\ \mathsf{in} \\
\quad\quad res(x_2) \\
\quad \mathsf{else} \\
\quad\quad \mathsf{call}\ x_3 := r_2.read()\ \mathsf{in} \\
\quad\quad res(x_3) \\
\mathsf{else}\ res(0)
\end{array}
$$

(b)

$$
\begin{array}{rcl}
[\![v]\!]\, k & = & k\, v \\
[\![\bot]\!]\, k & = & k\, \bot \\
[\![x]\!]\, k & = & k\, x \\
[\![e_1 \oplus e_2]\!]\, k & = & [\![e_1]\!]\, (\lambda x_1.\, [\![e_2]\!]\, (\lambda x_2.\, k\, (x_1 \oplus x_2))) \\
[\![\mathsf{if}\ e_0 & & \\
\mathsf{then}\ e_1 & = & [\![e_0]\!]\, (\lambda x.\, \mathsf{if}\ x\ \mathsf{then}\ [\![e_1]\!]\, k \\
\mathsf{else}\ e_2]\!]\, k & & \qquad\qquad \mathsf{else}\ [\![e_2]\!]\, k \\
[\![x := e_1; e_2]\!]\, k & = & [\![e_1]\!]\, (\lambda x.\, [\![e_2]\!](\lambda x_2.\, k\, x_2)) \\
[\![m(e)]\!]\, k & = & [\![e]\!]\, (\lambda x.\ k\ m(x)) \\
[\![o.m(e)]\!]\, k & = & [\![e]\!]\, (\lambda x_1.\, \mathsf{call}\ x_2 := o.m(x_1)\ \mathsf{in}\ k\, x_2)
\end{array}
$$

(c)

Fig. 4: Factoring. (a) Factored expressions. (b) Example factored expression $[\![transfer(x)]\!]\ res$. (c) CPS transformation.

$$
\textsc{Call}\quad \frac{[\![c]\!] \triangleright \langle f', X', D' \rangle \qquad \mathsf{fresh}\, m' \qquad X'' = \mathsf{FV}(f) \cup X' \setminus x}{\begin{array}{c}[\![\mathsf{call}\ x := o.m(f)\ \mathsf{in}\ c]\!] \triangleright \\ \langle m'(X''),\ X'',\ D' \cup \{m'(X'') := (x := o.m(f); f')\} \rangle\end{array}}
$$

$$
\textsc{If}\quad \frac{[\![c_1]\!] \triangleright \langle f_1, X_1, D_1 \rangle \qquad [\![c_2]\!] \triangleright \langle f_2, X_2, D_2 \rangle}{\begin{array}{c}[\![\mathsf{if}\ f\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] \triangleright \\ \langle \mathsf{if}\ f\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2, \mathsf{FV}(f) \cup X_1 \cup X_2, D_1 \cup D_2 \rangle\end{array}}
\qquad
\textsc{FExp}\quad [\![f]\!] \triangleright \langle f, \mathsf{FV}(f), \emptyset \rangle
$$

$$
\textsc{Method}\quad \frac{[\![e]\!] \triangleright \langle e', _, D \rangle}{[\![m(x) := e]\!] \triangleright D \cup \{m(x) := e'\}}
\qquad
\textsc{Class}\quad [\![\langle \overline{o}, \overline{d} \rangle]\!] \triangleright \langle \overline{o}, \cup \overline{[\![d]\!]} \rangle
$$

Fig. 5: Splitting Transformation

Reduction. The CPS transformation introduces *administrative* lambda abstractions to pass continuations and make the evaluation order explicit. After the transformation, we apply β-reduction to remove the administrative lambda abstractions and restore a concise representation.

$$\beta\text{-REDUCTION} \qquad (\lambda x.\, e)\, e' \rightarrow e[e'/x]$$

We note that the β-reduction cannot rewrite call expressions. Consider the expression $\mathsf{call}\ x_2 := o.m(x_1)\ \mathsf{in}\ k\, x_2$. The β-reduction cannot replace x_2 with $o.m(x_1)$. Therefore, even if x_2 is not used in k, the object-method call $o.m(x_1)$ is not removed. Further, the evaluation order of calls is preserved.

Splitting This-methods. Given a class such that the body of the this-methods are factored expressions c, this step splits each this-method into multiple this-methods such that each one calls at most one object-method call.

Fig. 5 presents the splitting transformation. The judgments that translate a factored expression c are of the form $[\![c]\!] \triangleright \langle f, X, D \rangle$ where f is the resulting expression, X is the set of free variables of f, and D is the set of generated this-method definitions that f transitively calls. The rule CALL translates a call expression $\mathsf{call}\ x := o.m(f)\ \mathsf{in}\ c$. It first translates c to f' with free variables X' and this-methods D'. It then generates a new this-method m' with the sequence $x := o.m(f); f'$ as the body. We note that the generated this-method includes only one object-method call. The free variable X'' in the body are the free variables of f and the free variables of f' (i.e., X') except x that is bound by the call. The free variables X'' should be passed as parameters to m'. Thus, the call expression is translated to the this-method call $m'(X'')$. If the translated call expression is part of a larger expression, the resulting this-method call $m'(X'')$ can inductively become a leaf expression of the body of the this-method that is generated for the larger expression. Thus, calls to the generated this-methods appear only as tail-calls. The rule IF inductively splits the branches of the if expression and results in an if statement that is free of object-method calls. The rule FEXP simply translates an f expression to the tuple of itself, its free variables $FV(f)$ and no new this-methods. The rule METHOD splits a this-method, and the rule CLASS splits each this-method of a class.

As an example, applying the splitting transformation to the factored expression in Fig. 4.(b) results in the split methods in Fig. 1.(b). Starting from the leaf expressions, the two branches of the inner if expression are split to the two methods m_1 and m_2. This split is crucial to the enforcement of the confidentiality policies. The objects r_1 and r_2 can be accessed by only the principals A and B respectively. After the split, the methods m_1 and m_2 can be separately placed on A and B principals. Next, the then branch of the outer if expression is split to the method m_3. We note that there is only one object-method call at the beginning of each generated method. Next, the outer if expression is split to the method m_4 where the object-method call $r.read()$ is at the top. Finally, the body of the top-level this-method *transfer* is translated to a call to m_4.

We note that partitioning allows maximum flexibility for placement inference; however, if a caller and a callee are placed on the same hosts, the call can be inlined. We also note that partitioning results in non-blocking methods where a call to the method immediately returns, and the return value is later passed by a callback method (e.g. *res*). If needed, a non-blocking method can be made blocking by simply waiting for the callback using standard synchronization mechanisms.

V. OPERATIONAL SEMANTICS

In the previous section, we saw how the methods of a class are partitioned. In this section, we present the operational semantics of these classes. We first present a baseline sequential semantics and then present a distributed semantics that replicates both objects and methods, and marshals call requests between quorums of hosts.

Sequential Semantics. The state of the operational semantics is defined in Fig. 6. For the sequential semantics, the state

is the pair $\langle e, S\rangle$ where e is an expression, and S is a mapping from objects o to their encapsulated states s. A reduction context $\mathcal{R}$ captures the next expression to be reduced. The sequential operational semantics is presented in Fig. 7. The transitions are straightforward. We saw in the previous section that after partitioning, this-method calls appear only as tail-calls. Thus, in the rule STHISCALL, this-method calls do not need a context $\mathcal{R}$. For brevity, we use _ in the place of symbols that are unused and stay the same in the transition. (In order to factor the reduction context, an extra rule could be added. For the next semantics, yet another rule would be needed to factor the context for multiple principals. The current representation with explicit contexts seems to be more concise.)

Distributed Semantics. The distributed state is the triple $\langle \mathbb{P}, \mathbb{S}, \mathbb{N}\rangle$. As defined in Fig. 6, the principal states $\mathbb{P}$ is a map from principals to their states. The state of a principal is the expression e that it is executing, the identifier id of the call that is being executed, and the number of calls n that have been executed by this call. Consider the call tree rooted at the initial method call. The unique identifier of a call is the list of the branch numbers in the path from the initial method call to that call. The object state $\mathbb{S}$ is a map from objects o to pairs $\langle s, r\rangle$ where s is the state of o, and r is the recorded calls, a map from call identifiers to their return values. Consider a this-method $m()$ that calls an object-method $o.m'()$. Since $m()$ is replicated on multiple principals, multiple requests to execute $o.m'()$ with the same identifier can be issued. To avoid duplicate execution of object-method calls, the storage quorum system of an object o not only keeps its state s but also a record r of the previously executed calls and their return values. The network $\mathbb{N}$ keeps pending request messages to execute this-method calls. It is a mapping from tuples $\langle p, id, m, v\rangle$ to a set of principals q (or $\perp$), where p is the receiver principal, id is the identifier of the call, m is the method requested to be called, v is the argument, and q is the set of sender principals that requested the call. The value of the mapping is $\perp$ when the requested call is already processed and should not be reprocessed. A transition label and a sequence of labels are denoted as l and L respectively.

Fig. 8 defines the distributed operational semantics that models the runtime system. It is parametric in terms of a class $C = \langle \overline{o}, \overline{d}\rangle$, the method and object placements $\mathcal{M}$ and $\mathcal{O}$, and the set of Byzantine hosts $\mathcal{B}$. We say that the resiliency of a system is $\{\overline{b}\}$ if it is resilient in all executions of all instantiations of the operational semantics with a $\mathcal{B}$ where $\{\overline{b}\} \sqsubseteq \{\mathcal{B}\}$, that is a $\mathcal{B}$ that are a subset of a b. Thus, in the theorems of § VII, $\mathcal{B}$ is universally quantified.

The rules OP, SEQ, IFTHEN and IFELSE make local transitions and are similar to their sequential counterparts.

This-method calls. The rule THISCALL reduces a this-method call $m(v)$ on a principal p. Let the identifier of the method call that is currently being executed be id, and the number of calls that it has executed be n. Thus, the identifier of the new method call is $id :: n$. Let the placement of m be the set of principals $\{\overline{p'}\}$. A request message from the sender p to execute the method m with identifier $id :: n$ and

$$
\begin{array}{rcll}
\langle e, S\rangle & & & \text{Sequential State} \\
\mathcal{R} & := & \mathcal{R} \oplus e \mid v \oplus \mathcal{R} \mid x := \mathcal{R};\ e & \text{Red. Context} \\
& \mid & \mathsf{if}\ \mathcal{R}\ \mathsf{then}\ e\ \mathsf{else}\ e \mid m(\mathcal{R}) \mid o.m(\mathcal{R}) \mid [] & \\
s & & & \text{Object State} \\
S & := & [\overline{o \mapsto s}] & \text{Seq. Object States} \\
\langle \mathbb{P}, \mathbb{S}, \mathbb{N}\rangle & & & \text{Distributed State} \\
\mathbb{P} & := & [\overline{p \mapsto \langle e, id, n\rangle}] & \text{Principal States} \\
id & : & \mathsf{list\ nat} & \text{Method call identifier} \\
\mathbb{S} & := & [\overline{o \mapsto \langle s, r\rangle}] & \text{Dist. Object States} \\
r & := & [\overline{id \mapsto v}] & \text{Method calls record} \\
\mathbb{N} & := & [\overline{\langle p, id, m, v\rangle \mapsto q \mid \perp}] & \text{Network} \\
l & := & p \mid \overline{p} \mid \langle p, id, m, v\rangle & \text{Label} \\
& \mid & \langle p, p, m, id, v\rangle \mid \langle p, v\rangle & \\
L & := & l^* & \text{Labels}
\end{array}
$$

Fig. 6: Runtime State

SOP
$$\frac{v_1 \oplus v_2 = v_3}{\langle \mathcal{R}[v_1 \oplus v_2], _\rangle \to \langle \mathcal{R}[v_3], _\rangle}$$

SSEQ
$$\langle \mathcal{R}[x := v; e], _\rangle \to \langle \mathcal{R}[e[v/x]], _\rangle$$

SIFTHEN
$$\langle \mathcal{R}[\mathsf{if}\ 1\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2], _\rangle \to \langle \mathcal{R}[e_1], _\rangle$$

SIFELSE
$$\langle \mathcal{R}[\mathsf{if}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2], _\rangle \to \langle \mathcal{R}[e_2], _\rangle$$

STHISCALL
$$\frac{(m(x) := e) \in \overline{d}}{\langle m(v), _\rangle \to \langle e[v/x], _\rangle}$$

SOBJCALL
$$\frac{S(o) = s \qquad m(s, v) = \langle s', v'\rangle}{\langle \mathcal{R}[o.m(v)], S\rangle \to \langle \mathcal{R}[v'], S[o \mapsto s']\rangle}$$

Fig. 7: Sequential Operational Semantics

argument v is sent to every receiver p'. For each receiver p', the principal p is added to the set of senders q. As we saw in § IV, after partitioning, this-method calls appear only as tail-calls. Thus, they transfer control to the hosts of the callee, and the new expression of the hosting principal p of the caller becomes the none value $\perp$ which denotes that p can execute another this-method call. We saw an example of this-method calls in Fig. 1.(e). The non-Byzantine hosting principals of m_3 (i.e., p_{B3} and p_{B4}) issue a call to m_1. They send call request messages to the hosting principals of m_1 (i.e., $A_{2..5}$).

The rule THISCALLEXEC executes a this-method call if request messages from a quorum are received. The receiving principal p is not processing any calls as its current expression is none $\perp$. Let the communication quorum system for the method m be $\{\overline{q}\}$. If the set of requesting principals q' to call $m(v)$ (with the same identifier) is a superset of a quorum q, the current expression of p becomes the body e of m after the parameter x is substituted with the argument v. Finally, the processed request message is mapped to $\perp$ to prevent duplicate executions. We saw an example in Fig. 1.(e). The communication quorum system for m_1 is $\mathcal{P}_2(B)$; a quorum should have at least two B principals. The non-Byzantine hosts of m_1 (i.e., p_{A2}, p_{A3} and p_{A4}) executed m_1, since each received the request from a quorum.

The rule THISCALLBYZ models the behavior of Byzantine principals that can arbitrarily change their state, and send arbitrary call requests to arbitrary principals. However, thanks to authentication upon receiving messages, they cannot send a

Op
$$\frac{v_1 \oplus v_2 = v_3}{\langle \mathbb{P}[p \mapsto \langle \mathcal{R}[v_1 \oplus v_2], _, _\rangle], _, _\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \mathcal{R}[v_3], _, _\rangle], _, _\rangle}$$

Seq
$$\langle \mathbb{P}[p \mapsto \langle \mathcal{R}[x := v; e], _, _\rangle], _, _\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \mathcal{R}[e[v/x]], _, _\rangle], _, _\rangle$$

IfThen
$$\langle \mathbb{P}[p \mapsto \langle \mathcal{R}[\mathsf{if}\ 1\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2], _, _\rangle], _, _\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \mathcal{R}[e_1], _, _\rangle], _, _\rangle$$

IfElse
$$\langle \mathbb{P}[p \mapsto \langle \mathcal{R}[\mathsf{if}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2], _, _\rangle], _, _\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \mathcal{R}[e_2], _, _\rangle], _, _\rangle$$

ThisCall
$$\frac{\mathcal{M}(m) = \langle \{\overline{p'}\}, _\rangle \quad \overline{\mathbb{N}(p', id :: n, m, v) = q} \quad \mathbb{N}' = \mathbb{N}[\overline{(p', id :: n, m, v) \mapsto q \cup \{p\}}]}{\langle \mathbb{P}[p \mapsto \langle m(v), id, n\rangle], _, \mathbb{N}\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \bot, id, n+1\rangle], _, \mathbb{N}'\rangle}$$

ThisCallExec
$$\frac{\mathcal{M}(m) = \langle _, \{\overline{q}\}\rangle \quad \mathbb{N}(p, id, m, v) = q' \quad q \subseteq q' \quad (m(x) := e) \in \overline{d} \quad \mathbb{N}' = \mathbb{N}[(p, id, m, v) \mapsto \bot]}{\langle \mathbb{P}[p \mapsto \langle \bot, _, _\rangle], _, \mathbb{N}\rangle \overset{\langle p, id, m, v\rangle}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle e[v/x], id, 0\rangle], _, \mathbb{N}'\rangle}$$

ThisCallByz
$$\frac{p \in \mathcal{B} \quad \mathbb{N}(p', id', m, v) = q' \quad \mathbb{N}' = \mathbb{N}[(p', id', m, v) \mapsto q' \cup \{p\}]}{\langle \mathbb{P}[p \mapsto \langle e, id, n\rangle], _, \mathbb{N}\rangle \overset{\langle p, p', id', m, v\rangle}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle e', id', n'\rangle], _, \mathbb{N}'\rangle}$$

ObjCall
$$\frac{\begin{array}{c}\mathcal{O}(o) = \langle H, Q_s, Q_c\rangle \quad Q_c = \{\overline{q}\} \quad q \subseteq \{\overline{p}\} \quad \mathit{SIntegrity}(Q_s) \sqsubseteq \{\mathcal{B}\} \quad \mathit{Availability}(Q_s, H) \sqsubseteq \{\mathcal{B}\} \\ \mathbb{S}(o) = \langle s, r\rangle \quad id :: n \notin \mathsf{dom}(r) \quad m(s, v) = \langle s', v'\rangle \quad \mathbb{S}' = \mathbb{S}[o \mapsto \langle s', r[id :: n \mapsto v']\rangle]\end{array}}{\langle \mathbb{P}[\overline{p \mapsto \langle \mathcal{R}[o.m(v)], id, n\rangle}], \mathbb{S}, _\rangle \overset{\overline{p}}{\rightsquigarrow} \langle \mathbb{P}[\overline{p \mapsto \langle \mathcal{R}[v'], id, n+1\rangle}], \mathbb{S}', _\rangle}$$

ObjReCall
$$\frac{\begin{array}{c}\mathcal{O}(o) = \langle H, Q_s, _\rangle \quad \mathit{SIntegrity}(Q_s) \sqsubseteq \{\mathcal{B}\} \\ \mathit{Availability}(Q_s, H) \sqsubseteq \{\mathcal{B}\} \quad \mathbb{S}(o) = \langle s, r\rangle \quad r(id :: n) = v'\end{array}}{\langle \mathbb{P}[p \mapsto \langle \mathcal{R}[o.m(v)], id, n\rangle], \mathbb{S}, _\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \mathcal{R}[v'], id, n+1\rangle], \mathbb{S}, _\rangle}$$

ObjCallByz
$$\frac{\mathcal{O}(o) = \langle _, Q_s, _\rangle \quad \mathit{SIntegrity}(Q_s) \not\sqsubseteq \{\mathcal{B}\}}{\langle \mathbb{P}[p \mapsto \langle \mathcal{R}[o.m(v)], _, _\rangle], _, _\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \mathcal{R}[v'], _, _\rangle], _, _\rangle}$$

Fig. 8: Distributed Operational Semantics. It is parametric in terms of the class $C = \langle \overline{o}, \overline{d}\rangle$, the method and object placements $\mathcal{M}$ and $\mathcal{O}$, and the Byzantine principals $\mathcal{B}$. In the ThisCall rule, the union operator $\cup$ is extended for $\bot$ values: $\bot \cup s = \bot$.

message on behalf of another principal. (A step by a Byzantine principal to impersonate another Byzantine principal can be simulated as two consecutive steps by the two principals.)

Object-method calls. The rule ObjCall executes an object-method call $o.m(v)$. Let the storage quorum system of o be Q_s, and the communication quorum system of o be $Q_c = \{\overline{q}\}$. If (1) a set of principals $\{\overline{p}\}$ that are a superset of a communication quorum q call the object-method, (2) the set of Byzantine principals $\mathcal{B}$ cannot compromise the integrity and availability of the storage system Q_s, and (3) the method call is not already executed, i.e., it is not in the recorded calls r, then the method call is executed on the current state s of o. The resulting state s' is stored, and the return value v' is recorded in r for the identifier of the call. The method call is evaluated to the value v in each of the principals in $\{\overline{p}\}$. Object-method calls block for the return value, and are unblocked once a quorum of principals make the same call, and the call is executed. The rule ObjReCall reduces an object-method call that is already executed (when the storage system is not compromised). It retrieves the return value from the recorded calls r. The rule ObjCallByz models the execution of an object-method call when the set of Byzantine principals is large enough to compromise the integrity of the storage system. In this case, the call returns an arbitrary value.

VI. Information Flow Type System

In this section, we present the information flow type inference system and its guarantees. Instances of a well-typed class preserve their type specifications for confidentiality, integrity and availability at run time.

We present the type inference system in Fig. 9. Given a class C, it yields constraints $\mathcal{C}$ on types and placements. The judgments are of the form $\Gamma, \mathcal{O}, \mathcal{M} \vdash C, \mathcal{C}$ that states that the class C is well-typed under the type environment Γ, and the placements $\mathcal{O}$ and $\mathcal{M}$ for the field objects and methods of C, if the constraints $\mathcal{C}$ are satisfied. (The judgments of the corresponding type *checking* system is $\Gamma, \mathcal{O}, \mathcal{M} \vdash C$. Instead of yielding constraints, it would simply check the same conditions.) The typing judgments for an object field o and a method definition d are $\Gamma, \mathcal{O}, \mathcal{M} \vdash o, \mathcal{C}$ and $\Gamma, \mathcal{O}, \mathcal{M} \vdash d, \mathcal{C}$ respectively. The typing judgments for an expression e is $\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e, \mathcal{C}$ where $\mathcal{H}$ is the set of hosts that replicate the execution of e, and τ_{x} is the type of the context under which e is executed. The context type $\tau_{\mathsf{x}} = \langle c_{\mathsf{x}}, i_{\mathsf{x}}, a_{\mathsf{x}}\rangle$ captures the implicit information flow. The context confidentiality type c_{x} represents the information that can be learned from the fact that the execution has reached the current expression e. Similarly, the context integrity type i_{x} represents the integrity of the information that determines the control flow to the current expression e. The conditions of the if expressions that enclose e determine its context confidentiality and integrity types. The context availability type a_{x} represents the availability of the information that the control flow requires to reach the current expression e. The conditions of the if expressions that enclose e, and the sequence expressions that precede e determine its context availability type.

The type environment Γ is a mapping from variables x and objects o to types τ, from this-methods m to function types $\tau_{\mathsf{x}}, \tau \rightarrow \tau'$, and from object-methods $o.m$ to function types $\tau \rightarrow \tau'$, where τ_{x} is the context type, τ is the parameter type, and τ' is the return type. The interface that objects expose can

VALT
$$\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash v : \bot,\, \emptyset$$

VART
$$\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash x : \Gamma(x),\, \emptyset$$

OPT
$$\frac{\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e_i : \tau_i,\, \mathcal{C}_i \quad \text{for } i \in \{1, 2\} \qquad \text{fresh } \tau \quad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \sqcup \tau_2 \sqsubseteq \tau\}}{\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e_1 \oplus e_2 : \tau,\ \mathcal{C}}$$

SEQT
$$\frac{\begin{array}{c}\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e_1 : \tau_1,\, \mathcal{C}_1 \quad \tau_{\mathsf{x}} = \langle c_{\mathsf{x}}, i_{\mathsf{x}}, a_{\mathsf{x}} \rangle \quad \tau_1 = \langle _, _, a_1 \rangle \\ \Gamma[x \mapsto \tau_1], \mathcal{O}, \mathcal{M}, \mathcal{H}, \langle c_{\mathsf{x}}, i_{\mathsf{x}}, a_{\mathsf{x}} \sqcup a_1 \rangle \vdash e_2 : \tau_2,\, \mathcal{C}_2 \\ \text{fresh } \tau \quad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \sqcup \tau_2 \sqsubseteq \tau\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash x \coloneqq e_1; e_2 : \tau,\ \mathcal{C}}$$

IFT
$$\frac{\begin{array}{c}\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e_0 : \tau_0,\, \mathcal{C}_0 \\ \Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \sqcup \tau_0 \vdash e_i : \tau_i,\, \mathcal{C}_i \quad \text{for } i \in \{1, 2\} \\ \text{fresh } \tau \quad \mathcal{C} = \mathcal{C}_0 \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_0 \sqcup \tau_1 \sqcup \tau_2 \sqsubseteq \tau\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau,\, \mathcal{C}}$$

THISCALLT
$$\frac{\begin{array}{c}\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e : \tau,\, \mathcal{C} \quad \Gamma(m) = \tau'_{\mathsf{x}}, \tau_1 \to \tau_2 \quad \tau_1 = \langle _, _, a_1 \rangle \\ \mathcal{M}(m) = \langle _, Q \rangle \quad \mathcal{C}' = \mathcal{C} \cup \{\tau_{\mathsf{x}} \sqsubseteq \tau'_{\mathsf{x}}, \\ \tau \sqcup \tau_{\mathsf{x}} \sqsubseteq \tau_1, \quad \mathit{Availability}(Q, \mathcal{H}) \sqsubseteq a_1\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash m(e) : \tau_2,\, \mathcal{C}'}$$

OBJCALLT
$$\frac{\begin{array}{c}\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e : \tau,\, \mathcal{C} \quad \Gamma(o.m) = \tau_1 \to \tau_2 \quad \tau_1 = \langle _, _, a_1 \rangle \\ \tau_2 = \langle c_2, _, _ \rangle \quad \mathcal{O}(o) = \langle _, _, Q \rangle \quad \mathcal{C}' = \mathcal{C} \cup \{c_2 \sqsubseteq \mathcal{H}, \\ \tau \sqcup \tau_{\mathsf{x}} \sqsubseteq \tau_1, \quad \mathit{Availability}(Q, \mathcal{H}) \sqsubseteq a_1\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash o.m(e) : \tau_2,\, \mathcal{C}'}$$

METHODT
$$\frac{\begin{array}{c}\mathcal{M}(m) = \langle H, Q_c \rangle \quad \Gamma(m) = \tau_{\mathsf{x}}, \tau_1 \to \tau_2 \quad \tau_{\mathsf{x}} = \langle c_{\mathsf{x}}, _, _ \rangle \\ \tau_1 = \langle c_1, i_1, _ \rangle \quad \Gamma[x \mapsto \tau_1], \mathcal{O}, \mathcal{M}, H, \tau_{\mathsf{x}} \vdash e : \tau,\, \mathcal{C} \\ \mathcal{C}' = \mathcal{C} \cup \{\tau \sqsubseteq \tau_2, \quad \tau_1 \sqsubseteq \tau_2, \quad c_1 \sqcup c_{\mathsf{x}} \sqsubseteq H, \quad \mathit{CIntegrity}(Q_c) \sqsubseteq i_1\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M} \vdash m(x) \coloneqq e,\, \mathcal{C}'}$$

FIELDT
$$\frac{\begin{array}{c}\mathcal{O}(o) = \langle H, Q_s, Q_c \rangle \quad \Gamma(o) = \langle c, i, a \rangle \quad M(o) = \overline{m} \\ \overline{\Gamma(o.m) = \langle c_m, i_m, a_m \rangle \to \langle c'_m, i'_m, a'_m \rangle} \\ \overline{\mathcal{C}_m = \{\langle c, i, a \rangle \sqsubseteq \langle c_m, i_m, a_m \rangle \sqsubseteq \langle c'_m, i'_m, a'_m \rangle\}} \\ \mathcal{C} = \overline{\cup \mathcal{C}_m} \cup \{\overline{\sqcup c'_m} \sqsubseteq H \quad \mathit{SIntegrity}(Q_s) \sqsubseteq \overline{\sqcap i'_m}, \\ \mathit{Availability}(Q_s, H) \sqsubseteq \overline{\sqcap a'_m}, \quad \mathit{CIntegrity}(Q_c) \sqsubseteq \overline{\sqcap i_m}\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M} \vdash o,\, \mathcal{C}}$$

CLASST
$$\frac{\overline{\Gamma, \mathcal{O}, \mathcal{M} \vdash o,\, \mathcal{C}} \qquad \overline{\Gamma, \mathcal{O}, \mathcal{M} \vdash m,\, \mathcal{C}'}}{\Gamma, \mathcal{O}, \mathcal{M} \vdash \langle \overline{o}, \overline{m} \rangle,\, \overline{\wedge \mathcal{C}} \wedge \overline{\wedge \mathcal{C}'}}$$

Fig. 9: Information Flow Type Inference System

be called from any context. Thus, the type of their context parameter is $\top$ and is elided in an object interface.

Values, Variables, Operations, and Sequences. The rule VALT simply type-checks a value v as $\bot$, and the rule VART type-checks a variable x according to the environment Γ. The rule OPT type-checks an operation $e_1 \oplus e_2$ as (a super-type of) the join of the types of the operands. Similarly, The rule SEQT type-checks a sequence $x \coloneqq e_1;\, e_2$ as the join of the types of the two operands. However, to type-check e_2, the type environment maps x to the type of e_1, and the availability of the context is reduced by the availability of e_1. The intuition is that e_2 cannot be evaluated if e_1 is unavailable.

Conditionals. The rule IFT type-checks a conditional expression as the join of the types of the condition and the branch expressions. To type-check the branch expressions, the given context type τ_{x} is joined with the type τ_0 of the condition expression e_0. The intuition is that e_0 implicitly flows to the branches. The fact that a branch is executed can leak the value of e_0. Therefore, the context confidentiality is increased by the confidentiality of τ_0. Further, the choice of the right branch is dependent on the integrity of e_0. Therefore, the context integrity is reduced by the integrity of τ_0. Further, the branches cannot be evaluated if e_0 is unavailable. Therefore, the context availability is reduced by the availability of τ_0.

Method Calls. The rule THISCALLT type-checks a this-method call $m(e)$. It first type-checks the argument e as τ, and then retrieves the type $\tau'_{\mathsf{x}}, \tau_1 \to \tau_2$ of m from the environment Γ, where τ'_{x} is the context parameter type, $\tau_1 = \langle _, _, a_1 \rangle$ is the parameter type, and τ_2 is the return type of the method. It then checks that the current context type τ_{x} is a subtype of the context parameter type τ'_{x}. It also checks that the argument type τ is a subtype of the parameter type τ_1. In addition, it checks that the current context type τ_{x} is a subtype of the parameter type τ_1. The intuition is that the argument can implicitly flow confidential information from the context (e.g. the enclosing conditionals) to the callee, and the integrity and availability of the context can affect the integrity and availability of the argument. Finally, the rule checks that the current hosts $\mathcal{H}$ that send the argument for the call to m meet the availability specification a_1 of the parameter. More precisely, let Q be the communication quorum system that an argument for a call to m is accepted from. If Q is restricted to $\mathcal{H}$, then it should be more available than a_1.

The rule OBJCALLT type-checks an object-method call $o.m(e)$. It is similar in structure to THISCALLT with two differences. First, it does not include the constraint on context types since the context parameter type of an object-method is implicitly $\top$. Second, the hosts $\mathcal{H}$ should be confidential enough to observe the return value.

Methods. The rule METHODT type-checks a method definition $m(x) \coloneqq e$. It first retrieves the type $\tau_{\mathsf{x}}, \tau_1 \to \tau_2$ of m from the given environment Γ. It then type-checks the body e under the context type τ_{x}, and the environment Γ extended with x typed as τ_1. The resulting type τ of e has to be a subtype of the return type τ_2. Further, the parameter type τ_1 has to be a subtype of the return type τ_2. This makes a this-method call have a stronger type than its argument. Further, the hosting principals H should be more confidential than c_1 and c_{x} because the hosts can learn confidential information in the argument and about the context from the fact that the call is made. Finally, the communication quorum system that accepts the arguments for m should provide more integrity than the integrity i_1 that the parameter expects.

Field Objects. The rule FIELDT checks the following conditions for an object o. (1) Let $\langle c, i, a \rangle$ be the type of o in

the environment Γ. The rule checks that c is a lower bound for the confidentiality, and i and a are higher bounds for the integrity and availability of the parameters and return values of the methods m of o. (These bounds are used to state the non-interference properties in the next section.) (2) Let H be the hosts for the the storage quorum system Q_s of o. (2.1) The rule checks that the hosting principals H are confidential enough to host the methods of o. More precisely, it checks that the join (i.e., intersection) of the confidentiality of the return values c'_m is less than (i.e., is a superset of) H. (2.2) Further, in order to host o, Q_s should provide more integrity and availability than the integrity i'_m and availability a'_m that the return value of each method m is expected to have. (3) Let Q_c be the communication quorum system that the arguments of method calls on o are accepted from. Q_c should provide more integrity than the integrity i_m that the parameter of each method m expects.

Class. The rule CLASST type-checks a class $\langle \overline{o}, \overline{m} \rangle$ by type-checking each object o and method m.

VII. Security and Resiliency Guarantees

The type system guarantees non-interference for confidentiality, integrity and availability of methods of well-typed classes. Further, their integrity and availability types characterize their resilience to Byzantine attacks. In this section, we will see that if a method is typed, it enjoys non-interference from objects of super-types, and resilience to Byzantine attacks of sub-types. After basic definitions, we first look at the non-interference theorems and then the resilience theorems. The proofs are available in the appendix § XIV.

The partitioning process splits methods to a sequence of methods. In the initial state, the client invokes the first method in that sequence that we call the initial method.

Definition 1 (Initial method): The initial (or client) method m_0 is a method that is hosted on a non-Byzantine (client) principal p_0 and can be directly invoked by p_0. More precisely, if $\mathcal{B}$ denotes the set of Byzantine principals, and $\mathcal{M}$ denotes the method placement, then $p_0 \notin \mathcal{B}$ and $\mathcal{M}(m_0) = \langle \{p_0\}, \{\{p_0\}\} \rangle$.

Definition 2 (Initial distributed state): In the initial distributed state, p_0 calls m_0 with the initial argument v_0, the objects have their initial states, and the set of messages in the system is empty. More precisely, the initial distributed state is $\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset \rangle$ where $\mathbb{P}_0 = \overline{[p \mapsto \langle \bot, 0, 0 \rangle]}[p_0 \mapsto \langle m_0(v_0), 0, 0 \rangle]$, $\mathbb{S}_0 = \overline{[o \mapsto \langle S_0(o), \emptyset \rangle]}$, where $S_0(o)$ is the initial state of o.

To state properties over only a subset of objects in the maps $\mathcal{O}$ and $\mathbb{S}$, we project them over the three type kinds.

Definition 3 (Projection over types): Given a typing environment Γ, and a map M on the objects domain, the projection of M over a confidentiality c restricts the domain of M to objects with confidentiality *less than* c in Γ. More precisely, $M \,|_\Gamma\, c = M \,|\, \{o \mid \mathsf{let}\ \langle c', _, _ \rangle := \Gamma(o)\ \mathsf{in}\ c' \sqsubseteq c\}$. Similarly, the projection of M over an integrity i restricts the domain of m to objects with integrity *more than* i. The projection over an availability a is similarly to objects of more availability than a. More, precisely, $M \,|_\Gamma\, i = M \,|\, \{o \mid \mathsf{let}\ \langle _, i', _ \rangle := \Gamma(o)\ \mathsf{in}\ i' \sqsubseteq i\}$ and $M \,|_\Gamma\, a = M \,|\, \{o \mid \mathsf{let}\ \langle _, _, a' \rangle := \Gamma(o)\ \mathsf{in}\ a' \sqsubseteq a\}$.

We lift the integrity of storage quorum systems, *SIntegrity*, to object placements $\mathcal{O}$, as the join of *SIntegrity* of the storage systems of all the objects in $\mathcal{O}$. More precisely, $\mathit{SIntegrity}(\overline{[o \mapsto \langle _, Q, _ \rangle]}) = \overline{\sqcup\, \mathit{SIntegrity}(Q)}$

Non-Interference. The type that the type system associates with a method m captures the trustworthiness of the objects that it accesses. If the return type of m is τ, then m accesses only objects that are typed as sub-types of τ, and enjoys non-interference from objects that are typed as super-types of τ. In fact, there is non-interference even if the super-type relation holds only for one of the three type components. If the type system associates a confidentiality type with (the return value of) m, then calls to m don't access *objects of higher confidentiality*. Similarly, if the type system associates an integrity or availability type with m, then calls to m don't access *objects of lower integrity or availability*. Changing the state of these objects doesn't interfere with the return value.

Confidentiality. Assume that the client method is type-checked with the confidentiality type c. Let O_c be the objects that are less (or as) confidential than c. The following non-interference theorem states that if two state maps $\mathbb{S}_1$ and $\mathbb{S}_2$ have the same states for the objects O_c, and the integrity of O_c is not compromised by the Byzantine principals $\mathcal{B}$, then any two executions with $\mathbb{S}_1$ and $\mathbb{S}_2$ that return a value to the client, return the same value. The integrity of the objects is required since a compromised object that loses integrity can behave non-deterministically.

Theorem 1 (Non-interference): For all Γ, $\mathcal{O}$, $\mathcal{M}$, C, $\mathcal{B}$, c, i, a, $\mathbb{S}_1$, $\mathbb{S}_2$, $\mathbb{P}'_1$, $\mathbb{P}'_2$, L, v and v', if

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$, and $\Gamma(m_0) = _, _ \to \langle c, i, a \rangle$, and either

- $\mathbb{S}_1 \,|_\Gamma\, c = \mathbb{S}_2 \,|_\Gamma\, c$, and $\mathit{SIntegrity}(\mathcal{O} \,|_\Gamma\, c) \sqsubseteq \{\mathcal{B}\}$, or
- $\mathbb{S}_1 \,|_\Gamma\, i = \mathbb{S}_2 \,|_\Gamma\, i$, and $\mathit{SIntegrity}(\mathcal{O} \,|_\Gamma\, i) \sqsubseteq \{\mathcal{B}\}$, or
- $\mathbb{S}_1 \,|_\Gamma\, a = \mathbb{S}_2 \,|_\Gamma\, a$, and $\mathit{SIntegrity}(\mathcal{O} \,|_\Gamma\, a) \sqsubseteq \{\mathcal{B}\}$, or

$\langle \mathbb{P}_0, \mathbb{S}_1, \emptyset \rangle \overset{L}{\rightsquigarrow}{}^* \langle \mathbb{P}'_1, _, _ \rangle$, and $\langle \mathbb{P}_0, \mathbb{S}_2, \emptyset \rangle \overset{L}{\rightsquigarrow}{}^* \langle \mathbb{P}'_2, _, _ \rangle$,
$\mathbb{P}'_1(p_0) = \langle v, _, _ \rangle$, and $\mathbb{P}'_2(p_0) = \langle v', _, _ \rangle$,

then $v = v'$.

The proof for confidentiality is by induction on the steps. Every step preserves the invariant that both the expressions in non-Byzantine principals, and the requested this-method calls in messages are less confidential than c. Thus, only objects that are less confidential than c are accessed. Further, these objects are assumed to have the same state in $\mathbb{S}_1$ and $\mathbb{S}_2$, and preserve integrity. Therefore, object-method calls behave deterministically and return the same value in the two executions, which in turn preserves the equality of the expression of every non-Byzantine principal in the two executions.

Integrity. Assume that the client method is type-checked with the integrity type i. Let O_i be the objects with integrity more than i. The above non-interference theorem states that if two state maps $\mathbb{S}_1$ and $\mathbb{S}_2$ have the same states for the objects O_i, and the integrity of O_i is not compromised by the Byzantine principals $\mathcal{B}$, then any two executions with $\mathbb{S}_1$ and $\mathbb{S}_2$ that return a value to the client, return the same value.

Availability. Similarly, the above theorem states non-interference for availability. If a method is type-checked with the availability type a, then different states for objects that are less available than a cannot interfere with the return value.

Resilience. Well-typed classes are resilient to Byzantine principals. In particular, the integrity and availability types that the type system associates with a method characterize the resilience of its integrity and availability to Byzantine principals. The integrity of the method is resilient to any Byzantine attack that is subsumed by the integrity type of (the return value of) the method. Similarly, the availability of the method is resilient to any Byzantine attack that is subsumed by its integrity and availability types.

Integrity Resiliency. If the *Byzantine principals are subsumed by the integrity type*, then *the results of distributed executions is the same value as the sequential execution*. More precisely, if the sequential semantics evaluates the client method call to the value v, and the set of Byzantine principals $\mathcal{B}$ is subsumed by the integrity type i of (the return value of) the method, then any distributed execution of the method call that results in a value, results in v as well. For example, in Fig. 1.(a) and (b), the integrity of the return type τ of *transfer* is $\mathcal{P}_2(A) \times_\cup \mathcal{P}_1(B)$. Therefore, the result of a distributed execution of *transfer* is the same as its sequential execution even if two A and one B principals are Byzantine.

Theorem 2 (Integrity Resilience): For all v, Γ, $\mathcal{O}$, $\mathcal{M}$, C, i, $\mathcal{B}$, $\mathbb{P}$, and v', if

$\langle m_0(v_0), S_0 \rangle \to^* \langle v, _ \rangle$,

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$, $\Gamma(m_0) = _, _ \to \langle _, i, _ \rangle$, and $i \sqsubseteq \{\mathcal{B}\}$,

$\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset \rangle \rightsquigarrow^* \langle \mathbb{P}, _, _ \rangle$, and $\mathbb{P}(p_0) = \langle v', _, _ \rangle$, $v' \neq \bot$,

then $v' = v$.

The proof is by induction on macro-steps where a step is taken by all replicating principals before the next step is taken. For every execution, there is a corresponding macro-step execution. A macro-step preserves the invariant that the expression of every non-Byzantine principal has more integrity than i, and there is a sequential execution from the initial call to that expression. In the case for execution of a this-method call, we show that the request is received from at least one non-Byzantine principal and use the invariant for that principal.

Availability Resiliency. If the *Byzantine principals are subsumed by the integrity and availability types*, then a *distributed execution can make progress and results in the same value as the sequential execution*. More precisely, if the sequential semantics evaluates the client method call to the value v, and the set of Byzantine principals $\mathcal{B}$ is subsumed by the integrity i and availability a types of (the return value of) the method, then a distributed execution results in v as well. For example, in Fig. 1.(a) and (b), the integrity and availability of the return type τ of *transfer* are $\mathcal{P}_2(A) \times_\cup \mathcal{P}_1(B)$, and $\mathcal{P}_1(A) \times_\cup \mathcal{P}_1(B)$ respectively. Therefore, a distributed execution of *transfer* results in the same value as its sequential execution, even if one A and one B principals are Byzantine.

Theorem 3 (Availability Resilience): For all v, Γ, $\mathcal{O}$, $\mathcal{M}$, C, i, a, $\mathcal{B}$, and $\mathbb{P}$, if

$\langle m_0(v_0), S_0 \rangle \to^* \langle v, _ \rangle$,

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$, $\Gamma(m_0) = _, _ \to \langle _, i, a \rangle$, and $i \sqcup a \sqsubseteq \{\mathcal{B}\}$,

then there exists $\mathbb{P}$ such that

$\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset \rangle \rightsquigarrow^* \langle \mathbb{P}, _, _ \rangle$, and $\mathbb{P}(p_0) = \langle v, _, _ \rangle$.

The set of Byzantine principals should be subsumed by not only the availability type a but also the integrity type i so that the storage quorum systems of the objects keep their quorum intersection and return sound values. Compromised return values can drift the execution to paths that do not match the sequential execution, and can even lead to non-termination.

We note that while Theorem 1 states the safety property that a typed method does not access objects that are less available than its type, Theorem 3 states the liveness property that it makes progress despite Byzantine principals that are not as strong as its integrity and availability types.

VIII. Constraint Solving

We translate the constraints $\mathcal{C}$ to the theory of linear arithmetic. Let the number of principal classes (trust domains) be n and let P_j denote the set of principals of class j. For example, in Fig. 1, n is 3, and the principal classes P_0, P_1 and P_2 are A, B and $C = \{p_0\}$. We represent a confidentiality value c as a tuple $\langle c_0, c_1, .., c_{n-1} \rangle$ where each c_j represents an integer variable with the value 1 or 0. The principal class j is trusted or untrusted if the value of c_j is 1 or 0 respectively.

We represent a set of hosting principals H as subsets of given sizes of the principal classes. The hosting principals H are represented as a tuple where H_j represents the number of hosting principal from the class j. Similarly, we represent a quorum system Q or a resiliency value B as subsets of certain sizes of the principal classes. For example, in Fig. 1.(a), the availability of τ is $\mathcal{P}_2(A) \times_\cup \mathcal{P}_1(B)$ that is all the subsets with 2 principals from A and 1 principals from B. This can be succinctly represented as the tuple $\langle 2, 1, 0 \rangle$. In general, we represent the set of subsets $s = \mathcal{P}_{s_0}(P_0) \times_\cup \mathcal{P}_{s_1}(P_1) \times_\cup .. \times_\cup \mathcal{P}_{s_{n-1}}(P_{n-1})$ as $\langle s_0, s_1, .., s_{n-1} \rangle$. We let a quorum system Q (or resiliency value B) be the union of n such sets: $Q = \cup_{i \in \{0,1,..,n-1\}} \langle Q_{ij_0}, Q_{ij_1}, .., Q_{ij_{n-1}} \rangle$. The indices i range over n tuples. In the tuple with index i, the number of principals from class j is Q_{ij}. Having n such tuples keeps the space of quorums tractable, and is expressive enough to capture common quorum systems. For example, the quorum system $\mathcal{P}_4(P_0) \cup \mathcal{P}_5(P_1) \cup \mathcal{P}_2(P_2)$ is represented as $\langle 4, 0, 0 \rangle \cup \langle 0, 5, 0 \rangle \cup \langle 0, 0, 2 \rangle$ that is all the subsets of size 4 from P_0, of size 5 from P_1, and of size 2 from P_2. Thus, each quorum system Q or resiliency value B can be represented by n^2 variables.

We now define the translation of two constraints. (The translation of all the constraints is available in the appendix.) Each constraint is translated to a constraint of size at most $O(n^4)$.

$$CIntegrity(Q) \sqsubseteq B \triangleright \bigwedge_{i} \bigwedge_{i'} \bigvee_{j} Q_{ij} > B_{i'j}$$

$$Availability(Q, H) \sqsubseteq B \triangleright \bigwedge_{i'} \bigvee_{i} \bigwedge_{j} Q_{ij} > 0 \to Q_{ij} \leq H_j - B_{i'j}$$

In the first rule, the assertion states that none of the quorums of Q is contained in a Byzantine set of B. More precisely, for

Table I: Partitioning and Type Inference.

	PT (s)	CN	GT (ms)	ST (s)	TT (s)
One-time Transfer	0.03	181	10.6	6.08	6.12
Ticket System	1.18	525	12.60	214.46	215.65
Oblivious Transfer	0.04	203	14.6	9.96	10.01
Auction	36.43	681	10.60	22.53	58.98
Friend Map	0.03	302	13.2	359.94	359.98
Salary Sharing	26.74	611	14.2	160.38	187.13

PT: Partitioning time, CN: Constraint number, GT: Constraint generation time, ST: Constraint solving time, TT: Total time

each tuple i in Q, and tuple i' in B, for at least one of the indices (principal classes) j, Q_{ij} is more than $B_{i'j}$. In the second rule, the assertion states that for every Byzantine set b in B, there is at least one quorum q in Q that falls inside the hosts H and doesn't intersect b. More precisely, for each tuple i' in B, there is a tuple i in Q such that for all indices j, if Q_{ij} is non-zero, it is less than or equal to H_j minus $B_{i'j}$.

We minimize the number principals in the host sets and quorum systems to reduce the load. Let $\mathcal{M} = [\overline{m \mapsto \langle H, Q\rangle}]$ and $\mathcal{O} = [\overline{o \mapsto \langle H', Q', Q''\rangle}]$. The optimization constraint is $\min \overline{\sum_j H_j + H'_j + \sum_i \sum_j Q_{ij} + Q'_{ij} + Q''_{ij}}$ that minimizes the size of hosts and quorums. For example, in our running example, the one-time transfer, for the given specification in Fig. 1.(a), type inference can find the correct placement that we saw in Fig. 1.(d). If we add the minimization constraint, the more efficient placement $\langle A_{1..5},\ \mathcal{P}_4(A_{1..5}),\ \mathcal{P}_3(A)\rangle$ for r_1 can be found; it has 5 hosts and quorums of size 4.

IX. Implementation and Experiments

We developed a tool called HAMRAZ and experimented with multiple resiliency specifications for six use-cases on a cluster of nodes. The experiments show that HAMRAZ can successfully infer the placements and replications for the given resiliency specifications. Further, it generates systems that can gracefully tolerate injected faults that are as strong as the resiliency specifications, and it can adjust the level of replication according to the resiliency specifications. We make HAMRAZ publicly available as open-source software.

Implementation. HAMRAZ is implemented in Java in two parts: the synthesizer and the runtime. The synthesizer closely follows the partitioning and type inference system of § IV and § VI, and the runtime closely follows the distributed operational semantics of § V. We used the Z3 SMT solver (v. 4.8.10) [20] for constraint solving and optimization. We implemented communication quorum systems on top of SSL StartTLS protocols from the Netty library [1], storage quorum systems for field objects on top of the BFT-SMaRt library [8].

Platform. The experiments are done on a high-performance cluster with Intel Broadwell CPUs with 4 cores, 2GB of RAM and CentOS-7 Linux x86_64 3.10.0. JDK is OpenJDK RE 18.9. The runtime is executed for each principal on a separate node of the cluster. The nodes are connected with 56 Gb/s InfiniBand. Each reported number is the arithmetic mean of 5 repetitions. Each repetition is the average of the response time for 150 method calls.

Use-cases. We experiment with six use-cases: one-time transfer that we saw in § II, oblivious transfer [58], ticketing system, auction system, friend map [35] and privacy-preserving salary averaging [37].

Consider the sets of principals A to Z. In our plots, we concisely represent a resiliency specification $\mathcal{P}_i(A) \times_\cup .. \times_\cup \mathcal{P}_k(Z)$ as $\langle i, .., k\rangle$ @ n where n is the total number of principals that is $(3 \times i + 1) + .. + (3 \times k + 1)$ plus 1 for the client. We use the same resiliency specification for both integrity and availability.

Partitioning and Type Inference. HAMRAZ successfully partitioned and inferred the placement and replication for each field object and method of the above use-cases. Table I shows the detailed execution times. The process takes less than 6 minutes. We see that the process is often dominated by constraint solving. However, when the use-case has a large number of object-method calls (e.g. the Auction use-case), the CPS transformation takes longer and the partitioning time is a larger faction.

Increasing Faults. In this experiment, we validate the hypothesis that HAMRAZ generates replicated systems that are as resilient as the specifications require. In this experiment (the first row of Fig. 10), the specification is fixed and is written in the plot of each use-case. We consider the effect of increasing the number of injected faults on the response time. Injected faults are randomly selected from three types: crash fault, corrupted payload, and delayed response. In these plots, we increase the number of faults for the principals (a) A, (b) A and B at the same time, (c) A, (d) A, and (e) C, from 0 to their specified resiliency. In plots (a)-(d), the failing principals host objects but in plot (e), the failing principals host only methods. Each plot shows three lines for three fault injection scenarios: solid: failing only leaders, dotted: failing only followers, and dashed: failing randomly.

In all the failure scenarios across the use-cases, the resulting systems can gracefully tolerate the injected faults. Random failures increase the response time between 3 and 42 percent. Further, we observe that tolerating failing leaders is considerably slower than failing followers due to leader reconfiguration. We observe in plots (a)-(d) that increasing the faults increases the response time. On the other hand, in plot (e), the increase in response time is negligible (overlapping lines). This is due to the fact that in contrast to principals that host objects (plots (a)-(d)), tolerating failure of principals that host methods (plot (e)) does not require reconfiguration.

Increasing Resiliency. In the following two experiments, we validate the hypothesis that HAMRAZ can adjust replication according to the strength of the specification. In the second row of Fig. 10, we consider the effect of increasing resiliency on the response time in normal (solid line) and faulty executions (dashed line) with maximum number of faults. Higher resiliency requires more replicas and more communication between them that affects the overall response time. If a fixed number of replicas was used, the response time would be flat.

Increasing the Load. In the auction use-case, if the initial offer o is less than 350, then A immediately wins with the offer $o - 1$. Otherwise, the two agents beat each other's offers $o - 350$ times until A offers 349 and wins. Fig. 11 shows the response time for increased initial offers. Larger initial offers

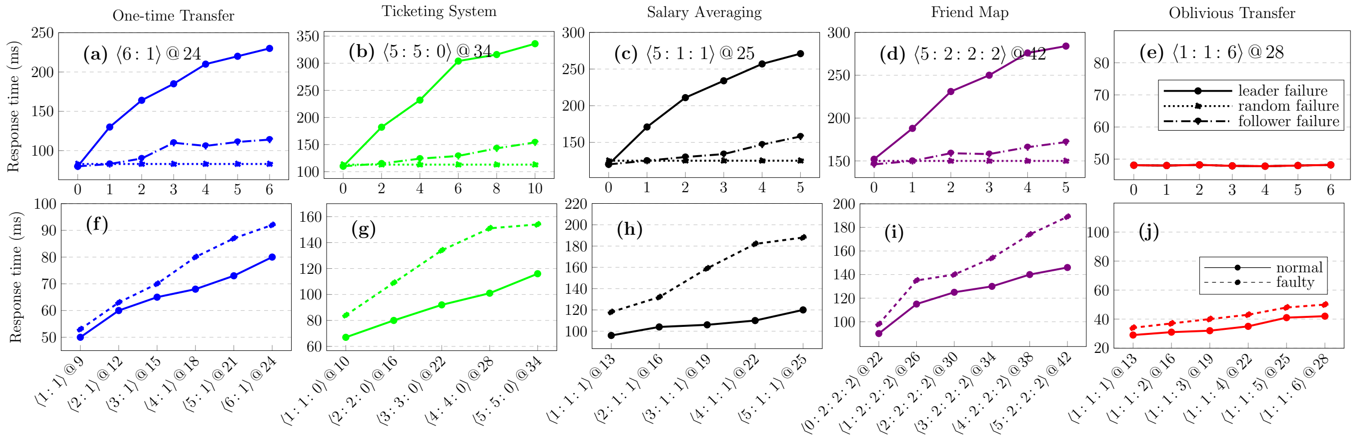

Fig. 10: Top row: Response time for increased faults. Bottom row: Response time for increased resiliency. $\mathcal{P}_i(A) \times_\cup .. \times_\cup \mathcal{P}_k(Z)$ is denoted as $\langle i, .., k \rangle$ @ n where n is the total number of principals.

lead to more object-calls and increased load. We experimented with three specifications where the resiliency for A and B is doubled from one to the next. We observe that as the load increases, the response time of more resilient systems grows slightly faster. More resilient system require more principals and more coordination.

X. Related Works

Information flow control [5], [7], [11], [21], [29], [40], [46], [47], [51], [55] has been widely used to enforce confidentiality and integrity policies. It has been applied to concurrent [52] and distributed [50] programs on trusted hosts. Further, Fabric [35] supports programming distributed systems on heterogeneously trusted hosts, and enforces confidentiality and integrity types, but doesn't provide Byzantine replication and doesn't enforce availability policies. Several previous works [9], [14]–[16], [41], [42], [54] automatically partition applications to multiple tiers, often to the web server and client tiers, and enforce confidentiality and integrity, but not availability. Jif/split [58], [59] partitions programs and replicates code partitions and data. It can replicate commitments instead of cleartexts to increase integrity without reducing confidentiality. Further, its secure communication assumptions between partitions were later lifted by a cryptographic back-end [23]. PtrSplit [36] splits programs with C++ pointers. However, these projects do not provide availability; in fact, Jif/split may reduce availability as all replicas need to be available. A related work [6] synthesizes cryptographic implementations for distributed applications; however, it does not consider availability polices.

Later, information flow type systems were applied to enforce availability policies [60] but assumed availability of the computation platform and did not consider Byzantine-resilient replication and type lattices. Similarly, RMS [34] adjusts the placement and replication of objects based on availability and performance specifications; however, it does not tolerate Byzantine failures. Qimp [61] provides a language construct for clients to run an expression on references at a specified quorum, and type-checks availability guarantees. (In addition, when it is provable that integrity is compromised, it can use a default value to provide low integrity but high availability.) In contrast, this paper allows the user to describe a class composing field objects, with confidentiality, integrity and availability type policies, and without distribution details. It then automatically partitions the class and infers adequate Byzantine quorum systems for methods and field objects to enforce the three polices.

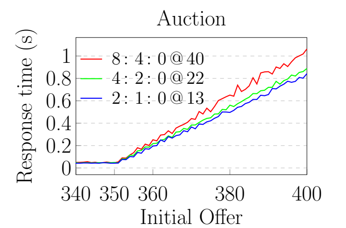

Fig. 11: Response time for increased load

State-machine replication is a well-known technique often used to tolerate crash failures [10], [31], [43], [44]. Byzantine failures were coined in the Byzantine Generals agreement problem [32] together with a few early protocols for Byzantine replication. Later, the more practical PBFT protocol [12], and an abundant number of optimized variants such as Q/U [2], HQ [19], Zyzzyva [30], Stewart [3], ABSTRACT [4], MinBFT [53], CheapBFT [27], ZZ [56], UpRight [17], BFT2F [33], Aardvark [18] and HoneyBadgerBFT [39] appeared. Further, researchers verified the replication protocols [13], [22], [24]–[26], [28], [45], [48], [49], [57]. However, these projects only consider a monolithic replicated system. In contrast, this project supports classes whose methods are implemented based on multiple objects. It partitions each method and separately replicates each partitioned method and field object, and yet guarantees end-to-end non-interference and resiliency.

XI. Conclusion

This paper presented a theoretical framework and a system for trustworthy distributed systems. It includes a lattice model of resiliency, a security-typed object-based language to capture end-to-end type polices for the three aspects of trustworthiness, a partitioning transformation, operational semantics, an information flow type inference system, and quorum constraint solving to automatically construct partitioned and replicated systems that guarantee non-interference and resiliency properties especially for availability in the face of Byzantine failures.

REFERENCES

[1] Netty project. https://netty.io/, 2021.

[2] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 59–74, New York, NY, USA, 2005. ACM.

[3] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. volume 7 of *IEEE Transactions on Dependable and Secure Computing*, pages 80–93. IEEE, 2008.

[4] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quema, and Marko Vukolic. The next 700 bft protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, January 2015.

[5] Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *2009 30th IEEE Symposium on Security and Privacy*, S&P '09, pages 141–153. IEEE, 2009.

[6] Michael Backes, Matteo Maffei, and Kim Pecina. Automated synthesis of privacy-preserving distributed applications. In *Proc. of ISOC NDSS*, NDSS '12, 2012.

[7] Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[8] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State machine replication for the masses with bft-smart. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, page 355–362, USA, 2014. IEEE Computer Society.

[9] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, volume 57, 2004.

[10] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[11] Darion Cassel, Yan Huang, and Limin Jia. Flownotation: An annotation system for statically enforcing information flow policies in c. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2207–2209, 2018.

[12] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[13] Saksham Chand, Yanhong A Liu, and Scott D Stoller. Formal verification of multi-paxos for distributed consensus. In *International Symposium on Formal Methods*, pages 119–136. Springer, 2016.

[14] Alvin Cheung, Owen Arden, Samuel Madden, and Andrew C Myers. Automatic partitioning of database applications. *Proceedings of the VLDB Endowment*, 5(11), 2012.

[15] Alvin Cheung, Owen Arden, Samuel Madden, Armando Solar-Lezama, and Andrew C Myers. Statusquo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.

[16] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. *ACM SIGOPS Operating Systems Review*, 41(6):31–44, 2007.

[17] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 277–290, New York, NY, USA, 2009. ACM.

[18] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[19] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[20] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[21] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[22] Cezara Drăgoi, Thomas A Henzinger, and Damien Zufferey. Psync: a partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices*, 51(1):400–415, 2016.

[23] Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 432–441, 2009.

[24] Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. Paxos consensus, deconstructed and abstracted. In *European Symposium on Programming*, pages 912–939. Springer, Cham, 2018.

[25] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

[26] Mauro Jaskelioff and Stephan Merz. Proving the correctness of disk paxos. *Archive of Formal Proofs*, 2005, 2005.

[27] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308, New York, NY, USA, 2012. ACM.

[28] Pertti Kellomäki. An annotated specification of the consensus protocol of paxos using superposition in pvs. Technical report, Technical report 36, Tampere University of Technology, Institute of Software ..., 2004.

[29] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. Crowdflow: Efficient information flow security. In *Information Security*, pages 321–337. Springer, 2015.

[30] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

[31] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

[32] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[33] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 10–10, Berkeley, CA, USA, 2007. USENIX Association.

[34] Mark C Little and Daniel L McCue. The replica management system: a scheme for flexible and dynamic replication. In *Proceedings of 2nd International Workshop on Configurable Distributed Systems*, pages 46–57. IEEE, 1994.

[35] Jed Liu, Owen Arden, Michael D George, and Andrew C Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security*, 25(4-5):367–426, 2017.

[36] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2359–2371, 2017.

[37] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. STOC '14, pages 1219–1234, 2012.

[38] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[39] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 31–42, New York, NY, USA, 2016. Association for Computing Machinery.

[40] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

[41] Sri Hari Krishna Narayanan, Mahmut Kandemir, and R Brooks. Performance aware secure code partitioning. In *2007 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6. IEEE, 2007.

[42] Matthias Neubauer and Peter Thiemann. From sequential programs to multi-tier applications by program transformation. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–232, 2005.

[43] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[44] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[45] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made epr: decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–31, 2017.

[46] James Parker, Niki Vazou, and Michael Hicks. Lweb: Information flow security for multi-tier web applications. volume 3 of *POPL '19*, pages 1–30. ACM New York, NY, USA, 2019.

[47] Francois Pottier and Vincent Simonet. Information flow inference for ml. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, 2002.

[48] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L Constable. Formal specification, verification, and implementation of fault-tolerant systems using eventml. *Electronic Communications of the EASST*, 72, 2015.

[49] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. Velisarios: Byzantine fault-tolerant protocols powered by coq. In *European Symposium on Programming*, pages 619–650. Springer, 2018.

[50] Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *International Static Analysis Symposium*, pages 376–394. Springer, 2002.

[51] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[52] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 355–364, 1998.

[53] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, Jan 2013.

[54] K Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 173–186, 2009.

[55] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2-3):167–187, 1996.

[56] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. Zz and the art of practical bft execution. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 123–138, New York, NY, USA, 2011. ACM.

[57] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

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

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

[60] Lantian Zheng and Andrew C Myers. End-to-end availability policies and noninterference. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 272–286. IEEE, 2005.

[61] Lantian Zheng and Andrew C Myers. A language-based approach to secure quorum replication. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 27–39, 2014.

XII. ACKNOWLEDGMENTS

We appreciate the S&P 2022 reviewers for the constructive and insightful comments. This project was supported by the NSF grants #1718997 and #1942711.