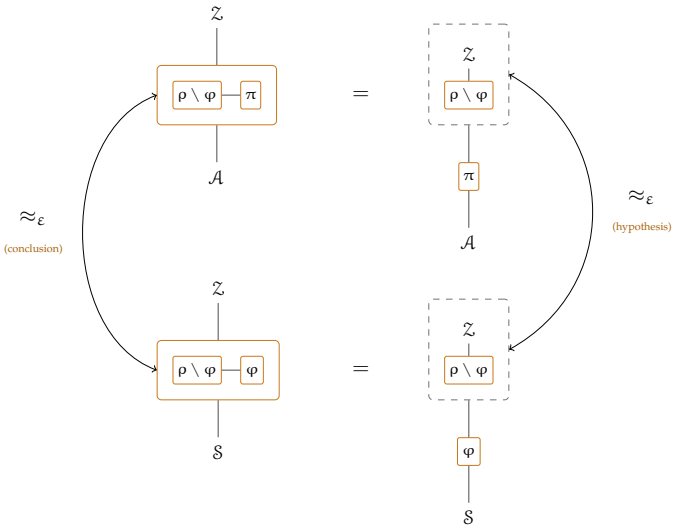
UC for Gamers



AI: Claude Opus 5 (1M context)
Prompts: Pooya Farshim
Reviewers: Nobody

Date: August 15, 2026

Universal composability is usually written in a mixture of prose and code, which makes UC research hard to falsify, and harder still to test or verify formally. Following the code-based style of property-based security, we set out a rigorous language for programming UC functionalities: identities and access control are data, wrappers enforce them, corruption is a register that every interface reads, and an execution is a handler passing a single token. Emulation is then defined over explicit machine classes with concrete bounds — no security parameter, no asymptotics — and the composition toolkit follows: substitution, parallel composition and a completeness theorem for the dummy adversary — all three corollaries of one absorption lemma, which moves a protocol or an adversary into the environment and is an exact regrouping of a single execution rather than a comparison of two — with transitivity coming free from the metric itself. Game-based properties turn out to be functionalities too, and they transfer along emulation. A signature functionality, a global clock and a bounded-delay network show the style at work. We hope this helps apply formal methods to UC, and ultimately give a formal UC proof.

Contents

Introduction

Universal composability is written, almost everywhere it is written, in a mixture of prose and code. A functionality's interfaces are code; who may call them is a sentence. The corruption model is a paragraph; what a corrupt party's interface actually returns is left to the reader. Each gap is small, and a human referee closes it without noticing. A machine cannot, which is why a UC proof is hard to falsify and harder still to check formally: much of what would have to be checked was never written down.

This paper writes those parts down as code. Identities and access control become data that a functionality carries in named fields. Wrappers enforce them: a guard decides which calls reach a core, a silencer decides what a core may claim, and a mediator hands a corrupt party's call to the adversary. Corruption is a register that every interface reads, twice, at points the code names. An execution is a handler that passes a single token. Nothing here is a new model — it is Canetti's [7], with several of its conventions made explicit — but stating them in code rather than prose is what lets the rest of the paper be proved rather than argued.

One lemma. Emulation is then defined over explicit machine classes, with concrete bounds: no security parameter, no asymptotics, a bound that is a number. What follows is a composition toolkit, and the shape of it is the paper's main structural claim. Substitution, parallel composition and the completeness of the dummy adversary are all corollaries of a single *absorption* lemma, which moves a protocol or an adversary into the environment. Absorption is not a comparison of two executions but an exact regrouping of one: the same run, bracketed differently, so the advantage on either side is the same number rather than two numbers with a shared bound. Transitivity is the exception and is of a different kind, resting on the triangle inequality over the metric rather than on

any regrouping; it is stated with the metric for that reason, before the machinery the other three need.

Games are functionalities. A property in the game-based style of Bellare and Rogaway [5] — unforgeability, liveness, authentication — turns out to be expressible as a functionality too, with the game's challenger as an ordinary machine and its verdict as an ordinary interface. Properties written this way transfer along emulation: if a protocol emulates an ideal object, the object's properties descend to it, under hypotheses the paper states and checks at each use rather than assuming.

What to read first. The apparatus is larger than the theorem count suggests, and not all of it is load-bearing. Five definitions carry most of the paper: a *functionality* and its five fields; an *execution* and what it means to occupy an identity; a *bundle*, which runs several machines as one occupant and is what absorption is built from; *absorption* itself; and the *regrouping correspondence* that pairs two executions machine by machine. A reader who has those has the spine, and the composition results follow from them in order. The concrete functionalities — a signature scheme, a global clock and a bounded-delay network after Katz, Maurer, Tackmann and Zikas [11] and Badertscher, Maurer, Tschudi and Zikas [3], and an authenticated channel realized over them — come last and exist to show the style at work rather than to be needed by anything before them.

What this is for. A UC researcher should find the conventions here the ones they already use, written down. The second audience is the one the extra precision is for. Anyone building tools over UC — a type-checker for functionality code, a proof-assistant encoding, a test harness — needs a specification that settles the questions an informal presentation can leave to its reader, and settling them is most of what this paper does: which caller a guard admits, what a silenced core may claim, when a corruption register is read. Whether that is enough to carry a machine-checked UC proof is not established here, and we do not claim it. It is why the framework is written this way.

CHAPTER 1

Functionalities and Systems

We work in the universal composability framework of Canetti [7], several of whose conventions we make explicit below.

A process id names a running protocol instance, and it has three parts:

$$pid \;=\; (\mathsf{F},\, \mathsf{s},\, \mathsf{i}),$$

a *functionality name* F, a *session id* s, and an *instance number* i. The name says what code runs, the session which run of a protocol the instance belongs to, and the number distinguishes several instances of one name within a session. We write $\mathtt{id}.\mathsf{F}$, $\mathtt{id}.\mathsf{s}$ and $\mathtt{id}.\mathsf{i}$ for the three parts of $\mathtt{id}.pid$, and likewise $\mathcal{F}.\mathsf{F}$, $\mathcal{F}.\mathsf{s}$, $\mathcal{F}.\mathsf{i}$ for those of $\mathcal{F}.pid$; a bare $\mathtt{id}.pid = \mathcal{F}.pid$ means all three agree.

Names are elements of $\{0,1\}^* \cup \{\mathsf{A},\mathsf{Z},\mathsf{C}\}$ and party ids of $\{0,1\}^* \cup \{\mathsf{A},\mathsf{Z}\}$, where A, Z and C are reserved for the adversary, the environment and the corruption register. Being reserved, none of the three is an element of $\{0,1\}^*$, so a field declared to contain $\{0,1\}^*$ contains none of them; and C names no party, the register being a machine that parties call rather than one any party runs. Each of the three runs as a single instance in a single session, fixed to be session 0 and instance 0, so their process ids are $(\mathsf{A},0,0)$, $(\mathsf{Z},0,0)$ and $(\mathsf{C},0,0)$. We abbreviate these to the bare names A, Z and C throughout, the other two components carrying no information.

Tests are almost always on the name alone. Where we write $\mathtt{id}.\mathsf{F} = \mathsf{A}$ we mean the name component, never the whole triple, and the same for Z, for C, and for any functionality named in a membership test.

Two of the fields below collect names of this kind, and they collect different things. A party set holds *party ids*, elements of $\{0,1\}^* \cup \{\mathsf{A},\mathsf{Z}\}$; a caller set holds whole *process ids*, because who may call is a question

about instances and not only about code. A caller set is therefore written with the abbreviations

$$\mathbf{A} := \{pid : pid.\mathsf{F} = \mathsf{A}\}, \qquad \mathbf{Z} := \{pid : pid.\mathsf{F} = \mathsf{Z}\},$$
$$\mathbf{Std} := \{pid : pid.\mathsf{F} \in \{0,1\}^*\}$$

for the process ids of the adversary, of the environment, and of everything standard. Where an older notation would have admitted "A and Z" it now admits $\mathbf{A} \cup \mathbf{Z}$, of which only $(\mathsf{A}, 0, 0)$ and $(\mathsf{Z}, 0, 0)$ ever occur, the execution supplying one of each; where it admitted "$\{0,1\}^*$" it now admits **Std**, and there every session and instance genuinely does occur. Party sets keep the bare $\{\mathsf{A}, \mathsf{Z}\}$, those being party ids and not process ids, so a declaration that once read $\mathbf{P} := \mathbf{N}$ has to be split.

A functionality is named the same way, by its name component written as a subscript: $\mathcal{F}_{\mathsf{Sig}}$ is a functionality with $\mathcal{F}_{\mathsf{Sig}}.\mathsf{F} = \textsc{Sig}$, its session and instance left implicit, so the notation stands for a process id $(\textsc{Sig}, \mathtt{s}, \mathtt{i})$ with $\mathtt{s}$ and $\mathtt{i}$ unspecified. Where a particular instance is meant they are supplied separately; where only the code matters, they are not written at all.

Little is lost by suppressing them, because a functionality's own code hardly ever reads them. Local or global, a core is written for the party it serves and the state it keeps; which session it belongs to, and which instance of its name it is, are the execution's business rather than its own. In this paper $\mathtt{s}$ occurs in three places only — line 3 of GUARD, the wrappers that invoke it, and the static check of Definition 1.3 — and $\mathtt{i}$ is never read at all. The instance number earns its place another way: it is what lets two instances of one name in one session carry the distinct process ids that Definition 1.2 asks for. Chapter 7 makes the observation into a statement: renaming sessions and instances is an isomorphism of executions, and an emulation proved at one process id holds at every other, under a hygiene condition on who may be called across sessions that every functionality here meets.

Definition 1.1 (Identities and functionalities)**.** An *identity* is a pair $\mathtt{id} = (pid, \mathsf{P})$ of a process id and a party id, and an *identity set* is any set $\mathcal{I}$ of these — a process id being a triple, no longer a set of pairs of names. A *functionality* $\mathcal{F}$ carries the five fields

$\mathcal{F}.pid$	a process id	
$\mathcal{F}.\mathbf{P}$	set of *served* parties	the interfaces that exist
$\mathcal{F}.\mathbf{N}$	caller process ids it admits	who may call those interfaces
$\mathcal{F}.\mathbf{U}$	functionalities it *uses*	what must be there for it to run
$\mathcal{F}.par$	any further parameters	what the code reads besides the above

and *occupies* the identity set

$$\text{IDs}(\mathcal{F}) \;:=\; \big\{\, (\mathcal{F}.pid,\, \mathsf{P}) \;:\; \mathsf{P} \in \mathcal{F}.\mathbf{P} \,\big\}.$$

Identity sets say two things: which identities a collection of functionalities occupies, and which a machine may claim. The five fields recur in the parameter line of every box below, written without the owner's name since the box heading supplies it. Only the first two fix $\text{IDs}(\mathcal{F})$: the process id is shared by all its identities and the served parties supply the second component.

The fourth field records dependence. An entry of $\mathcal{F}.\mathbf{U}$ is a pair $(\mathcal{F}', \mathsf{R})$: $\mathcal{F}'$ names a functionality that must be present for $\mathcal{F}$ to run, and R is a predicate saying what $\mathcal{F}$ requires of whatever is found there. Naming a subroutine is rarely enough on its own — a subroutine that served none of its caller's parties, or admitted none of its callers, would be useless — so we write

$$\text{SERVES}(\mathcal{F}, \mathcal{G}) \;:\Longleftrightarrow\; \mathcal{G}.\mathbf{P} \supseteq \mathcal{F}.\mathbf{P} \;\wedge\; \mathcal{F}.pid \in \mathcal{G}.\mathbf{N}$$

for the requirement that comes up most: $\mathcal{G}$ serves every party $\mathcal{F}$ serves, and admits $\mathcal{F}$ among its callers. Without the second conjunct GUARD would refuse every call from $\mathcal{F}$ to $\mathcal{G}$.

What **U** records is what the *code* of $\mathcal{F}$ names. The calls its wrapper makes on its behalf — to $\mathcal{C}$ on lines 2 and 6 of Section 3.5, and to $\mathcal{A}$ through MEDIATE — are common to every standard functionality and are left out, which costs nothing: both targets serve $\{0,1\}^* \cup \{\mathsf{A}, \mathsf{Z}\}$ and admit **Std**, so $\text{SERVES}(\mathcal{F}, \mathcal{C})$ and $\text{SERVES}(\mathcal{F}, \mathcal{A})$ hold of every standard $\mathcal{F}$ whatever its own fields, and an entry for either could never fail.

Definition 1.2 (Systems)**.** A functionality is *standard* if $\mathcal{F}.\mathsf{F} \notin \{\mathsf{A}, \mathsf{Z}, \mathsf{C}\}$, and each of its cores is wrapped as in Section 3.5 — save LEAK, which is wrapped as in Section 3.2, and INITIALIZE, which is a procedure and is not wrapped at all. A *system* (or *world*) π is a finite set of standard functionalities carrying distinct process ids: if

$\mathcal{F}, \mathcal{F}' \in \pi$ and $\mathcal{F} \neq \mathcal{F}'$ then $\mathcal{F}.pid \neq \mathcal{F}'.pid$. It occupies whatever its members do,

$$\text{IDs}(\pi) \;:=\; \bigcup_{\mathcal{F} \in \pi} \text{IDs}(\mathcal{F}) \;=\; \big\{\, (\mathcal{F}.pid,\, \mathsf{P}) \;:\; \mathcal{F} \in \pi,\; \mathsf{P} \in \mathcal{F}.\mathbf{P} \,\big\}.$$

A system is what a protocol designer writes down — we use *protocol* and *system* interchangeably, *ideal* and *real* naming styles of writing one (below), not disjoint classes. Three machines are no part of it: the environment $\mathcal{Z}$, the adversary $\mathcal{A}$, and the corruption register $\mathcal{C}$ of Section 3.1. None is standard, so by Definition 1.2 no system contains any of them and $\text{IDs}(\mathcal{Z})$, $\text{IDs}(\mathcal{A})$ and $\text{IDs}(\mathcal{C})$ lie outside every system; the execution of Section 3.6 supplies all three instead. For $\mathcal{Z}$ and $\mathcal{A}$ that is what lets each claim its own identity while silenced on the system under test. For $\mathcal{C}$ it keeps the register clear of the set operations of Section 4.1 and Chapter 6: no blocker is built for it, no replacement moves it, and every comparison runs over one and the same register. Reserving the three names also keeps them out of every $\mathcal{F}.\mathbf{N}$ declared as **Std**, so no functionality admits the register among its callers — which is right, the register being a callee only.

Distinctness of process ids does real work. A system is determined by what it places at each id it uses, so membership is settled by the id alone. And every identity records that id in its first component, so distinct members have $\text{IDs}(\mathcal{F}) \cap \text{IDs}(\mathcal{F}') = \varnothing$: the union above is disjoint, and no identity of a system is occupied twice.

Definition 1.3 (Well-formed systems)**.** Write $\pi^+ := \pi \cup \{\mathcal{Z}, \mathcal{A}, \mathcal{C}\}$ for a system together with the three machines every execution supplies. The system π is *well-formed*, written $\text{WF}(\pi)$, if

$$\forall \mathcal{F} \in \pi^+ \;\; \forall (\mathcal{F}', \mathsf{R}) \in \mathcal{F}.\mathbf{U} \;\; \exists \mathcal{G} \in \pi^+ \;:$$
$$\mathcal{G}.\mathsf{F} = \mathcal{F}'.\mathsf{F} \;\wedge\; \big(\mathcal{G}.\mathsf{s} = \mathcal{F}.\mathsf{s} \;\vee\; \text{GLOBAL}(\mathcal{G})\big)$$
$$\wedge\; \mathsf{R}(\mathcal{F}, \mathcal{G}).$$

Three things about this. The conditions on $\mathcal{G}$ are exactly those GUARD imposes when the call is placed — name admitted, session matching unless the callee is shared, fields fitting — so $\text{WF}(\pi)$ says precisely that the calls the members of π are built to make can succeed.

Matching is by *name*, not by the functionality itself, which is what lets a dependence survive subsystem replacement: an entry naming φ_q is met by whatever sits at that name, ideal or real, so long as it meets the rest. And the requirement half ties the fields of caller and callee together; WF would say almost nothing without it.

Both quantifiers range over π^+, and each enlargement earns its place: on the right it makes an entry naming one of the three supplied machines satisfiable at all, none being a member of any system, and on the left it checks their own entries — immediately here, since SERVES$(\mathcal{Z}, \mathcal{A})$ and SERVES$(\mathcal{A}, \mathcal{Z})$ hold and $\mathcal{C}$.**U** is empty. What WF does not do on its own is survive the constructions that follow: adding members could introduce obligations, and replacing a subsystem re-evaluates every requirement at the new occupant. Proposition 6.4 (Chapter 6) settles the first and Theorem 4.4 the second.

Conditions, collected. Well-formedness is one of several static conditions this paper puts on systems, cores and machines, and they are defined where each is first needed — which leaves a reader no way to see which result reads which. The table collects them once. None of them is read during an execution: EXEC reads fields, not conditions, as Proposition 7.7 records for **U** in particular.

condition	constrains	read by
WF(π), Def. 1.3	a system: every **U** entry has a witness — the calls its members are built to *make* can succeed	Thm. 4.4, Prop. 6.4, Def. 5.1
well-formed replacement, Def. 4.2	an incoming system against what stays: no process id in common	Prop. 4.3, Thm. 4.4, Prop. 4.16, Thm. 4.20, Thm. 4.24, Prop. 5.10
compatible, Def. 6.1	two systems: a shared process id serves and admits alike	Thm. 6.3, second claim
session-uniform, Def. 7.2	a system or environment: who can *reach* a member does not depend on its session	Lem. 7.9, Prop. 7.10, Cor. 7.12, Cor. 7.13
tight at I, Rem. 6.5	a system: members outside I pin callers by process id	Def. 5.1
places no responsive call, Def. 9.2	the cores of a system	Thm. 4.29, Lem. 4.18 (ceding half), Prop. 5.9, Cor. 5.11
refusal-oblivious, Def. 4.28	a machine in the adversary slot	the same four
identity-atomic, Def. 7.4	all code; a standing convention rather than a per-system check	Lem. 7.9, Prop. 7.15
leakage-well-formed, Section 3.2	a realization	nothing: recorded, not used

Two things the table makes visible. The first two rows share a name and are different conditions: WF is about a system on its own, well-formed *replacement* about where an incoming system lands. And WF and session-uniformity are duals — the calls a member is built to make, against who may reach it — which invites folding them into one condition. They are kept apart because no result needs both, and because one definition could not carry both anyway: the game of Definition 5.1 is required to be well-formed and is deliberately *not* session-uniform, as Remark 5.6 explains.

We say $\mathcal{F}$ is *local* if $\mathcal{F}.\mathbf{N} = \{pid : pid.\mathsf{F} = \mathsf{F}^{\uparrow}\}$ for a designated parent (caller) name $\mathsf{F}^{\uparrow}$. (We leave it to the functionality to say whether **A** and **Z** may call it.) Intermediate notions are possible: a setup reachable from two named protocols admits $\{pid : pid.\mathsf{F} \in \{\mathsf{F}_1, \mathsf{F}_2\}\}$; and an **N** may pin whole process ids rather than names, listing its callers' occupied instances outright — Remark 6.5 says why the private members of a realization should. At the other end we say $\mathcal{F}$ is *global*, in the sense of generalized UC's shared setup [8], if

$$\textsc{global}(\mathcal{F}) \;:\Longleftrightarrow\; \mathbf{Std} \subseteq \mathcal{F}.\mathbf{N} \;\vee\; \mathcal{F}.\mathsf{F} \in \{\mathsf{A}, \mathsf{Z}, \mathsf{C}\},$$

one notion and not two: the first disjunct is the ordinary case, any standard functionality may call it, and the second covers the three machines the execution supplies — and whatever stands in their slots — global for a different reason and treated so wherever GLOBAL is tested.

The distinction is what the session component is for. A local functionality is part of one run of one protocol, so a call may reach it only from its own session: line 3 of GUARD asks $\mathtt{id}'.s = \mathtt{id}.s$. The instance number is not checked, so one session may hold as many instances of a name as it likes and they may call one another. A global functionality is shared across runs — one session of it, reachable from every session — so that check is waived for it, and the sessions of $\mathcal{F}$ may all call one shared instance. We name a global ideal functionality with a leading G and a global real one with a leading γ, as in $\mathcal{F}_{\mathsf{GSIG}}$; the calligraphic $\mathcal{G}$ stays what it has been, a second functionality alongside $\mathcal{F}$. The three machines the execution supplies are global: $\mathcal{A}$ and $\mathcal{C}$ already by the first clause, and $\mathcal{Z}$ by the second, which is there because $\mathcal{Z}.\mathbf{N} = \mathbf{A} \cup \mathbf{Z}$ is deliberately narrow and yet the environment must be reachable from every session.

Real vs. ideal specifications. Ideal functionalities and real protocols differ in several respects. First, ideal functionalities know the corruption status of parties in their party sets. Second, they can pass the execution to the simulator and may offer simulator-specific interfaces (see, e.g., $\mathcal{F}_{\mathsf{Diffuse}}$). Finally, they may share state across parties — and sharing is the point: the guard of Chapter 2 ties every call to one party, so parties communicate only through functionalities that share state across them, and a real protocol talks to its peers through ideal subroutines and in no other way. Here and below, $\mathcal{F}_{\mathsf{Diffuse}}$ and $\mathcal{F}_{\mathsf{IO}}$ name functionalities we do not give: a diffusion channel and an input–output relay. Nothing in this paper depends on their code. The randomness

source $\mathcal{F}_{\mathsf{Rand}}$ and the store $\mathcal{F}_{\mathsf{Store}}$ are given, in Chapters 13 and 14; they are the smallest examples in the paper and are where a reader new to the style should start.

The core interfaces of well-specified *real* protocols have none of these features, beyond those implicit in idealized building blocks and access-control wrappers; and real realizations do not specify how leakage is computed upon corruption, it being the contents of work tapes.[1]

Initialization. INITIALIZE is a procedure and not an interface: it runs once, when the functionality is first activated, and sets up the variables the rest of the code uses. It places no calls — the execution of Section 3.6 initializes every machine before its dispatcher is installed, so there would be nothing to serve one — no caller can reach it, it takes no claimed caller-id, and a functionality declaring none gets the empty one. Real protocols carry no global initializer of this kind, only ones specific to a party within a process, written (pid, P).INITIALIZE() and abusing notation: they are private, not interfaces anyone may address.

[1] A more disciplined approach would model storage as an ideal functionality offering secure erasures; we avoid the extra layer to ease notation.

CHAPTER 2

Guards, Silencers, and Mediators

This chapter fixes three pieces of machinery. The predicate GUARD constrains the calls that reach a machine; the wrapper SILENCE constrains the calls it makes; and the wrapper MEDIATE hands to the adversary a call that a corrupt party cannot answer for itself. The full interface of Section 3.5 is their composition.

Write OP for a generic core interface of $\mathcal{F}$. We specify only the cores; the wrapper below turns each into a full interface, and a caller reaches FULLOP, never OP. The access control checks are collected in the predicate GUARD, which is not a functionality but an algorithm: it takes the target identity $\mathtt{id}$, the claimed caller identity $\mathtt{id}'$ and the functionality $\mathcal{F}$, and returns a boolean. Two conventions of presentation. Boxes come in two weights: a filled one marks a functionality, which occupies identities, and an unfilled one a predicate, wrapper or procedure, which occupies none. And a call is written $\mathtt{id}.\text{OP}(\mathtt{in})$ **as** $\mathtt{id}'$ where $\mathtt{id}'$ is the identity claimed in placing it, and received as $\mathtt{id}.\text{OP}(\mathtt{in})$ **from** $\mathtt{id}'$ where $\mathtt{id}'$ is the identity it was claimed under. Enforcement sits in the wrapper, where line 1 **require**s it, so a call it rejects never reaches a core. Line 1 checks that the interface exists; line 2, that the caller's name is admitted and that it acts for the right party; line 3, that caller and callee lie in one session, unless the callee is shared across sessions.

Predicate GUARD

$\text{GUARD}_{\mathcal{F}}(\mathtt{id}, \mathtt{id}')$:

1: **return** $\mathtt{id}.\mathit{pid} = \mathcal{F}.\mathit{pid} \wedge \mathtt{id}.\mathrm{P} \in \mathcal{F}.\mathbf{P}$ *// interfaces that exist*
2: $\quad \wedge\ \mathtt{id}'.\mathit{pid} \in \mathcal{F}.\mathbf{N} \wedge \mathtt{id}'.\mathrm{P} = \mathtt{id}.\mathrm{P}$ *// callers admitted*
3: $\quad \wedge\ (\mathtt{id}'.\mathrm{s} = \mathtt{id}.\mathrm{s} \vee \text{GLOBAL}(\mathcal{F}))$ *// one session, unless shared*

GUARD constrains the calls that reach a machine; the wrapper

SILENCE below constrains the calls it *makes*. It takes an identity set $\mathcal{I}$ and a programme, runs it, and intercepts every call it issues. A call claiming an identity in $\mathcal{I}$ is refused with REJ without ever being placed; any other is placed and its answer handed back. So $\mathcal{I}$ is exactly where the wrapped programme is silenced, and it may speak as anything outside.

The scope is the wrapped programme and nothing further. A call it places passes the token to another machine, and what *that* machine says is governed by its own wrapper, not by this one. Were it otherwise a functionality would silence its subroutines transitively, and none could speak as itself. Every use of SILENCE below is of this shape: a full interface wrapping the core it serves.

It is convenient to read this as a handler, in the sense of algebraic effects [12, 4]. A call the programme issues is an operation of the ambient effect, and $\mathtt{k}$ is the continuation waiting for its answer. The wrapper SILENCE decides what to resume with: REJ for a silenced claim, the answer of the call actually placed otherwise. The **return** clause passes the programme's own result through untouched, so silencing changes what the programme may say and nothing else. The analogy only fixes the shape of the wrapper.

Wrapper SILENCE
identity set $\mathcal{I}$, programme $\mathtt{id}.\mathrm{OP}(\mathtt{in})$ from $\mathtt{id}'$

SILENCE$(\mathcal{I},\ \mathtt{id}.\mathrm{OP}(\mathtt{in})$ from $\mathtt{id}')$:

handle $\mathtt{id}.\mathrm{OP}(\mathtt{in})$ from $\mathtt{id}'$ **with**
  return $\mathtt{out}$ $\mapsto\ \mathtt{out}$
  $\mathtt{id}''.\mathrm{FULLOP}(\mathtt{in}'')$ as $\mathtt{id}'''$; $\mathtt{k}$ $\mapsto\ \mathtt{k}(\mathrm{REJ})$ if $\mathtt{id}''' \in \mathcal{I}$
  $\mapsto\ \mathtt{k}(\mathtt{id}''.\mathrm{FULLOP}(\mathtt{in}'')$ as $\mathtt{id}''')$ otherwise

With SILENCE in hand we can say what happens to a call that a corrupt party cannot answer for itself. Every full interface below hands such a call to the adversary twice, on the way in and on the way out, and both times in the same shape: the wrapper MEDIATE. It takes a corruption set $\mathbf{C}$, the operation $\mathtt{id}.\mathrm{OP}$ being served, the value $\mathtt{x}$ to hand over, and the claimed caller $\mathtt{id}'$ of the operation it stands in. Where the answer comes from is fixed inside it: whatever the operation, a corrupt party's answers come from $(A, \mathtt{id}.\mathrm{P})$. One thing about it is not an ordinary argument: both **require** and the **return** of MEDIATE return from the operation in whose code they stand, not merely from the line

itself. When MEDIATE's test fails it does nothing and that operation carries on.

The hook is an ordinary call on $\mathcal{A}$'s full interface, claiming the identity id of the interface being served. It therefore meets $\text{GUARD}_{\mathcal{A}}$, which it passes because the claimed party id.P is the party of the target; and the answering adversary is confined by $\mathcal{A}$'s own silencer, which admits only $(\mathcal{A}, \text{id.P})$. So while answering for id.P the adversary may speak as itself at id.P and as nothing else, and MEDIATE needs no silencing of its own.

Corruption wrapper MEDIATE
corruption set **C**, operation id.OP, value x, claimed caller id′

MEDIATE(**C**, id.OP, x, id′):
1: **if** (id′.F $\neq \mathcal{A} \wedge$ id.P $\in$ **C**) **then**
2: **return** $(\mathcal{A}, \text{id.P})$.FULLA(id.OP, x) as id

CHAPTER 3

The Execution Model

3.1 Corruption

Corruption is not an interface that real protocols realize (what would a real protocol do when "instructed" to get corrupted?), but it must exist for emulation. We can assume real protocols always reject corruptions, but then they are never registered as corrupted.

We model corruptions as a global functionality $\mathcal{C}$. Several machines have their full interfaces given outright rather than obtained from cores — the environment, the adversary, and the leakage operation of Section 3.2 — but $\mathcal{C}$ is the only one that uses neither SILENCE nor MEDIATE, and the status report is why: that wrapper reads the corruption status on every call, which is what this interface implements, so wrapping it would be circular. Each interface below therefore carries by hand only the checks it needs, and FULLSTATUS in particular *always* returns the corrupted set without first asking whether the party is corrupt.

$\mathcal{C}$'s own leakage is constant, so the corruption test that Section 3.2 imposes on every other leakage operation would have nothing to protect; the guard is kept all the same, so the adversary reads the register at the party it is acting for and at no other.

Reading the register. $\mathbf{C}$ is $\mathcal{C}$'s own state, so no other machine holds a copy. Wherever $\mathbf{C}$ appears in the code of a functionality other than $\mathcal{C}$ itself, the understanding is that it was fetched first, by

$$\mathbf{C} \leftarrow (\mathtt{C}, \mathtt{id}.\mathtt{P}).\textsc{FullStatus}() \text{ as } \mathtt{id},$$

and we suppress that line when it would only repeat what the surrounding code makes plain. Inside $\mathcal{C}$ there is nothing to fetch: $\mathbf{C}$ is read and written directly, and CORRUPT updates it in place.

Corruption register $\mathcal{C}$
pid $:= C$, $\mathbf{P} := \{0,1\}^* \cup \{A, Z\}$, $\mathbf{N} := \mathbf{Std} \cup \mathbf{A} \cup \mathbf{Z}$, $\mathbf{U} := \varnothing$, *par* $:= \perp$

INITIALIZE():
1: $\mathbf{C} \leftarrow \varnothing$

id.FULLCORRUPT() from id′
2: **require** GUARD$_{\mathcal{C}}$(id, id′)
3: **require** id′.F $\in \{A, Z\}$ *// the outside only, never a subroutine*
4: **require** id.P $\in \{0,1\}^*$ *// reserved names stay honest*
5: $\mathbf{C} \leftarrow \mathbf{C} \cup \{\text{id.P}\}$
6: **return** OK

id.FULLSTATUS() from id′
7: **require** GUARD$_{\mathcal{C}}$(id, id′)
8: **return** **C**

id.FULLLEAK() from id′
9: **require** GUARD$_{\mathcal{C}}$(id, id′)
10: **require** id′.F $= A$
11: **return** $\perp$ *// constant: the register keeps no secrets*

Remark 3.1 (Why FULLSTATUS is not mediated)**.** Wrapped like the other operations, FULLSTATUS would be useless exactly when it matters. Suppose $\mathcal{Z}$ reaches out as id′ to ask the corruption status of id′.P. If id′.P is corrupt, the gate lets the call through and MEDIATE hands it to the adversary — so the answer to "is id′.P corrupt?" is whatever the adversary says, and it may say no. Since every caller reads the register to decide how to treat a corrupt party, an adversary-controlled answer would switch the mediation machinery off at will.
FULLSTATUS therefore mediates and silences nothing: past the guard it returns **C**, the register's own state. It is the one operation whose answer must come from the register, and being read-only it gives the adversary nothing it could not already infer. The core STATUS of earlier drafts goes with it, there being no wrapper left to keep out of.

Corruption is the outside's to do. Line 3 admits id′.F $\in \{A, Z\}$, so the adversary and the environment may each put a party into **C** and no standard functionality may put anything there at all. The guard above it binds the corrupting interface to the party it corrupts, asking id′.P $=$ id.P: to corrupt P the adversary must run at (A, P), and the

environment must claim (Z, P), which line 3 lets its root do for any party. Both may read as well: $\mathbf{A} \cup \mathbf{Z} \subseteq \mathcal{C}.\mathbf{N}$, and FULLSTATUS returns all of $\mathbf{C}$, so either may ask at any point and always learns the full set. And only genuine parties can be corrupted: FULLCORRUPT asks $\mathtt{id}.\mathsf{P} \in \{0,1\}^*$, so the reserved names stay honest by construction.

Commensurateness comes out right, and it is the *reading* that secures it rather than any division of who may write. The machine at $(\mathcal{A}, \mathsf{P})$ is $\mathcal{A}$ in one execution of an emulation experiment and $\mathcal{S}$ in the other, and nothing forces the two to corrupt the same parties. But $\mathcal{Z}$ is the same machine on both sides and reads $\mathbf{C}$ on both, so an occupant of the slot corrupting a different set is caught by an environment that simply asks. Commensurateness is thus a consequence, not an assumption: no admissible environment tells the two corruption sets apart with advantage beyond what Definition 4.11 allows the simulator. That the environment may corrupt for itself only sharpens this, the corruptions it makes being its own on both sides; what is left to catch is the occupant's own initiative.

Corruption is indexed by party name alone, so P cannot be corrupt at one process id and honest at another.[1]

3.2 Leakage

Functionality $\mathcal{F}$, leakage operation
pid, **P**, **N**, **U**, *par*

$\mathtt{id}$.FULLLEAK() from $\mathtt{id}'$
1: **require** $\mathtt{id}.pid = \mathcal{F}.pid \wedge \mathtt{id}.\mathsf{P} \in \mathcal{F}.\mathbf{P}$ // *the interface exists*
2: **require** $\mathtt{id}'.\mathsf{F} = \mathcal{A} \wedge \mathtt{id}'.\mathsf{P} = \mathtt{id}.\mathsf{P}$ // *$\mathcal{A}$, for this party*
3: $\mathbf{C} \leftarrow (\mathsf{C}, \mathtt{id}.\mathsf{P})$.FULLSTATUS() as $\mathtt{id}$
4: **require** $\mathtt{id}.\mathsf{P} \in \mathbf{C}$ // *only a corrupt party leaks*
5: **return** SILENCE($\{\mathtt{id}'' : \mathtt{id}'' \neq \mathtt{id}\}$, $\mathtt{id}$.LEAK() from $\mathtt{id}'$)

The three requirements are those the general wrapper of Section 3.5 imposes, restated because leakage is not obtained from a core in the usual way. Line 1 is the existence half of GUARD; the caller half is deliberately weaker, since $\mathcal{A}$ must be able to leak from a functionality that does not admit it as a caller, so line 2 asks only that the adversary

[1] One could instead corrupt an *identity* — $\mathbf{C}$ collecting identities, every test $\mathtt{id}.\mathsf{P} \in \mathbf{C}$ becoming $\mathtt{id} \in \mathbf{C}$ — at the cost of "spreading" corruptions to subprotocols. The party-wide convention is cleaner, infecting a whole family at once.

act for the party it is reading. Line 4 is the corruption gate. Without it an honest party's state would be readable at will, which is what line 3 of the full interface refuses everywhere else. The silencer is line 5's, so a leakage core that places calls claims its own identity like any other core.

For an ideal functionality, LEAK is part of the specification; the messages received so far and the contents of internal lists are typical. For real protocols it can be set to leak nothing, which models no leakage at all; ideal functionalities that say which parts of the state leak are the fix.

A protocol is *leakage-well-formed* if every machine body is pure between invocations, all sampled randomness is drawn through $\mathcal{F}_{\mathsf{Rand}}$ and all state persisting across invocations held in $\mathcal{F}_{\mathsf{Store}}$ rather than on a machine tape — each exposing it through its own LEAK — and inputs and outputs leak only where the protocol routes them through an input–output functionality $\mathcal{F}_{\mathsf{IO}}$, which is not the default. Work tapes are then never leaked, purity between invocations leaving nothing on them. No result below assumes any of this; it is the condition a realization must meet for its LEAK to mean anything, and we record it rather than use it.

$\mathcal{Z}$ cannot call this interface, so it need not be realized for indistinguishability. It must still be simulable, since the adversary can call it.

3.3 The Environment

We define a special functionality $\mathcal{Z}$, with a single per-id interface $\mathtt{id}$.FULLZ that guards and silences the core $\mathtt{id}$.Z. The field $\mathcal{Z}.\mathbf{N} = \mathbf{A} \cup \mathbf{Z}$ deserves emphasis: it admits the adversary and the environment itself and nothing else, so no standard functionality may call $\mathcal{Z}$. Anything wider would let a protocol call the environment as one of its own subroutines — an "open protocol," whose subroutines are left undefined. Two distinct things rule that out. The narrowness of $\mathcal{Z}.\mathbf{N}$ refuses the call itself: a standard caller's process id lies in **Std**, which meets $\mathbf{A} \cup \mathbf{Z}$ nowhere, so line 2 of GUARD rejects it whatever the caller declares. Definition 1.3 refuses the other half of openness, a member naming a dependence no member supplies. It is the first that does the work here: naming $\mathcal{Z}$ would not trip WF, since $\mathcal{Z} \in \pi^+$ makes such an entry satisfiable, so a system could be well-formed and still be open were the

field any wider.

The **Z** in $\mathcal{Z}$.**N** is there for a different reason: it lets $\mathcal{Z}$ call itself. The Z interfaces of the environment at different parties are separate, and admitting **Z** is what lets them reach one another.

Which of them may reach which is settled by the silenced set, and there it does more than keep $\mathcal{Z}$ off the systems. Every other machine is tied to where it runs: a standard functionality may claim only its own identity, and $\mathcal{A}$ is silenced on $\{\mathtt{id}'' \neq \mathtt{id}\}$. Nothing would tie $\mathcal{Z}$ on its own, since SILENCE tests only the claimed $\mathtt{id}'$ and GUARD only the target, so (Z, P) and $(\mathsf{Z}, \mathsf{P}')$ would be interchangeable and the party component of a $\mathcal{Z}$ identity would carry no information. Line 3 supplies the tie, as loosely as applications allow: an interface at $\mathtt{id}$ may claim $\mathtt{id}'$ when $\mathtt{id}'.\mathsf{P} = \mathtt{id}.\mathsf{P}$, when $\mathtt{id}'.\mathsf{P} = \mathsf{Z}$, or when $\mathtt{id}.\mathsf{P} = \mathsf{Z}$. In words, *act for your own party, or for the root; and the root may act for anyone.*

Both escapes are needed. Without $\mathtt{id}.\mathsf{P} = \mathsf{Z}$ the root is stranded, and it is where EXEC begins and the only interface that can reach the others. Without $\mathtt{id}'.\mathsf{P} = \mathsf{Z}$ the leaves are: the adversary's only route back into the environment is $(\mathcal{A}, \mathsf{P}) \to (\mathsf{Z}, \mathsf{P})$, and MEDIATE wakes $\mathcal{A}$ at $(\mathcal{A}, \mathtt{id}.\mathsf{P})$ for the party it answers for, so an environment needing to compare what the adversary said at P against what it said at P' could never bring the two together — these interfaces are separate, and cannot pool it through shared state either — and the class of Definition 4.11 would genuinely shrink. With both escapes, connectivity is a two-way star rooted at (Z, Z), and we know of nothing an unrestricted environment could do that the star cannot. Line 2 restricts in the same spirit: it stops $\mathcal{Z}$ claiming $(\mathcal{A}, \mathsf{P})$, and so stops it passing the corruption gate of line 3 and driving an *honest* party's interface — a capability the adversary it would be impersonating does not have either, as Remark 4.8 records of a simulator silenced on less. What that line no longer has to do is keep $\mathcal{Z}$ away from FULLCORRUPT, which line 3 admits it to under its own name.

Together, $\mathbf{A} \cup \mathbf{Z}$ is what applications need and no more. Admitting **A** is necessary because the environment has no other way back into the execution: once $\mathcal{Z}$ passes the token on it runs again only when something calls it, and only the adversary may; that same channel is how $\mathcal{Z}$ hears whatever the adversary chooses to tell it. Admitting **Z** is necessary because the environment is one interface per party rather than one machine, and without it the root could reach none of them. Nothing beyond these two is: a functionality that had to call $\mathcal{Z}$ would

treat it as a subroutine it does not define, and $\mathcal{C}$ reports corruptions to no one — the adversary knows what it corrupts, having done it — which is why $\mathcal{Z}$ must *ask* for the corruption set. So the environment is reachable only from the two machines outside every system.

Environment $\mathcal{Z}$
pid $:= \mathsf{Z}$, $\mathbf{P} := \{0,1\}^* \cup \{\mathsf{A}, \mathsf{Z}\}$, $\mathbf{N} := \mathbf{A} \cup \mathbf{Z}$, $\mathbf{U} := \{(\mathcal{A}, \text{SERVES})\}$, *par* $:= \mathcal{I}$

$\mathtt{id}.\text{FULLZ}(\mathtt{in})$ from $\mathtt{id}'$
1: **require** $\text{GUARD}_{\mathcal{Z}}(\mathtt{id}, \mathtt{id}')$
2: $\overline{\mathcal{I}} \leftarrow \mathcal{I} \cup \{\mathtt{id}'' : \mathtt{id}''.\mathsf{F} = \mathsf{A}\}$ *// $\mathcal{A}$ out of reach*
3: $\quad \cup \{\mathtt{id}'' : \mathtt{id}.\mathsf{P} \neq \mathsf{Z} \wedge \mathtt{id}''.\mathsf{P} \notin \{\mathtt{id}.\mathsf{P}, \mathsf{Z}\}\}$ *// own party, or the root*
4: $\mathtt{out} \leftarrow \text{SILENCE}(\overline{\mathcal{I}}, \mathtt{id}.\mathsf{Z}(\mathtt{in}) \text{ from } \mathtt{id}')$
5: **return** $\mathtt{out}$

3.4 The Adversary

The adversary $\mathcal{A}$ has the same shape: a single per-id interface $\mathtt{id}.\text{FULLA}$ guarding and silencing the core $\mathtt{id}.\mathsf{A}$. What separates it from the environment is who may reach it. Where $\mathcal{Z}.\mathbf{N}$ is only $\mathbf{A} \cup \mathbf{Z}$, the field $\mathcal{A}.\mathbf{N} = \mathbf{Std} \cup \mathbf{A} \cup \mathbf{Z}$ holds every process id that ever places a call. Only $\mathcal{C}$'s is missing, and $\mathcal{C}$ places none. That width is a requirement rather than a convenience: the mediation hooks of Chapter 2, and the calls ideal functionalities place through the abbreviation $\mathcal{A}(\cdot)$, are both calls placed on the adversary by a machine acting for some party.

These interfaces are what a backdoor looks like in this notation. The adversary holds no token of its own and cannot begin a call, so every attack it mounts starts from another machine handing it control: a party's interface through MEDIATE, or the environment directly, which is how an environment instructs an adversary at all. What it may say in return is narrow: the silenced set is $\{\mathtt{id}'' : \mathtt{id}'' \neq \mathtt{id}\}$, so an interface of $\mathcal{A}$ may claim only the identity it runs at.

Inside an interface with identity $\mathtt{id}$ we write $\mathcal{A}(\mathtt{in})$ for the call $(\mathsf{A}, \mathtt{id}.\mathsf{P}).\text{FULLA}(\mathtt{in})$ as $\mathtt{id}$, and $\mathcal{Z}(\mathtt{in})$ likewise. The claim must be $\mathtt{id}$ and no other, since line 5 silences a core on everything but its own identity; the adversary therefore learns the caller as the claimed id, and needs no separate field for it. Its core is entered as $\mathtt{id}.\mathsf{A}(\mathtt{in})$ from $\mathtt{id}'$, the shape SILENCE declares and the shape every other core is entered in. The target's party component is $\mathtt{id}.\mathsf{P}$, so the call also records on whose behalf it is made.

Adversary $\mathcal{A}$
pid $:= A$, $\mathbf{P} := \{0,1\}^* \cup \{A, Z\}$, $\mathbf{N} := \mathbf{Std} \cup \mathbf{A} \cup \mathbf{Z}$, $\mathbf{U} := \{(\mathcal{Z}, \text{SERVES})\}$, *par* $:= \bot$

id.FULLA(in) from id′
1: **require** $\text{GUARD}_{\mathcal{A}}(\text{id}, \text{id}')$
2: out $\leftarrow$ SILENCE({id″ : id″ $\neq$ id}, id.A(in) from id′)
3: **return** out

3.5 The Full Interface

Everything the wrapper needs is now in place: GUARD and MEDIATE and SILENCE from Chapter 2, and the register $\mathcal{C}$ of Section 3.1 that supplies **C**. The full interface of a standard functionality is their composition.

Functionality $\mathcal{F}$
pid, **P**, **N**, **U**, *par*

id.FULLOP(in) from id′
1: **require** $\text{GUARD}_{\mathcal{F}}(\text{id}, \text{id}')$
2: **C** $\leftarrow$ (C, id.P).FULLSTATUS() as id
3: **require** $\neg(\text{id}'.\text{F} = A \wedge \text{id}.\text{P} \notin \mathbf{C})$ *// $\mathcal{A}$ only at a corrupt party*
4: MEDIATE(**C**, id.OP, in, id′)
5: out $\leftarrow$ SILENCE({id″ : id″ $\neq$ id}, id.OP(in) from id′)
6: **C** $\leftarrow$ (C, id.P).FULLSTATUS() as id
7: MEDIATE(**C**, id.OP, out, id′)
8: **return** out

Lines 2–4 and 6–7 handle corruption, and are the only place a full interface consults the register. It is read twice because **C** changes as the execution proceeds and a read is only a snapshot: the tests before line 6 use the first, the test on line 7 the second. Membership is monotone, CORRUPT only ever adding, so a party honest at the first read may be corrupt at the second but never the reverse.

Line 3 refuses one combination: the adversary at an honest party, which has no business there. Everything else passes, and MEDIATE settles the rest. So an honest party is addressable by everyone but $\mathcal{A}$ and runs its own core; a corrupt party is addressable by everyone, but only $\mathcal{A}$ reaches its core, every other caller being answered by $\mathcal{A}$ in the party's place. The environment needs no case of its own: whether it claims its own identity or an admitted external one, a corrupt party answers it through $\mathcal{A}$ either way.

That leaves a corrupt party addressed by anyone but $\mathcal{A}$, whose call cannot simply be run, so $\mathcal{A}$ answers in the party's place. Lines 4 and 7 are the same wrapper at two moments. The first catches a party already corrupt on arrival and hands over $\mathtt{in}$, so the core never runs. The second catches one that turned corrupt while its core ran on line 5, and hands over $\mathtt{out}$, no longer the party's to return. Monotonicity makes the two cover the cases exactly once: a party corrupt at the first has already returned, and none is corrupt at the first and honest at the second. Both hooks address $\mathcal{A}$ at $(\mathsf{A}, \mathtt{id}.\mathsf{P})$ through its full interface, so the answering adversary is guarded and silenced like any other callee. Line 8 returns the core's own value, which happens in two cases: for a party honest throughout, and for a corrupt party addressed by $\mathcal{A}$ itself, which the gate admits and which neither hook catches, both testing $\mathtt{id}'.\mathsf{F} \neq \mathsf{A}$.

Remark 3.2 (Corruption mid-call is not preemptive)**.** Nothing interrupts a core that is already running. Suppose the core on line 5 calls the adversary — through $\mathcal{A}(\cdot)$, or through a subroutine that mediates — and the adversary corrupts $\mathtt{id}.\mathsf{P}$ before returning a value. The register records that at once, **C** being its own state, but the core is neither told nor stopped: it resumes with whatever the adversary returned and runs to completion. Only on its return does line 6 read **C** again, find $\mathtt{id}.\mathsf{P}$ in it, and hand $\mathtt{out}$ to MEDIATE on line 7 to be replaced.

So the adversary holds the token twice over such a call, in this order: once inside the core, for the call the core placed, and once after the core has returned, for the mediation of its output. No moment in between lets the corruption take effect, since a party that turns corrupt mid-call cannot un-run what it has already done. Nor is the call re-gated: line 3 was decided on the first read, and a call admitted while $\mathtt{id}.\mathsf{P}$ was honest stays admitted. What the wrapper guarantees is therefore weaker than aborting, and sufficient: nothing a newly corrupt party computed reaches its caller unmediated. Calls arriving *after* the corruption are caught on the way in instead, by line 4.

3.6 Executions

In an execution exactly one machine holds the activation token. A call $\mathtt{out} \leftarrow \mathtt{id}.\textsc{Op}(\mathtt{in})$ abbreviates suspending the caller with a continuation and passing the token to interface Op of $\mathcal{F}$ at identity $\mathtt{id}$. Every call between machines addresses a full interface, and FullOp is what Exec dispatches; a core is invoked only by the wrapper that serves it, inside the machine that carries it, and never across a token transfer. A call addressed to an identity no machine occupies is answered with REJ. An instance may hold several outstanding continuations and may be re-entered while suspended, in particular by $\mathcal{A}$.

Definition 3.3 (Machines and occupancy)**.** A *machine* is anything an execution runs: the members of the system, and whatever the execution supplies alongside them. A machine *occupies* a finite set of identities — for a functionality, the $\textsc{IDs}(\mathcal{F})$ of Definition 1.1 — and Exec dispatches to it exactly the calls addressed to identities it occupies. The occupants of one execution carry pairwise disjoint identity sets, and Exec is defined for no other family. An *occupant of the adversary slot* is a machine occupying the identities of process id A and no others; the adversary of Section 3.4 is one, and so is every simulator of Chapter 4. An *activation* is a maximal interval during which one machine holds the token.

Every call carries a claimed id $\mathtt{id}'$. Standard functionalities may claim only the identity of the interface placing the call; $\mathcal{Z}$ and $\mathcal{A}$ may claim others, subject to the checks of Chapter 2. Those checks are the guard, which tests $\mathtt{id}'$ against the interface's identity $\mathtt{id}$ and against the functionality's parameters, and we specify them with **require** statements. A failed guard returns the distinguished value REJ to the sender, and REJ is a refusal rather than a value: no machine may return it as its own output. The rule is enforced rather than assumed — a core whose code would return REJ, and a mediated answer of REJ from the adversary's slot, are delivered as $\perp$ instead — so a caller receiving REJ learns that a guard refused its call, and nothing else can produce one. In particular $\text{REJ} \neq \perp$.

Randomness is per machine: each machine carries its own coin tape, infinite, uniform, and independent of every other machine's, read only when its code samples. Probabilities over an execution are over these

tapes jointly. Bundling machines into one occupant — as Chapter 4 and Section 4.4 do — does not merge tapes: a hosted copy's coins are what they were before absorption.

Besides the system π, the execution carries three machines: the environment $\mathcal{Z}$, the adversary $\mathcal{A}$, and the corruption register $\mathcal{C}$, whose state $\mathbf{C}$ every wrapper reads. By Chapter 1 none is a member of π, which makes line 1 unambiguous — the register is initialized once, as an argument, not a second time as a member — and keeps π the object a designer wrote. The procedure initializes the three, then activates the environment at (Z, Z) as though it had called itself. That first activation goes through $\mathcal{Z}$'s full interface like any other, so the environment is silenced from the outset, and $\text{GUARD}_{\mathcal{Z}}$ passes trivially, the claimed and target identities being the same. Whatever the environment returns is the outcome.

Execution EXEC
system π, environment $\mathcal{Z}$, adversary $\mathcal{A}$, corruption register $\mathcal{C}$

EXEC$(\pi, \mathcal{Z}, \mathcal{A}, \mathcal{C})$:
1: $\mathcal{C}$.INITIALIZE(); $\mathcal{F}$.INITIALIZE() for every $\mathcal{F} \in \pi$
2: **handle** (Z, Z).FULLZ() from (Z, Z) **with** *// $\mathcal{Z}$ is activated first and holds the token*
3: **return** out $\mapsto$ out
4: id.FULLOP(in) from id′; k $\mapsto$ k(id.FULLOP(in) from id′)

Read as a handler [12, 4], the execution has exactly one operation: placing a call. Line 2 activates the environment, the only activation not placed by a machine. Line 4 handles every call thereafter, and the continuation k is where the token comes back: suspending the caller, passing the token to id, and resuming the caller with the answer are precisely what invoking k on that answer means. Exactly one machine holds the token throughout. The **return** clause is reached when the environment's program halts, and its output is the execution's.

CHAPTER 4

UC Emulation and Composition

4.1 Subsystem Replacement

A protocol is normally written against a subroutine later supplied by another protocol. Replacing one by the other is a set operation on systems.

Definition 4.1 (Subsystem replacement)**.** Let ρ be a system and $\varphi \subseteq \rho$ a subsystem of it. For a system π, the *replacement of* φ *by* π *in* ρ is

$$\rho[\pi/\varphi] \;:=\; (\rho \setminus \varphi) \cup \pi.$$

Nothing so far stops the incoming π from landing on a process id the surrounding protocol already uses, which would leave a call ambiguously addressed. We rule this out rather than repair it.

Definition 4.2 (Well-formed replacement)**.** The replacement of φ by π in ρ is *well-formed* if no process id occurs both in $\rho \setminus \varphi$ and in π, that is, if

$$\{\, \mathcal{F}.\mathit{pid} : \mathcal{F} \in \rho \setminus \varphi \,\} \cap \{\, \mathcal{F}.\mathit{pid} : \mathcal{F} \in \pi \,\} \;=\; \varnothing.$$

Proposition 4.3 (Replacement)**.** *Let ρ and π be systems, let $\varphi \subseteq \rho$, and let the replacement of φ by π in ρ be well-formed. Then $\rho[\pi/\varphi]$ is a system. If moreover* $\mathrm{IDs}(\pi) = \mathrm{IDs}(\varphi)$*, then* $\mathrm{IDs}(\rho[\pi/\varphi]) = \mathrm{IDs}(\rho)$.

Proof. The process ids of $\rho[\pi/\varphi]$ are those of $\rho \setminus \varphi$ together with those

of π. Each system carries distinct ids internally and well-formedness makes the two collections disjoint, so no id repeats; the members are standard and finitely many, both inherited from ρ and π, so $\rho[\pi/\varphi]$ is a system. That first claim rests on the well-formedness of the replacement alone, $\mathrm{IDS}(\pi) = \mathrm{IDS}(\varphi)$ entering nowhere in it. Identities need no separate hypothesis: an identity carries its process id in the first component, so members with distinct ids have disjoint identity sets. For the identities themselves,

$$\begin{aligned}\mathrm{IDS}(\rho[\pi/\varphi]) &= \mathrm{IDS}(\rho \setminus \varphi) \cup \mathrm{IDS}(\pi) \\ &= \mathrm{IDS}(\rho \setminus \varphi) \cup \mathrm{IDS}(\varphi) \;=\; \mathrm{IDS}(\rho),\end{aligned}$$

using $\mathrm{IDS}(\pi) = \mathrm{IDS}(\varphi)$ in the middle step and $\varphi \subseteq \rho$ in the last. □

The two halves of the proposition are used differently below. That $\rho[\pi/\varphi]$ is a system lets Chapter 4 speak of it at all, and rests on well-formedness alone. That its identities are those of ρ needs $\mathrm{IDS}(\pi) = \mathrm{IDS}(\varphi)$, and no result in Chapter 4 assumes it: admissibility there is stated for a pair of systems through the union of their identity sets, and Proposition 4.16 and Theorem 4.20 say outright that no relation between the two is used. Where the equality is wanted, blocking supplies it.

Definition 4.2 settles process ids, and Proposition 4.3 identities. Neither addresses dependence, and dependence is where replacement can go wrong: the names $\rho \setminus \varphi$ relies on are still occupied after φ gives way to π, but every requirement is re-evaluated at the new occupant. Two conditions close the gap.

Informally, Theorem 4.4 below says that a well-formed replacement preserves WF when the entries of the incoming π can be discharged in the new world, and some member of π stands in for φ wherever an entry of $\rho \setminus \varphi$ was witnessed inside φ. Hypothesis (ii) is what it means for π to be a drop-in for φ, and emulation does not supply it.

Theorem 4.4 (Replacement preserves well-formedness)**.** *Let ρ and π be systems, let $\varphi \subseteq \rho$, let the replacement of φ by π in ρ be well-formed in the sense of Definition 4.2, and let $\mathrm{WF}(\rho)$. Suppose*

(i) every entry of every $\mathcal{F} \in \pi$ has a witness in $\big(\rho[\pi/\varphi]\big)^+$, in the sense of Definition 1.3; and

(ii) whenever $\mathcal{F} \in \rho \setminus \varphi$ has an entry $(\mathcal{F}', \mathsf{R}) \in \mathcal{F}.\mathbf{U}$ for which some *witness in the sense of Definition 1.3 lies in φ, some $\mathcal{G} \in \pi$ satisfies*

$\mathcal{G}.\mathrm{F} = \mathcal{F}'.\mathrm{F},\ \mathcal{G}.\mathrm{s} = \mathcal{F}.\mathrm{s} \vee \textsc{global}(\mathcal{G})$, *and* $\mathrm{R}(\mathcal{F}, \mathcal{G})$.
Then $\mathrm{WF}\big(\rho[\pi/\varphi]\big)$.

Proof. Write $\rho' := \rho[\pi/\varphi] = (\rho \setminus \varphi) \cup \pi$, a system by the first claim of Proposition 4.3 — the second is not available here, $\textsc{ids}(\pi) = \textsc{ids}(\varphi)$ being no hypothesis of this theorem. Take $\mathcal{F} \in \rho'^{+}$ and an entry $(\mathcal{F}', \mathrm{R}) \in \mathcal{F}.\mathbf{U}$; we must place a witness in ρ'^{+}. There are three cases, by where $\mathcal{F}$ comes from.

If $\mathcal{F} \in \pi$, hypothesis (i) is exactly the claim.

If $\mathcal{F}$ is one of $\mathcal{Z}$, $\mathcal{A}$, $\mathcal{C}$, its entries are settled once and for all, independently of the system: $\mathcal{C}.\mathbf{U}$ is empty, and the entries of $\mathcal{Z}$ and $\mathcal{A}$ name each other, with $\mathcal{A}, \mathcal{Z} \in \rho'^{+}$ and SERVES holding between them as Chapter 1 records. All three are global, so the session conjunct is waived.

If $\mathcal{F} \in \rho \setminus \varphi$, then $\mathcal{F} \in \rho^{+}$, so $\mathrm{WF}(\rho)$ supplies a witness $\mathcal{G} \in \rho^{+}$ for this entry. Either $\mathcal{G} \notin \varphi$, in which case $\mathcal{G}$ lies in $(\rho \setminus \varphi) \cup \{\mathcal{Z}, \mathcal{A}, \mathcal{C}\} \subseteq \rho'^{+}$ and serves again: it is the same machine with the same fields, so $\mathcal{G}.\mathrm{F}$, $\mathcal{G}.\mathrm{s}$ and whatever R reads are unchanged, and $\mathcal{F}$ is unchanged too. Or $\mathcal{G} \in \varphi$, and hypothesis (ii) supplies a member of π meeting the same three conditions in its place. Either way the entry has a witness in ρ'^{+}. □

Hypothesis (ii) is what it means for π to be a drop-in for φ, and emulation does not supply it. Definition 4.11 compares behaviour at the boundary; it says nothing about the declared fields of the machines behind that boundary, and a π that emulates φ perfectly may still fail to admit a caller φ admitted, or serve a party φ served. The check is cheap — it ranges over the entries of $\rho \setminus \varphi$ that pointed into φ, and nothing else — but it is a check, and belongs beside well-formedness of the replacement rather than inside the composition theorem.

4.2 Emulation and Absorption

Everything an execution needs is now fixed, so we can say when one system is at least as secure as another. The two are placed under one and the same environment, so what it may claim must be settled first; that is the role of $\mathcal{Z}.\mathit{par}$ below.

The value of $\textsc{Exec}(\pi, \mathcal{Z}, \mathcal{A}, \mathcal{C})$ is whatever the environment returns. We take an environment to return a bit at the root — its guess as to

which system it was placed in — and read the execution as returning 1 exactly when the environment does. Nothing else in the framework constrains that value, so this is a condition on the environments quantified over, not a consequence. The gap between two executions is then a number, not an informal resemblance.

Definition 4.5 (UC advantage)**.** Let π and φ be systems. For an environment $\mathcal{Z}$, an adversary $\mathcal{A}$ and a simulator $\mathcal{S}$,

$$\mathrm{ADV}^{\mathrm{UC}}_{\pi,\varphi}(\mathcal{Z},\mathcal{A},\mathcal{S}) := \Big|\Pr\big[\textsc{Exec}(\pi,\mathcal{Z},\mathcal{A},\mathcal{C}) = 1\big] - \Pr\big[\textsc{Exec}(\varphi,\mathcal{Z},\mathcal{S},\mathcal{C}) = 1\big]\Big|,$$

the probabilities being over the coins of every machine in the two executions, and $\textsc{Exec} = 1$ abbreviating the event that the execution halts with the environment returning 1.

The two executions differ in the system and in the machine that occupies the adversary's identities: $\mathcal{A}$ on one side, $\mathcal{S}$ on the other. Both are typed by the same pair of definitions.

Definition 4.6 (Adversaries)**.** An *adversary* is a machine that occupies $\mathrm{IDs}(\mathcal{A})$ and no other identities and carries the adversary's interfaces — the wrapper of Section 3.4, guard and silencer, and any further interface that section's machine is given (Chapter 9 adds one) — around an arbitrary core at each of them. The machine of Section 3.4 is one; the dummy of Section 4.4 is another.

Definition 4.7 (Simulators)**.** A *simulator* is an adversary that admits at least $\mathbf{Std} \cup \mathbf{Z}$ — the process ids of every standard functionality together with the environment's.

Every simulator is thus an adversary, which is what lets one stand in an adversary class: Proposition 4.12 and Theorem 4.24 read simulators as adversaries, and the budget containments of Section 4.5 compare the two classes directly. Occupancy and the admitted callers are what standing in the adversary's slot means, so everything Section 3.4 settles about a machine at (A, P) holds of a simulator unchanged: it begins

no call of its own, and its silencer lets it claim only the identity it runs at. Proposition 4.19 asks only occupancy of whatever it is given; Theorem 4.29 uses the wrapper and the admitted callers too, and Remark 4.8 explains why neither is a restriction.

The metric takes the three machines as arguments, so emulation can be stated by quantifying over classes of them and bounding the result. Which environments may be used at all is settled by the pair being compared.

Remark 4.8 (What emulation fixes about the adversary slot)**.** Definition 4.10 below constrains $\mathcal{Z}.\mathit{par}$ and says nothing about the machines in the adversary slot; Definitions 4.6 and 4.7 say everything needed, and it is less than the five fields. The fields *pid* and **P** come free with occupancy: $\mathrm{IDS}(\mathcal{F}) = \{(\mathcal{F}.\mathit{pid}, \mathsf{P}) : \mathsf{P} \in \mathcal{F}.\mathbf{P}\}$, so occupying $\mathrm{IDS}(\mathcal{A})$ pins both, giving $\mathcal{S}.\mathit{pid} = (\mathsf{A}, 0, 0)$ at line 1 and $\textsc{global}(\mathcal{S})$ at line 3; and pinning **P** is what is wanted, a slot occupant serving fewer parties leaving an A-identity unoccupied on one side alone, answered REJ where the other answers. The field *par* is never read — the silenced set is the literal $\{\mathtt{id}'' : \mathtt{id}'' \neq \mathtt{id}\}$ of line 2, carried as code by the required wrapper, and it is no formality: a simulator silenced on *less* could claim a party's identity, pass the corruption gate $\mathtt{id}'.\mathsf{F} = \mathsf{A}$ of line 3, and drive an honest party's interface, a capability the adversary it stands in for lacks. The field **U** is inert, read only by WF, where π^{+} names the reserved $\mathcal{A}$ rather than the slot's occupant.

That leaves **N**, which is why Definition 4.7 requires it outright. Widening is impossible, $\mathcal{A}.\mathbf{N}$ already holding every process id that places a call; narrowing gains a simulator nothing and costs both theorems below. Dropping **Std** turns every mediation hook at a corrupt party into the deemed $\perp$ of Section 3.6 — and Theorem 4.20 quantifies over every ρ, so a simulator admitting only φ's process ids meets others in the first outer world that has them. Dropping **Z** shuts the environment out on the ideal side alone, line 2 at the slot refusing the very instructions the real adversary answers. And nothing in any of this mentions the pair being simulated for, which is what lets Theorem 4.20 hand the hypothesis's simulator on unchanged.

Definition 4.9 (Environments)**.** An *environment* is a machine of the execution that occupies $\mathrm{IDs}(\mathcal{Z})$ together with the identities of finitely many standard functionalities it *hosts* — none for the machine of Section 3.3, copies of the surrounding protocol for the absorbed environments of Definition 4.15 below. It carries the interface of Section 3.3, silencer and parameter *par* included, at each Z-identity, and each hosted functionality's own full interface at that functionality's identities; EXEC dispatches to it at every identity it occupies, and its output is what its $\mathcal{Z}$ part returns.

Definition 4.10 (Admissible environments)**.** For a system π, write $\langle\pi\rangle := \{\,\mathrm{id} \;:\; \mathrm{id}.\mathit{pid}.\mathsf{F} = \mathcal{F}.\mathsf{F} \text{ for some } \mathcal{F} \in \pi\,\}$ for its *name closure*: the identities of every process id carrying a name π uses, occupied or not, in every session and every instance. For systems π and φ, the environments *admissible* for the pair are

$$\mathbb{Z}_{\pi,\varphi} := \big\{\, \mathcal{Z} \;:\; \mathcal{Z}.\mathit{par} \supseteq \langle\pi\rangle \cup \langle\varphi\rangle \text{ and } \mathcal{Z} \text{ hosts at no identity of } \mathrm{IDs}(\pi) \cup \mathrm{IDs}(\varphi) \,\big\}.$$

The second condition is occupancy hygiene: an execution is defined only for occupants with pairwise disjoint identity sets (Definition 3.3), so an environment hosting at an identity one of the systems occupies could not be placed beside it, and admissibility rules it out for both worlds at once.

The containment is justified below. A statement of emulation then fixes three classes: a class $\mathbb{A}$ of adversaries (Definition 4.6), a class $\mathbb{S}$ of simulators (Definition 4.7), and a class $\mathbb{Z} \subseteq \mathbb{Z}_{\pi,\varphi}$ of environments (Definition 4.9). Taking $\mathbb{Z} = \mathbb{Z}_{\pi,\varphi}$ gives the strongest statement of the three, and is the default reading.

Definition 4.11 (UC emulation)**.** Let π and φ be systems, let $\varepsilon \geqslant 0$, and let $\mathbb{Z} \subseteq \mathbb{Z}_{\pi,\varphi}$. We say that π *UC-emulates* φ *within* ε *against* $(\mathbb{A}, \mathbb{S}, \mathbb{Z})$ if

$$\forall \mathcal{A} \in \mathbb{A}\ \ \exists \mathcal{S} \in \mathbb{S}\ \ \forall \mathcal{Z} \in \mathbb{Z} : \qquad \mathrm{ADV}^{\mathrm{UC}}_{\pi,\varphi}(\mathcal{Z}, \mathcal{A}, \mathcal{S}) \leqslant \varepsilon.$$

The order of the quantifiers is the substance of the definition: the

simulator may depend on the adversary but not on the environment, so one $\mathcal{S}$ must work against every $\mathcal{Z} \in \mathbb{Z}$ at once. Otherwise the two executions are one experiment with a single system swapped, $\mathcal{Z}$ being literally the same machine on both sides. Weakening it to $\forall \mathcal{A}\, \forall \mathcal{Z}\, \exists \mathcal{S}$ — a simulator built for the environment it faces — gives a notion Definition 4.11 implies and whose converse we do not prove, the two reading as $\max_{\mathcal{A}} \min_{\mathcal{S}} \max_{\mathcal{Z}}$ and $\max_{\mathcal{A}} \max_{\mathcal{Z}} \min_{\mathcal{S}}$ of one quantity in the concrete form of Section 4.5. The composition results survive the weakening, each proof below instantiating the hypothesis at an environment fixed by the conclusion's; what the order buys is therefore meaning rather than composition, since only under it is $(\varphi, \mathcal{S})$ one system behaving like $(\pi, \mathcal{A})$ whoever is watching — an ideal world one can name and hand on.

Definition 4.10 is what makes the comparison meaningful. Since $\mathcal{Z}.\mathit{par}$ silences $\mathcal{Z}$, it may claim no identity inside it, and three kinds must be kept out of reach. An identity in $\mathrm{IDs}(\pi) \triangle \mathrm{IDs}(\varphi)$ is occupied in one world and free in the other, so an environment allowed to speak as one is told which world it is in before anything happens, and no simulator can repair that. An identity internal to *both* separates nothing, but an environment claiming it speaks as a component of the system rather than from outside, which an environment is not for. And an *unoccupied* identity of a used name is as good as an occupied one, because GUARD admits callers by name: a local callee admits every instance of its parent's name, line 2 asking no more, so an environment claiming a fresh instance of that name drives internal subroutines exactly as their callers do — and against a blocked pair (Chapter 6) it tells the worlds apart outright, the blocker refusing a call the genuine occupant admits. Silencing the closure disallows all three at once. What stays claimable is fresh *names*, and those reach only what admits all of **Std**: the global functionalities, shared between the two worlds in any comparison where they are not themselves at issue.

Silencing governs what $\mathcal{Z}$ may *claim*, and calling is a second channel it does not touch. An identity of $\mathrm{IDs}(\pi) \triangle \mathrm{IDs}(\varphi)$ may be called in either world: where nothing occupies it the answer is REJ (Section 3.6), and where something does the answer is REJ again — provided that something refuses **Z** at line 2. The proviso is a condition on the systems being compared rather than on the environment class, and a pair breaking it is simply not emulable, no simulator being able to invent an answer the ideal world has no machine to give. It is why the local

subroutines of Chapter 20 keep Z out of their $\mathbf{N}$.

Smaller sets being less restrictive, the least restricted admissible environments are those whose *par* is the closure exactly — and the blocking of Chapter 6 delivers them for a compatible pair:

$$\mathrm{IDs}(\overline{\pi}) \;=\; \mathrm{IDs}(\overline{\varphi}) \;=\; \mathrm{IDs}(\pi) \cup \mathrm{IDs}(\varphi)$$

by the second claim of Theorem 6.3, and a blocker copies its name from the member it stands in for, so

$$\langle \overline{\pi} \rangle \;=\; \langle \overline{\varphi} \rangle \;=\; \langle \pi \rangle \cup \langle \varphi \rangle.$$

So $\mathcal{Z}.\mathit{par} := \langle \overline{\pi} \rangle$ is not merely a value the two blocked systems can share, but the weakest restriction the pair admits.[1]

One property of the relation is already available, and it is of a different kind from everything that follows. The composition results below — substitution, parallel composition, the completeness of the dummy adversary — are corollaries of a single absorption lemma, each regrouping one execution and reading the same advantage twice. Transitivity regroups nothing. Its proof is the triangle inequality over Definition 4.5 together with the check that the conclusion's class is admissible for the outer pair, so it belongs to the metric rather than to the machinery and needs none of the apparatus of the rest of this section. It is stated here for that reason.

Proposition 4.12 (Transitivity)**.** *Let π, φ and ψ be systems and let $\mathbb{Z} \subseteq \mathbb{Z}_{\pi,\varphi} \cap \mathbb{Z}_{\varphi,\psi}$. If π UC-emulates φ within ε_1 against $(\mathbb{A}, \mathbb{S}, \mathbb{Z})$ and φ UC-emulates ψ within ε_2 against $(\mathbb{S}, \mathbb{S}', \mathbb{Z})$, then π UC-emulates ψ within $\varepsilon_1 + \varepsilon_2$ against $(\mathbb{A}, \mathbb{S}', \mathbb{Z})$.*

Proof. First, the conclusion is a legal statement of emulation. A $\mathcal{Z}$ in both classes has $\mathcal{Z}.\mathit{par} \supseteq \langle \pi \rangle \cup \langle \varphi \rangle$ and $\mathcal{Z}.\mathit{par} \supseteq \langle \varphi \rangle \cup \langle \psi \rangle$, hence $\mathcal{Z}.\mathit{par} \supseteq \langle \pi \rangle \cup \langle \psi \rangle$; and it hosts at no identity of $\mathrm{IDs}(\pi) \cup \mathrm{IDs}(\varphi)$ nor of $\mathrm{IDs}(\varphi) \cup \mathrm{IDs}(\psi)$, hence at none of $\mathrm{IDs}(\pi) \cup \mathrm{IDs}(\psi)$ — so both clauses of Definition 4.10 hold and $\mathbb{Z} \subseteq \mathbb{Z}_{\pi,\psi}$; the three systems need not be related for this. Now let $\mathcal{A} \in \mathbb{A}$. The first emulation supplies $\mathcal{S} \in \mathbb{S}$,

[1] Quantifying over all environments regardless of *par* is coherent but too strong to be useful: the simulator would have to reproduce φ's internal traffic call by call. Letting the simulator *spawn functionalities* at identities φ does not occupy would relax this, leaving only $\mathrm{IDs}(\varphi) \setminus \mathrm{IDs}(\pi)$ to be withheld — empty for the usual pairing; what spawning costs in composition we leave to future work.

and since the second quantifies its adversaries over $\mathbb{S}$, reading $\mathcal{S}$ as an adversary against φ yields $\mathcal{S}' \in \mathbb{S}'$. For every $\mathcal{Z} \in \mathbb{Z}$ the triangle inequality gives

$$\begin{aligned} \mathrm{Adv}^{\mathrm{UC}}_{\pi,\psi}(\mathcal{Z},\mathcal{A},\mathcal{S}') &\leqslant \mathrm{Adv}^{\mathrm{UC}}_{\pi,\varphi}(\mathcal{Z},\mathcal{A},\mathcal{S}) + \mathrm{Adv}^{\mathrm{UC}}_{\varphi,\psi}(\mathcal{Z},\mathcal{S},\mathcal{S}') \\ &\leqslant \varepsilon_1 + \varepsilon_2, \end{aligned}$$

since the middle term $\Pr[\textsc{Exec}(\varphi,\mathcal{Z},\mathcal{S},\mathcal{C}) = 1]$ cancels. As $\mathcal{S}'$ depends on $\mathcal{A}$ alone, it is the simulator required. □

Three constructions below — the absorbed environment here, and the two of Section 4.4 — fold several machines into one occupant of the execution. They share a shape, which we name once.

Definition 4.13 (Bundle)**.** A *bundle* runs several machines together as a single occupant of the execution, passing the token among them as EXEC does and initialising each as EXEC would. It occupies a declared set of identities, which the members' own identities must cover, and takes its output and parameters from a designated member. A call between two hosted machines is served inside, by the bundle's own routing and *without being placed on the execution*; a call arriving from outside is served by the member carrying the target identity, through that member's full interface, guard included; a call a hosted machine places outward is placed on the execution when its target lies in a declared *outward set*, and answered REJ otherwise. An instance may in addition route particular calls between its members by hand — interposing one before another, even when the target lies outside every member's identities — and such an explicit routing governs the calls it names, leaving only the rest to the target rule above.

How a hosted machine's outward call is claimed turns on occupancy, since a silencer wraps the core of one interface and reaches no further (Chapter 2). A member *whose identities the bundle occupies* leaves its calls under its own identity, meeting its own silencer alone; a member *whose identities the bundle does not occupy* has none to claim, so the bundle rewrites each of its outward calls to leave under an identity it does hold — which identity being part of what an instance declares, alongside its outward set — and the rewritten claim meets that interface's silencer. Which case a member is in is

what keeps a bundle faithful to the execution it stands for.

That internal traffic is *served* rather than *placed* is what lets a bundle occupy the adversary slot of a system whose cores call it responsively, and it is worth saying why the distinction is not a quibble. The wrapper RESPOND (Definition 9.1) refuses every call the responder *places*, so a bundle obliged to place its internal calls could reach none of its own members while answering one — and the simulator of Chapter 20, which drives three hosted hybrids inside each of two responsive answers, could not run at all. Serving them inside leaves the token where it was, which is the whole of what responsiveness asks. What a bundle still may not do inside a responsive answer is reach anything it does not host: the register and the shared functionalities are outside it, so their calls are placed, and refused.

Definition 4.13 already says that a hosted machine comes in two kinds, according to whether the bundle occupies its identities, and the difference is the whole of what follows. A machine whose identities the bundle keeps disappears from the execution's dispatch and is served inside; one whose identities the bundle cedes must leave them to somebody, and its calls must be placed by that somebody on its behalf. Both moves come up — the first absorbs a protocol, the second an adversary — so we define them together and prove them right once.

Definition 4.14 (Absorption)**.** Let $\mathcal{Z}$ be an environment and H a finite set of machines of an execution, split as $H = H_{\text{KEEP}} \uplus H_{\text{CEDE}}$, the members of H_{CEDE} being occupants of the adversary slot. The *absorption* of H into $\mathcal{Z}$ is the bundle (Definition 4.13) $\mathcal{Z}^H$ of $\mathcal{Z}$ with one copy of each member of H; it occupies $\text{IDs}(\mathcal{Z}) \cup \text{IDs}(H_{\text{KEEP}})$, takes its output and *par* from $\mathcal{Z}$, and has as outward set the identities the surrounding execution still serves. A member of H_{KEEP} keeps its identities, so its calls leave under its own claim and calls addressed to it are served inside the bundle. A member of H_{CEDE} keeps none, so the bundle rewrites each outward call $\mathrm{id}''.\text{FULLOP}(x)$ it places while running at (A, P) as the instruction

$$(A, P).\text{FULLA}\big(\mathrm{id}''.\text{OP},\ x\big) \ \text{ as } (Z, P),$$

and the identities it cedes are occupied in the new execution by a *relay* for it, which throughout is the dummy $\mathcal{D}$ of Section 4.4.

We state the ceding case for the adversary slot alone. It is the only one used below, and the only one whose interface has an instruction shape to rewrite calls into — a standard functionality serves what its own code names and could not stand in for another machine's calls.

Definition 4.15 (Absorbed environment)**.** Let $\varphi \subseteq \rho$, let π be a system to be put in place of φ, and let $\mathcal{Z}$ be an environment. The *absorbed* environment $\mathcal{Z}^{\rho\setminus\varphi}$ is the absorption (Definition 4.14) of $\rho \setminus \varphi$ into $\mathcal{Z}$ with every member kept and none ceded, so it is the bundle of $\mathcal{Z}$ with one copy of each $\mathcal{F} \in \rho \setminus \varphi$, every copy keeping its identities; it occupies $\mathrm{IDS}(\mathcal{Z}) \cup \mathrm{IDS}(\rho \setminus \varphi)$, takes its output and *par* from $\mathcal{Z}$, and has outward set $\mathrm{IDS}(\pi) \cup \mathrm{IDS}(\varphi) \cup \{\mathtt{id} : \mathtt{id}.\mathsf{F} \in \{\mathsf{A}, \mathsf{C}\}\}$. It is an environment (Definition 4.9), not a functionality, since it spans several process ids. The superscript records the hosted shell only — the outward set reads π as well — so $\mathcal{Z}^{\rho\setminus\varphi}$ is fixed relative to a replacement given by context, and two replacements sharing $\rho \setminus \varphi$ but differing in π name different machines.

Both halves of the outward set are forced by what the absorbed execution has to reproduce. The first is the slot, whose identities go to π or to φ according to which world the absorbed environment sits in. We take the union rather than $\mathrm{IDS}(\varphi)$ alone so the same machine serves in both, which matters now that the two need not occupy the same identities. The second is the two reserved ids A and C, named as ids rather than as machines because the adversary slot holds $\mathcal{S}$ on the ideal side while Proposition 4.19 needs one and the same absorbed machine on both. They are there because the hosted machines are *full* interfaces rather than bare cores: each reads the register on lines 2 and 6 of Section 3.5, and each hands a call at a corrupt party to the adversary through MEDIATE, which hooks to $(\mathsf{A}, \mathtt{id}.\mathsf{P})$. Neither id belongs to any system. Without this half a call to a hosted functionality would be refused at its first line, and a corrupt party's hook would be refused instead of reaching the adversary: the absorbed execution would stop being the outer one regrouped exactly when $\mathbf{C}$ met $\rho \setminus \varphi$. Everything else is answered with REJ, nothing serving it in the outer execution either.

The parameter is carried over untouched, and it must be. It is the silenced set of $\mathcal{Z}$'s own interface, so any change to it changes which calls $\mathcal{Z}$ may place — and $\mathcal{Z}$ is the one machine that has to behave identically

on both sides. One might think the hosted functionalities need $\mathrm{IDs}(\rho \setminus \varphi)$ removed from it to speak as themselves, but they do not: each is a full interface, and line 5 of its own wrapper is what licenses its claim, $\mathcal{Z}$'s reaching no further than $\mathcal{Z}$'s own core (Definition 4.13). Removing those identities would licence nothing new for them, and would unsilence $\mathcal{Z}$ on exactly the identities it was silenced on before absorption.

Absorbing thus turns an environment for the outer pair into one for the inner pair, preserving admissibility. The register needs no special treatment: being a member of no system it lies in neither φ nor $\rho \setminus \varphi$, so the absorbed environment does not host it and blocking builds no blocker for it. Like $\mathcal{Z}$ and $\mathcal{A}$ it is supplied by EXEC, the same single machine on both sides of every comparison. Its hosted copies keep their identities, so by Definition 4.13 they leave their calls under their own claims and meet their own silencers — not $\mathcal{Z}$'s — which is what the regrouping below needs; the constructions of Section 4.4 are the opposite case, hosting an adversary whose identities the bundle does not occupy.

Proposition 4.16 (Absorption preserves admissibility)**.** *Let ρ and π be systems, let $\varphi \subseteq \rho$, and let the replacement of φ by π in ρ be well-formed. Then*

$$\mathcal{Z} \in \mathbb{Z}_{\rho[\pi/\varphi],\,\rho} \qquad \Longrightarrow \qquad \mathcal{Z}^{\rho\setminus\varphi} \in \mathbb{Z}_{\pi,\varphi}.$$

Proof. For the *par* half, absorption carries the field over untouched, so it is enough that

$$\langle \rho[\pi/\varphi] \rangle \cup \langle \rho \rangle \;\supseteq\; \langle \pi \rangle \cup \langle \varphi \rangle.$$

Every name φ uses is used by ρ, since $\varphi \subseteq \rho$, and every name π uses is used by $\rho[\pi/\varphi]$, since $\pi \subseteq \rho[\pi/\varphi]$. For the hosting half, $\mathcal{Z}^{\rho\setminus\varphi}$ hosts at $\mathrm{IDs}(\rho \setminus \varphi)$ beyond whatever $\mathcal{Z}$ hosted: the latter avoids $\mathrm{IDs}(\rho[\pi/\varphi]) \cup \mathrm{IDs}(\rho)$, which contains $\mathrm{IDs}(\pi) \cup \mathrm{IDs}(\varphi)$; and $\mathrm{IDs}(\rho \setminus \varphi)$ meets $\mathrm{IDs}(\varphi)$ nowhere, the two being disjoint subsystems of ρ, and meets $\mathrm{IDs}(\pi)$ nowhere by well-formedness of the replacement — the one place the hypothesis is used, and it is used for typing alone. No relation between $\mathrm{IDs}(\pi)$ and $\mathrm{IDs}(\varphi)$ enters. One thing the statement does not say: the absorbed machine *claims* identities inside its own *par*, each hosted copy claiming its own. That is no contradiction — by Definition 4.13 a hosted copy's claims meet its own silencer and never the environment's, and admissibility is a condition on the field, read where the field applies. □

What makes the theorem work is that absorption regroups an execution without changing it. Since the same correspondence serves Section 4.4, we set it down before the lemma.

Configurations. By a *configuration* of an execution we mean its state between activations: the state of every machine, the unread part of every coin tape, which machine holds the token and where in its code, and the callers suspended awaiting answers. Calls nest, and Section 3.6 lets an instance hold several outstanding continuations, so the last of these is a chain rather than a set: one entry per call placed and not yet answered, in the order placed, recording the caller and the call it waits on.

The two executions do not run the same machines — that is what absorbing does — but they run the same ones once $\mathbb{Z}^{\mathsf{H}}$ is counted as the machines it bundles rather than as one, and but for the relays a ceded identity needs. Definition 4.13 passes the token among those as EXEC does, so each holds it in the same sense on either side. Counted so, the machines of the two executions correspond one to one apart from the relays, and we write ι for the bijection: the hosted copy of each member of H to that member, the hosted $\mathbb{Z}$ to $\mathbb{Z}$, and everything the execution still runs, $\mathcal{C}$ included, to itself.

Definition 4.17 (Regrouping correspondence)**.** A configuration c of the left execution and a configuration $\hat{c}$ of the right *correspond*, written $c \approx \hat{c}$, when

(C1) every machine of c and its image under ι are in the same state — $\mathcal{C}$ included, so the two agree on **C**;

(C2) every machine and its image have the same unread coin tape;

(C3) the same machine holds the token in both, under ι, at the same point of the same operation and with the same local values; and

(C4) the two chains of suspended callers agree under ι: equal length, and for each k the k-th entries name machines ι identifies, waiting on the same call — same target identity, operation, input and claimed identity.

Where the right-hand family carries a relay, ι is a bijection between the machines of the left and those of the right *other than* the relay,

and (C4) is read with the relay's own frames elided: a relayed call contributes one frame the left does not have, resumed exactly once with the answer the left's call returned — save that a REJ the left reads may arrive as $\perp$ on the right, which is the one place refusal-obliviousness is spent.

Only (C4) is at risk from the bracketing. On the left every suspension is recorded by EXEC; on the right they divide between EXEC and the bundle, which records those of the calls internal to it. The clause asks that the two chains agree once the bundle's own are counted where they were placed. Regrouping is exactly that: one chain of suspensions, kept by two dispatchers.

The first activation. The two executions begin in correspondence, and it is worth saying why, since this is the one activation the lemma's case analysis does not reach: it is placed by no machine. Line 1 initialises $\mathcal{C}$ and everything the execution still runs on both sides, Definition 4.14 initialises each hosted copy as EXEC would, and a relay carries no state, so every machine and its image start in the same state, which is (C1). No tape has been read, giving (C2). The operation INITIALIZE places no calls, so nothing is suspended and both chains are empty, giving (C4) — and with it the reason the order in which the bundle initialises its copies cannot be observed: that order can be read off nothing in the configuration. Line 2 then activates $(Z, Z).\text{FULLZ}()$ from (Z, Z). On the left the token goes to $\mathcal{Z}$; on the right it goes to $\mathcal{Z}^H$, where (Z, Z) is carried by the hosted $\mathcal{Z}$, which Definition 4.14 makes the bundle occupy. Both enter FULLZ at the same identity on the same input, which is (C3).

Thereafter every activation is a call placed or a value returned, and the lemma comes to showing that correspondence survives one of them on either side, and that corresponding configurations halt together with the same output. Its paragraphs take those activations case by case.

Lemma 4.18 (Absorption)**.** *Let an execution run the occupants* $\{\mathcal{Z}\} \cup H \cup R$*, with* $\mathcal{C} \in R$ *and* $\mathcal{Z}$ *an environment, let* $H = H_{\text{KEEP}} \uplus H_{\text{CEDE}}$*, and let* $\mathcal{Z}^H$ *be the absorption of Definition 4.14. Ask of each member of* H_{CEDE}*, and of nothing in* H_{KEEP}*, that*

(a) it be refusal-oblivious (Definition 4.28);

(b) *no core of* R *place a responsive call on the identities it cedes (Definition 9.2); and*

(c) $\mathcal{Z}.par$ *contain no* Z*-identity.*

Then, the two families being legal occupant families in the sense of Definition 3.3,

$$\textsc{Exec}\big(\{\mathcal{Z}\} \cup \mathsf{H} \cup \mathsf{R}\big) \qquad \textit{and}$$

$$\textsc{Exec}\big(\{\mathcal{Z}^{\mathsf{H}}\} \cup \mathsf{R} \cup \{\textit{a relay at each ceded identity}\}\big)$$

are identically distributed — equal as distributions, not merely close.

Proof. The argument is one induction on activations, and the five paragraphs below discharge its step. *Machines* pairs the two occupant families and checks that every machine and its image start alike. *Routing* asks where each call lands and finds four cases — internal to the group, leaving it, entering at a kept identity, entering at a ceded one — of which only the last differs between the two families. *Claims* splits on which kind of member placed the call: one keeping its identities meets the same silencers on both sides, while a ceded member has none to claim, so its outward calls are rewritten, and the rewriting must survive three tests — where the bundle may speak from, whether the claim is silenced, and whether it passes the relay's guard. *Refusals* isolates the one value that can differ, a REJ the left reads arriving as $\perp$ on the right, and spends hypothesis (a) on it; hypothesis (b) blocks the mirror problem, a responsive call on a relayed slot. *Token* checks that the two chains of suspensions agree once the bundle's own are counted where they were placed. Hypothesis (c) is spent in *Claims*, on the ceded member's right to claim (Z, P). Two conventions of Chapters 1 and 3.6 are used throughout and are worth restating. Distinct machines draw from independent coins, and bundling machines into one does not merge their tapes, so a hosted copy's randomness is what it was before absorption. And INITIALIZE places no calls, so the order in which the bundle initialises its hosted copies is unobservable. The proof is an induction on activations: the initial configurations correspond, as shown before the lemma; corresponding configurations produce the same next activation on both sides and step to corresponding configurations, each machine and its image reading the same coins in the same order, which is what (C2) of Definition 4.17 preserves; and corresponding

halting configurations output alike, the output being the hosted $\mathcal{Z}$'s by Definition 4.14. The paragraphs below discharge the inductive step: which machines exist and start alike, (C1); where a call goes and what it claims, (C4); and how the token moves, (C3). The first two paragraphs are common to both kinds of member; the two after them are what a ceded member costs.

Machines. The left runs $\{\mathcal{Z}\} \cup \mathsf{H} \cup \mathsf{R}$; the right runs $\{\mathcal{Z}^{\mathsf{H}}\} \cup \mathsf{R}$ together with a relay at each ceded identity, and $\mathcal{Z}^{\mathsf{H}}$ is $\mathcal{Z}$ with one copy of each member of H. Counted as the machines it bundles, the two collections coincide but for the relays, so ι pairs each member of H with its copy, each member of R and $\mathcal{C}$ with itself, and leaves the relays over, which is the reading Definition 4.17 gives them. Line 1 initialises R on both sides, Definition 4.14 initialises the hosted copies as EXEC would, and a relay is stateless, so every machine and its image start alike. Occupancy is disjoint on both sides by hypothesis, so each call has a unique target: on the right the bundle holds $\mathrm{IDs}(\mathcal{Z}) \cup \mathrm{IDs}(\mathsf{H}_{\mathrm{KEEP}})$ and the relays hold what $\mathsf{H}_{\mathrm{CEDE}}$ let go.

Routing. Take a call and ask where it lands. With both endpoints in $\{\mathcal{Z}\} \cup \mathsf{H}$ it is dispatched by EXEC on the left and served inside $\mathcal{Z}^{\mathsf{H}}$ on the right, by the same machine either way. Leaving that group it is dispatched by EXEC on the left and placed outward on the right, its target lying in the outward set, which Definition 4.14 makes the identities the execution still serves. Entering the group at an identity of $\mathsf{H}_{\mathrm{KEEP}}$, it is dispatched by EXEC on the left and served on the right by the hosted copy carrying that identity, which the bundle occupies. Entering at an identity of $\mathsf{H}_{\mathrm{CEDE}}$, it is dispatched to the ceded member on the left and to the relay on the right, which is the one place the two families differ and the subject of the two paragraphs below. Addressed anywhere else it has no machine on either side: EXEC finds no target and $\mathcal{Z}^{\mathsf{H}}$ answers REJ. Guards are evaluated at the callee from the claimed id, so they agree once the claims do.

Claims, for a member that keeps its identities. Each such call carries the same claimed id on both sides, and the two silencers that could change one are the same silencers on both. A hosted interface is governed by its own silencer and not by the environment's, by Definition 4.13: it keeps its identities, so SILENCE at $\mathcal{Z}$ does not reach it. Being a full interface it claims exactly its own identity on either side, line 5 silencing its core on $\{\mathtt{id}'' \neq \mathtt{id}\}$; and the calls its wrapper places, on $\mathcal{C}$ and through MEDIATE, stand outside even that silencer on both sides. The environment's own

interface is governed by *par*, which absorption carries over unchanged, so $\bar{\jmath}$ is computed from the same set at the same identity and admits exactly the same claims. Neither machine gains nor loses a claim.

Claims, for a member that cedes them. Such a member has no identity of its own to claim inside the bundle, so Definition 4.14 rewrites its outward calls, and the rewriting has to survive three tests. First, where the bundle may speak from. Whenever the hosted member runs on behalf of party P — that is, where the left has it running at $(\mathcal{A}, \mathsf{P})$ — the bundle is running at $(\mathcal{Z}, \mathsf{P})$ or at $(\mathcal{Z}, \mathcal{Z})$, or else $\mathsf{P} = \mathcal{Z}$: it is entered either by $\mathcal{Z}$ calling an $\mathcal{A}$-identity, served internally with $\textsc{Guard}_{\mathcal{A}}$ forcing the claimed party to be the target's, and line 3 letting an interface at $(\mathcal{Z}, \mathsf{Q})$ claim party P only when $\mathsf{Q} \in \{\mathsf{P}, \mathcal{Z}\}$ or $\mathsf{P} = \mathcal{Z}$; or by the relay, whose inward call targets $(\mathcal{Z}, \mathsf{P})$ exactly. Second, whether the claim $(\mathcal{Z}, \mathsf{P})$ is silenced: not by the party-tie just settled, not by line 2 since its name is $\mathcal{Z}$, and not by *par*, which hypothesis (c) empties of $\mathcal{Z}$-identities. Third, whether it passes $\textsc{Guard}$ at the relay: $\mathsf{P} \in \mathcal{A}.\mathbf{P}$ at line 1, $\mathbf{Z} \subseteq \mathcal{A}.\mathbf{N}$ with the parties equal at line 2, and $\textsc{Global}$ at line 3. The relay then places the very call the left placed, under the very claim the left claimed, so both sides meet the same conjuncts at the same callee. A call arriving *at* the ceded identity travels the other way by the same three tests: it reaches the relay under the guard it passed on the left, is handed to $\mathcal{Z}$ at $(\mathcal{Z}, \mathsf{P})$ with its caller in the payload — admitted by the relay's own silencer and by $\textsc{Guard}_{\mathcal{Z}}$, $\mathbf{A} \subseteq \mathcal{Z}.\mathbf{N}$ and $\textsc{Global}(\mathcal{Z})$ — and reaches the hosted member's interface at $(\mathcal{A}, \mathsf{P})$ through that interface's guard, the guard that passed on the left.

Refusals. One value can differ. Where a call the ceded member places is refused, the left reads $\textsc{Rej}$ from its own call, while on the right the relay returns that $\textsc{Rej}$ and Section 3.6 delivers it to the hosted member as $\bot$. Hypothesis (a) is exactly what closes this: a refusal-oblivious machine answers alike to the two, so its distribution is unchanged. Hypothesis (b) is what stops the other direction of the same problem, a responsive call on a ceded identity: $\textsc{Respond}$ refuses every call the responder places, and the relay must place one to answer at all, so a responsive call on a relayed slot would be answered $\bot$ where the left answered with substance — which is why no core of R may place one. Nothing else the relay does is visible: $\textsc{Corrupt}$ survives the detour, its gate $\mathtt{id}'.\mathsf{F} = \mathcal{A}$ met by the relay's claim, which is the one call the rewriting must place under an $\mathcal{A}$-identity and the bundle cannot.

Token. $\textsc{Exec}$ has one operation, placing a call, and its continuation

is where the token comes back. On the right the calls internal to $\mathcal{Z}^{H}$ no longer reach that handler, but Definition 4.13 passes the token among the hosted machines as EXEC does, so each internal caller is suspended on its own continuation and resumed with the same answer; and each relayed call adds one frame on the right, resumed exactly once, which is the elision Definition 4.17 allows. Two properties of EXEC are used, and Section 3.6 grants both to every instance: a machine may hold several outstanding continuations, and may be re-entered while suspended. The second is what a call arriving from R needs while $\mathcal{Z}^{H}$ is itself waiting on an outward call. Exactly one machine holds the token at every moment on either side.

Every machine therefore sees the same inputs in the same order and answers from the same tape, so correspondence is preserved at every activation and the two executions agree as distributions — in particular on the probability that $\mathcal{Z}$ returns 1. □

Diagrammatically, the lemma is one equality with its two cases side by side: a kept member disappears into the bundle outright, a ceded one is bridged back out through a relay.

A member of H_{KEEP} — like all of $\rho \setminus \varphi$ in Theorem 4.20's figure above — is absorbed outright: $\mathcal{Z}^{H}$ occupies its identity, so R reaches it only as part of the bundle. A member of H_{CEDE} is hosted too, running on the same tape, but the bundle does not occupy its identity: the relay $\mathcal{D}$ (the dummy of Section 4.4) stands in its place, handing R's calls to $\mathcal{Z}$ tagged with their caller and re-emitting the hosted member's outward calls under its original claim, so neither side can tell a relay sits between them. Hypotheses (a)–(c), asked only of H_{CEDE}, are what keep that indistinguishable: (a) is exactly refusal-obliviousness, since $\mathcal{D}$ turns every REJ it relays into a $\perp$; (b) and (c) rule out a responsive call and a clash on $\mathcal{Z}$'s own claim, the two other ways a relay's extra hop could show. Theorem 4.20's figure is the case $H_{CEDE} = \varnothing$; the mirror case, one ceded occupant standing for the whole adversary slot ($H_{KEEP} = \varnothing$), is what Section 4.4's dummy-adversary construction runs on.

The two halves of the lemma are used apart. Absorbing a protocol cedes nothing, so all three hypotheses fall away and what is left is a pure rebracketing; that is the case Theorem 4.20 runs on, and it is worth stating separately for the emphasis that it costs nothing at all.

Proposition 4.19 (Regrouping). *Let ρ and π be systems, let $\varphi \subseteq \rho$, let the replacement of φ by π in ρ be well-formed, and let $\mathcal{Z}$ be an environment hosting at no identity of* $\mathrm{IDs}(\rho) \cup \mathrm{IDs}(\pi)$. *Then for σ either π or φ, and for any occupant $\mathcal{X}$ of the adversary slot (Definition 3.3),*

$$\textsc{Exec}\big(\rho[\sigma/\varphi],\, \mathcal{Z},\, \mathcal{X},\, \mathcal{C}\big) \qquad \textit{and} \qquad \textsc{Exec}\big(\sigma,\, \mathcal{Z}^{\rho\setminus\varphi},\, \mathcal{X},\, \mathcal{C}\big)$$

are identically distributed — equal as distributions, not merely close.

Proof. Lemma 4.18 at $H = \rho \setminus \varphi$ with $H_{\textsc{cede}} = \varnothing$, so that $\mathcal{Z}^{H}$ is the absorbed environment of Definition 4.15 and no hypothesis of the lemma applies: (a), (b) and (c) are asked of ceded members and there are none. Note $\rho[\varphi/\varphi] = \rho$, so the case $\sigma = \varphi$ is the ideal side. The two occupant families are legal: the left-hand collection is a system in both cases, so no identity is occupied twice — for $\sigma = \pi$ because well-formedness says $\rho \setminus \varphi$ and π share no process id, and for $\sigma = \varphi$ because that collection is ρ itself — while on the right the hosting hypothesis on $\mathcal{Z}$ keeps the bundle clear of $\mathrm{IDs}(\sigma)$. □

The theorem now says that substituting one subsystem for another inside a protocol costs nothing in advantage.

Informally, Theorem 4.20 says: if π UC-emulates φ within ε, then $\rho[\pi/\varphi]$ UC-emulates ρ within the same ε. The adversary and simulator classes carry over unchanged; the environments of the conclusion are those admissible for the pair $(\rho[\pi/\varphi], \rho)$ whose absorbed form $\mathcal{Z}^{\rho\setminus\varphi}$ lies in the class of the hypothesis.

Theorem 4.20 (Composition). *Let ρ and π be systems, let $\varphi \subseteq \rho$, and let the replacement of φ by π in ρ be well-formed in the sense of Definition 4.2. Suppose*

$$\pi \textit{ UC-emulates } \varphi \textit{ within } \varepsilon \textit{ against } (\mathbb{A}, \mathbb{S}, \mathbb{Z}).$$

Then

$$\rho[\pi/\varphi] \textit{ UC-emulates } \rho \textit{ within } \varepsilon \textit{ against } (\mathbb{A}', \mathbb{S}', \mathbb{Z}'),$$

where

$$\mathbb{A}' = \mathbb{A}, \qquad \mathbb{S}' = \mathbb{S}, \qquad \mathbb{Z}' = \{\mathcal{Z} \in \mathbb{Z}_{\rho[\pi/\varphi],\rho} : \mathcal{Z}^{\rho\setminus\varphi} \in \mathbb{Z}\}.$$

Explicitly,

$$\forall \mathcal{A} \in \mathbb{A}' \ \exists \mathcal{S} \in \mathbb{S}' \ \forall \mathcal{Z} \in \mathbb{Z}' :$$
$$\mathrm{ADV}^{\mathrm{UC}}_{\rho[\pi/\varphi],\rho}(\mathcal{Z},\mathcal{A},\mathcal{S}) \leqslant \mathrm{ADV}^{\mathrm{UC}}_{\pi,\varphi}(\mathcal{Z}^{\rho\setminus\varphi},\mathcal{A},\mathcal{S}) \leqslant \varepsilon.$$

Proof. Three relations, of three kinds. The adversary and simulator classes carry over by copying: an adversary is the same machine whichever system it attacks, and nothing typing a simulator mentions the pair it simulates for. The environment class is the one thing computed rather than copied, being the preimage of the hypothesis's class under absorption. The bound is then two steps. Proposition 4.19, applied once on each side, turns the outer advantage into the inner one *exactly* — the same absorbed machine serving both sides, which is what its outward set was chosen to allow — and the emulation hypothesis bounds the inner one. So the first relation of the display is an equality, shown as an inequality only for uniformity with the second. The three relations are of three kinds. The adversaries carry over unchanged because the reduction hands on the very same machine: an $\mathcal{A} \in \mathbb{A}'$ attacking $\rho[\pi/\varphi]$ is read as an $\mathcal{A} \in \mathbb{A}$ attacking π. Any $\mathbb{A}' \subseteq \mathbb{A}$ would serve, and equality is the largest such choice. The simulators carry over for the mirror reason: the $\mathcal{S} \in \mathbb{S}$ the hypothesis supplies is returned as the simulator for the conclusion, so any $\mathbb{S}' \supseteq \mathbb{S}$ would serve, and equality is the smallest. That it may be returned at all is Remark 4.8: nothing typing a simulator mentions the pair it simulates for, so one for (π, φ) already stands in the slot for $(\rho[\pi/\varphi], \rho)$.

The environments are the one place something is computed rather than copied. Here $\mathbb{Z}$ and $\mathbb{Z}'$ are classes for two different pairs of systems, and $\mathbb{Z}'$ is the preimage of $\mathbb{Z}$ under absorption,

$$\mathbb{Z}' = \left((\cdot)^{\rho\setminus\varphi}\right)^{-1}(\mathbb{Z}) \cap \mathbb{Z}_{\rho[\pi/\varphi],\rho},$$

so an outer environment counts exactly when what it becomes on absorption is one the hypothesis covers. It is a genuine class, since $\mathbb{Z}' \subseteq \mathbb{Z}_{\rho[\pi/\varphi],\rho}$ by construction. When $\mathbb{Z}$ is the largest admissible class $\mathbb{Z}_{\pi,\varphi}$, Proposition 4.16 makes the second condition redundant

and the preimage is all of $\mathbb{Z}_{\rho[\pi/\varphi],\rho}$: emulation against the largest class gives the largest class again.

Proposition 4.16 also makes the right-hand advantage defined, needing no closure hypothesis on the environments and no relation between $\mathrm{IDs}(\pi)$ and $\mathrm{IDs}(\varphi)$. The first inequality is Proposition 4.19 applied twice, with $\sigma = \pi$ and $\mathcal{X} = \mathcal{A}$ on the real side and $\sigma = \varphi$ and $\mathcal{X} = \mathcal{S}$ on the ideal one. Subtracting,

$$\mathrm{Adv}^{\mathrm{UC}}_{\rho[\pi/\varphi],\rho}(\mathcal{Z},\mathcal{A},\mathcal{S}) \;=\; \mathrm{Adv}^{\mathrm{UC}}_{\pi,\varphi}\left(\mathcal{Z}^{\rho\setminus\varphi},\mathcal{A},\mathcal{S}\right),$$

so it is in fact an equality: the two experiments are one experiment bracketed differently, and the same $\mathcal{S}$ serves on both sides because the ideal execution is regrouped by the very same absorption. The second inequality is the emulation hypothesis applied to $\mathcal{Z}^{\rho\setminus\varphi}$, which lies in $\mathbb{Z}$ because $\mathcal{Z}$ lies in $\mathbb{Z}'$. The theorem displays the first relation as an inequality only for uniformity with the second. □

Diagrammatically, fixing one $\mathcal{A} \in \mathbb{A}'$, a serving $\mathcal{S} \in \mathbb{S}'$ and one $\mathcal{Z} \in \mathbb{Z}'$, the proof is the commuting square below, drawn in the string-diagram style of [9]: a solid box is a system, a wire is an interface, and the box hanging off the bottom of a system is whichever machine currently occupies its adversarial slot.

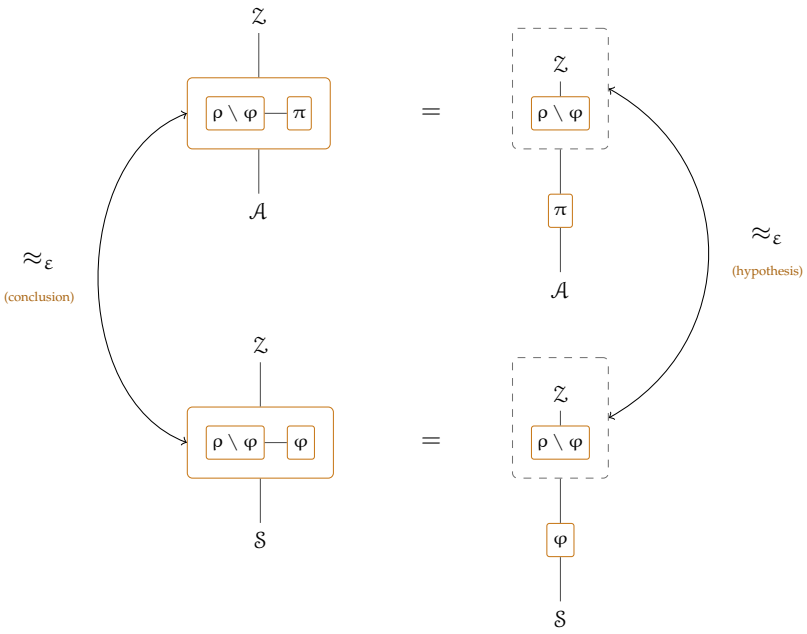


Figure 1: Substitution, as a commuting square. Reading down each column is one execution; the dotted box marks the pair being exchanged. Both horizontal edges are absorption, so both are equalities rather than bounds, and the right edge is the hypothesis. The conclusion on the left is then the same quantity, not merely one with the same bound. (Theorem 4.20.)

The top and bottom edges are Proposition 4.19, applied to $(\pi, \mathcal{A})$ and to $(\varphi, \mathcal{S})$ respectively — regrouping is free, so both are equalities, not mere bounds, which is why the dashed box on the right (the absorbed environment $\mathcal{Z}^{\rho \setminus \varphi}$ of Proposition 4.19) is the only thing that changes between a column and its neighbour. The right edge is the theorem's hypothesis, applied to that absorbed environment. The left edge — the theorem's conclusion — is then forced: chasing the square shows the two $\approx_\varepsilon$'s are not merely both bounded by the same ε but, by the two equalities, are the very same quantity. Composing n such squares in a chain, one hybrid at a time, is exactly Theorem 4.24 below.

Substitution and transitivity now combine into the form in which composition is actually used. Proposition 4.12 was stated with the metric above, its proof needing none of this machinery; here is where it does its work.

Corollary 4.21 (Composition with a specification)**.** *Let π, ρ and ψ be systems, let $\varphi \subseteq \rho$, let the replacement of φ by π in ρ be well-formed, and let π UC-emulate φ within ε_1 against $(\mathbb{A}, \mathbb{S}, \mathbb{Z})$. Write $\mathbb{Z}'$ for the class of Theorem 4.20. If ρ UC-emulates ψ within ε_2 against $(\mathbb{S}, \mathbb{S}', \mathbb{Z}')$, then $\rho[\pi/\varphi]$ UC-emulates ψ within $\varepsilon_1 + \varepsilon_2$ against $(\mathbb{A}, \mathbb{S}', \mathbb{Z}')$.*

Proof. By Theorem 4.20, $\rho[\pi/\varphi]$ UC-emulates ρ within ε_1 against $(\mathbb{A}, \mathbb{S}, \mathbb{Z}')$, and $\mathbb{Z}' \subseteq \mathbb{Z}_{\rho[\pi/\varphi],\rho}$. The second hypothesis is a statement of emulation against that same class, so $\mathbb{Z}' \subseteq \mathbb{Z}_{\rho,\psi}$ as well. Both containments needed by Proposition 4.12 therefore hold, and it applies to the two statements with ρ in the middle. □

This is the form in which composition is used. One analyzes ρ with the idealized subroutine φ in place, proving it emulates whatever specification ψ is wanted — against the classes the corollary names, whose adversaries are the first step's simulators; the corollary transfers that conclusion to the system that runs. One hypothesis is easy to miss: emulation against $\mathbb{Z}'$ asserts, as Definition 4.11's side condition, that $\mathbb{Z}' \subseteq \mathbb{Z}_{\rho,\psi}$ — so ψ's names must lie under the *par* of every environment the class retains, and the corollary is vacuous where they do not.

4.3 Parallel Composition

Theorem 4.20 replaces one subsystem. Replacing several at once is not a new argument but n applications of that one along a chain of hybrids; what needs care is the bookkeeping of the three classes down the chain.

The disjointness hypotheses below are meant literally, and are satisfiable only because of the convention of Chapter 1: the register is a machine of the execution and a member of no system, and there is only ever one of it. Counted a member, it would lie in every φ_i and every π_k at once, so no two subsystems of ρ would be disjoint and no two π_k would carry disjoint process ids; the hypotheses would be unsatisfiable for $n \geqslant 2$ and would have to be read modulo $\mathcal{C}$. Keeping the register out of the systems is what lets them be read as written.

Definition 4.22 (Simultaneous replacement)**.** Let ρ be a system and let $\varphi_1, \ldots, \varphi_n \subseteq \rho$ be pairwise disjoint subsystems. For systems

$\pi_1, \ldots, \pi_n$,

$$\rho[\pi_1/\varphi_1, \ldots, \pi_n/\varphi_n] \;:=\; \Big(\rho \setminus \bigcup_{i=1}^{n} \varphi_i\Big) \cup \bigcup_{i=1}^{n} \pi_i.$$

Write $\rho_k := \rho[\pi_1/\varphi_1, \ldots, \pi_k/\varphi_k]$ for $0 \leqslant k \leqslant n$, so that $\rho_0 = \rho$ and $\rho_n = \rho'$. Since the φ_i are pairwise disjoint, φ_k is untouched by the first $k-1$ replacements, so $\varphi_k \subseteq \rho_{k-1}$ and

$$\rho_k \;=\; \rho_{k-1}[\pi_k/\varphi_k].$$

Consecutive hybrids therefore differ by a single replacement, which lets Theorem 4.20 act on each step; the proof below walks the chain as n game hops, one per replacement. Fitting the steps together also needs the classes to be adjustable, and they are.

Lemma 4.23 (Monotonicity)**.** *Suppose π UC-emulates φ within ε against $(\mathbb{A}, \mathbb{S}, \mathbb{Z})$. Then it does so against $(\mathbb{A}^-, \mathbb{S}^+, \mathbb{Z}^-)$ for any $\mathbb{A}^- \subseteq \mathbb{A}$, any class of simulators $\mathbb{S}^+ \supseteq \mathbb{S}$ and any $\mathbb{Z}^- \subseteq \mathbb{Z}$.*

Proof. Definition 4.11 is a $\forall\exists\forall$ statement over the three classes. Narrowing the range of either universal quantifier and widening that of the existential one each preserve it, and $\mathbb{Z}^- \subseteq \mathbb{Z} \subseteq \mathbb{Z}_{\pi,\varphi}$ keeps the result a legal statement of emulation. □

Informally, Theorem 4.24 says that replacing n pairwise disjoint subsystems at once costs the sum of the individual errors: it is Theorem 4.20 applied n times along a chain of hybrids, the adversaries being those of the last replacement, the simulators those of the first, and the environments admissible for every step at once.

Theorem 4.24 (Parallel composition)**.** *Let ρ be a system, let $\varphi_1, \ldots, \varphi_n \subseteq \rho$ be pairwise disjoint, and let $\pi_1, \ldots, \pi_n$ be systems such that the π_k carry pairwise disjoint process ids and, for each k, π_k shares no process id with $\rho \setminus \varphi_k$. Suppose that for each k*

$$\pi_k \text{ UC-emulates } \varphi_k \text{ within } \varepsilon_k \text{ against } (\mathbb{A}_k, \mathbb{S}_k, \mathbb{Z}_k),$$

and write $\mathbb{Z}'_k$ for the class Theorem 4.20 attaches to the k-th step,

$$\mathbb{Z}'_k \;=\; \big\{\, \mathcal{Z} \in \mathbb{Z}_{\rho_k, \rho_{k-1}} \;:\; \mathcal{Z}^{\rho_{k-1} \setminus \varphi_k} \in \mathbb{Z}_k \,\big\}.$$

Assume finally that

$$\mathbb{S}_{k+1} \subseteq \mathbb{A}_k \quad (1 \leqslant k < n), \qquad \mathbb{Z}^\star \subseteq \bigcap_{k=1}^{n} \mathbb{Z}'_k.$$

Then $\mathbb{Z}^\star \subseteq \mathbb{Z}_{\rho',\rho}$, *and*

$$\rho' := \rho[\pi_1/\varphi_1, \ldots, \pi_n/\varphi_n] \ \textit{UC-emulates} \ \rho$$
$$\textit{within} \sum_{k=1}^{n} \varepsilon_k \ \textit{against} \ (\mathbb{A}_n, \mathbb{S}_1, \mathbb{Z}^\star).$$

Proof. A hybrid argument over n steps, and the work is in checking that each step is a legal instance of Theorem 4.20 before any of them is taken. Three preliminaries do that. The first shows each replacement is well-formed and each ρ_k a system, by induction on k, so that the composition theorem applies at every step. The second shows the class $\mathbb{Z}^\star$ is admissible for every intermediate pair at once, and — taking the union over k — for the outer pair too, so that the conclusion is itself a legal statement of emulation. The third builds the chain of simulators *downwards*, $\mathcal{S}_{n+1} := \mathcal{A}$ and each $\mathcal{S}_k$ supplied by the k-th step for the adversary $\mathcal{S}_{k+1}$, which is what makes $\mathcal{S}_1$ depend on $\mathcal{A}$ alone. Only then is the hybrid taken: consecutive games differ in one replaced subsystem and one adversary, that difference is one application of Theorem 4.20, and summing the n hops telescopes to $\sum_k \varepsilon_k$. First, each step is well-formed. The members of $\rho_{k-1} \setminus \varphi_k$ are those of $\rho \setminus \varphi_k$ the first $k-1$ replacements leave in place, together with $\pi_1, \ldots, \pi_{k-1}$; by hypothesis π_k shares a process id with neither group. So the replacement of φ_k by π_k in ρ_{k-1} is well-formed in the sense of Definition 4.2. Each ρ_k is a system, by induction on k: $\rho_0 = \rho$ is one, and $\rho_k = \rho_{k-1}[\pi_k/\varphi_k]$ is one by Proposition 4.3 from that well-formed replacement. So Theorem 4.20 and Proposition 4.16, which ask ρ_{k-1} and π_k to be systems, apply to the k-th step.

Next, admissibility. Let $\mathcal{Z} \in \mathbb{Z}^\star$ and fix k. From $\mathbb{Z}^\star \subseteq \mathbb{Z}'_k \subseteq \mathbb{Z}_{\rho_k,\rho_{k-1}}$ we get

$$\mathcal{Z}.\mathit{par} \supseteq \langle \rho_k \rangle \cup \langle \rho_{k-1} \rangle,$$

so $\mathcal{Z}$ is admissible for the k-th pair of hybrids and the advantage $\mathrm{ADV}^{\mathrm{UC}}_{\rho_k,\rho_{k-1}}$ below is defined. Proposition 4.16, applied to the replacement of φ_k by π_k in ρ_{k-1}, then places the absorbed environment in

$\mathbb{Z}_{\pi_k,\varphi_k}$: its parameter is $\mathcal{Z}$'s, which already contains the identities of π_k and of φ_k, so the inner advantage $\mathrm{ADV}^{\mathrm{UC}}_{\pi_k,\varphi_k}$ is defined too. Taking the union of the displayed containments over all k,

$$\mathcal{Z}.\mathit{par} \;\supseteq\; \bigcup_{k=0}^{n} \langle \rho_k \rangle \;\supseteq\; \langle \rho' \rangle \cup \langle \rho \rangle,$$

and likewise $\mathcal{Z}$ hosts at no identity of $\bigcup_k \mathrm{IDs}(\rho_k) \supseteq \mathrm{IDs}(\rho') \cup \mathrm{IDs}(\rho)$ — the hosting clause of each $\mathbb{Z}_{\rho_k,\rho_{k-1}}$, at $k = n$ and $k = 1$ — so both clauses of Definition 4.10 give $\mathbb{Z}^\star \subseteq \mathbb{Z}_{\rho',\rho}$ and the conclusion is itself a legal statement of emulation.

Fix $\mathcal{A} \in \mathbb{A}_n$ and put $\mathcal{S}_{n+1} := \mathcal{A}$. For $k = n, n-1, \ldots, 1$ let $\mathcal{S}_k \in \mathbb{S}_k$ be the simulator the k-th step supplies for the adversary $\mathcal{S}_{k+1}$. This is legitimate at every index: at $k = n$ because $\mathcal{S}_{n+1} = \mathcal{A} \in \mathbb{A}_n$, and at $k < n$ because $\mathcal{S}_{k+1} \in \mathbb{S}_{k+1} \subseteq \mathbb{A}_k$. The chain of simulators is built downwards, each from the one before it, and $\mathcal{S}_1$ depends on $\mathcal{A}$ alone.

Now fix $\mathcal{Z} \in \mathbb{Z}^\star$ and consider the $n+1$ games

$$G_k \;:=\; \textsc{Exec}(\rho_k, \mathcal{Z}, \mathcal{S}_{k+1}, \mathcal{C}), \qquad p_k \;:=\; \Pr[\,G_k = 1\,], \qquad 0 \leqslant k \leqslant n.$$

Here G_0 is the execution of ρ under $\mathcal{S}_1$ and G_n that of ρ' under $\mathcal{A}$, so the sequence starts at ρ_0 and ends at ρ_n. Consecutive games differ in one replaced subsystem and in the machine playing the adversary, and that difference is exactly one application of Theorem 4.20: for $1 \leqslant k \leqslant n$,

$$\begin{aligned} |\,p_k - p_{k-1}\,| \;&=\; \mathrm{ADV}^{\mathrm{UC}}_{\rho_k,\rho_{k-1}}(\mathcal{Z}, \mathcal{S}_{k+1}, \mathcal{S}_k) \\ &\leqslant\; \mathrm{ADV}^{\mathrm{UC}}_{\pi_k,\varphi_k}(\mathcal{Z}^{\rho_{k-1}\setminus\varphi_k}, \mathcal{S}_{k+1}, \mathcal{S}_k) \;\leqslant\; \varepsilon_k. \end{aligned}$$

The equality is Definition 4.5 read backwards, since G_k and G_{k-1} are the two executions that the advantage $\mathrm{ADV}^{\mathrm{UC}}_{\rho_k,\rho_{k-1}}$ compares. The first inequality is the reduction of Theorem 4.20, available because $\mathcal{Z} \in \mathbb{Z}^\star \subseteq \mathbb{Z}'_k$ by hypothesis, so Lemma 4.23 makes the k-th step a statement against $\mathbb{Z}^\star$. The second is the emulation of φ_k by π_k, applied to the adversary $\mathcal{S}_{k+1}$ and the absorbed environment.

Summing the n hops and telescoping,

$$\mathrm{ADV}^{\mathrm{UC}}_{\rho',\rho}(\mathcal{Z}, \mathcal{A}, \mathcal{S}_1) \;=\; |\,p_n - p_0\,| \;\leqslant\; \sum_{k=1}^{n} |\,p_k - p_{k-1}\,| \;\leqslant\; \sum_{k=1}^{n} \varepsilon_k.$$

As $\mathcal{S}_1 \in \mathbb{S}_1$ was chosen from $\mathcal{A}$ alone and $\mathcal{Z} \in \mathbb{Z}^\star$ was arbitrary, this is the statement of the theorem. □

Diagrammatically, hop k is Theorem 4.20's own square (the figure above its proof) with $\varphi \mapsto \varphi_k$, $\pi \mapsto \pi_k$, $\mathcal{A} \mapsto \mathcal{S}_{k+1}$, $\mathcal{S} \mapsto \mathcal{S}_k$, and the shell $\rho \setminus \varphi$ widened to carry every untouched slot along for the ride. One picture, hop 1 drawn in full and hops 2, 3 continuing compactly from where it ends, $\rho_0 = \rho$ to $\rho_3 = \rho'$:

Z
ρ \ ⋃ᵢ φᵢ
φ1
φ2
φ3
S1
(ρ0 = ρ)
=
Z
ρ \ ⋃ᵢ φᵢ
φ1
φ2
φ3
S1
φ1
≈ε1
(hyp. on π1, φ1)
Z
ρ \ ⋃ᵢ φᵢ
φ2
φ3
S2
π1
=
Z
ρ \ ⋃ᵢ φᵢ
φ1
φ2
φ3
S2
≈ε2
Z
ρ \ ⋃ᵢ φᵢ
π1
φ2
φ3
S3
≈ε3
Z
ρ \ ⋃ᵢ φᵢ
π1
π2
π3
A
(ρ3 = ρ′)
Z
ρ \ ⋃ᵢ φᵢ
π1
φ2
φ3
S2
(continuing from ρ1 above)

Reading left to right: ρ_0 (all φ's) is absorbed — the true remainder of ρ and the untouched φ_2, φ_3 folding into $\mathcal{Z}$ along with it, leaving only φ_1 exposed. Theorem 4.20's hypothesis swaps it for π_1 within ε_1; handing the fold back out reassembles ρ_1, $\mathcal{S}$ advanced from $\mathcal{S}_1$ to $\mathcal{S}_2$ to match. Hops 2 and 3 are the identical square, only compressed to their flat ends — redrawing it twice more would repeat rather than add — with the dotted boxes marking, at each hop, exactly which slot is φ_k (about to flip) and π_k (just flipped). Summing the three hops and telescoping, as in the last lines of the proof above, turns three separate ε_k's, none of which appear in the theorem's own statement, into the one bound $\rho \approx_{\varepsilon_1+\varepsilon_2+\varepsilon_3} \rho'$.

Corruption along the chain. Corruption behaves along the chain as it does at a single step. Within a game the register is one machine underneath $\rho \setminus \varphi_k$, π_k and φ_k alike, keyed by party, so a hop cannot corrupt a party halfway; across games the registers are separate instances and carry nothing between hops. The environment corrupts the same parties at every hop, being the same machine placing the same calls under line 3; the slot does not have to, so the two machines that change per hop — the replaced subsystem and the slot occupant $\mathcal{S}_k \to \mathcal{S}_{k+1}$ — need not corrupt the same parties; but the environment reads $\mathbf{C}$ unmediated (Remark 3.1), so a $\mathcal{S}_k$ corrupting differently is caught by the same ε_k, and commensurateness is again a consequence rather than an assumption. The parties $\rho_{k-1} \setminus \varphi_k$ serves are handled alike: their MEDIATE hooks leave the absorbed environment for the slot, which is why the outward set of Definition 4.15 carries the ids A and C. So the theorem claims only that no environment in $\mathbb{Z}^\star$ tells the corruption patterns apart by more than $\sum_k \varepsilon_k$.

Remark 4.25 (The largest classes)**.** The union of the hybrid closures has a closed form. Each ρ_k consists of the untouched part of ρ together with $\pi_1, \ldots, \pi_k$, so

$$\bigcup_{k=0}^{n} \langle \rho_k \rangle \;=\; \langle \rho \rangle \cup \bigcup_{i=1}^{n} \langle \pi_i \rangle \;=\; \langle \rho' \rangle \cup \langle \rho \rangle.$$

Suppose now that each $\mathbb{Z}_k$ is the largest admissible class $\mathbb{Z}_{\pi_k,\varphi_k}$. Proposition 4.16 then makes the second condition defining $\mathbb{Z}'_k$

redundant, so $\mathbb{Z}'_k = \mathbb{Z}_{\rho_k,\rho_{k-1}}$, and the largest permissible choice of $\mathbb{Z}^\star$ is

$$\bigcap_{k=1}^{n} \mathbb{Z}_{\rho_k,\rho_{k-1}} \;=\; \{\, \mathcal{Z} : \mathcal{Z}.\mathit{par} \supseteq \textstyle\bigcup_{k=0}^{n} \langle \rho_k \rangle \,\} \;=\; \mathbb{Z}_{\rho',\rho}.$$

Parallel composition against the largest classes therefore delivers the largest class again, as the single-step theorem does.

The three classes behave as the chain suggests. The adversaries are those of the *last* replacement, the argument starting from an adversary against $\rho_n = \rho'$; the simulators are those of the *first*, the chain ending by producing a simulator against $\rho_0 = \rho$. Each intermediate class serves twice, as the simulators of one step and the adversaries of the next, which is what $\mathbb{S}_{k+1} \subseteq \mathbb{A}_k$ arranges. The environments must be admissible for every step at once, hence the intersection. The advantage is additive, as a hybrid argument over n steps always makes it.

4.4 Completeness of the Dummy Adversary

The dummy adversary carries out whatever the environment tells it and reports back whatever it is told; it has no strategy of its own. Anything a real adversary achieves, an environment can achieve by driving the dummy, which is why quantifying over adversaries can be replaced by quantifying over environments. This section defines it and proves that, for the systems and classes the theorem makes precise — Remark 9.4 records the one regime, cores placing responsive calls, that the claim does not reach.

What the framework already fixes. The dummy is not a new kind of machine. It is the adversary of Section 3.4 with a particular core, so it carries $\mathcal{A}$'s fields and its calls pass through $\mathcal{A}$'s wrapper. Three consequences settle the shape of the code before a line of it is written.

First, it may claim one identity only. Line 2 of the $\mathcal{A}$ box silences a core on $\{\mathtt{id}'' \neq \mathtt{id}\}$, so every call the dummy places claims $\mathtt{id}$, the identity it runs at. An instruction of the form "call $\mathtt{id}''$.Op *as* $\mathtt{id}'''$" therefore cannot be carried out for an arbitrary $\mathtt{id}'''$: the dummy speaks as $(\mathsf{A}, \mathtt{id}.\mathsf{P})$ or not at all. What the environment chooses is not the claimed id but the party, by instructing the dummy at (A, P) rather

than at (A, Q); the root of $\mathcal{Z}$ may act for any party and so may pick either.

Second, the party of the target is forced. The predicate GUARD asks $\mathtt{id}'.P = \mathtt{id}.P$ of the callee, and the claim is $(A, \mathtt{id}.P)$, so the dummy at (A, P) reaches interfaces of party P and no other. A target that does not admit **A**, or an honest party, refuses the call by line 2 or line 3; the dummy passes the refusal back, and the rule of Section 3.6 delivers it to the dummy's own caller as $\perp$.

Third, its route to the environment is fixed too: $(A, P) \to (Z, P)$, which $\text{GUARD}_{\mathcal{Z}}$ admits because $\mathbf{A} \subseteq \mathcal{Z}.\mathbf{N}$ and the parties agree.

The code. The core dispatches on who called it. A call from the environment is an instruction to carry out; a call from anything else — a mediation hook, or an $\mathcal{A}(\cdot)$ placed inside some core — is a message to hand on. In both directions the payload has the shape MEDIATE already uses, an operation together with a value.

Dummy adversary $\mathcal{D}$
pid $:= A$, $\mathbf{P} := \{0,1\}^* \cup \{A, Z\}$, $\mathbf{N} := \mathbf{Std} \cup \mathbf{A} \cup \mathbf{Z}$, $\mathbf{U} := \{(\mathcal{Z}, \text{SERVES})\}$, *par* $:= \perp$

id.A(in) from $\mathtt{id}'$
1: **if** $\mathtt{id}'.F = Z$ **then**
2: parse in as $(\mathtt{id}''.\text{OP}, x)$ *// unparseable: return $\perp$*
3: **return** $\mathtt{id}''.\text{FULLOP}(x)$ as id *// carry out the instruction*
4: **else**
5: **return** $(Z, \mathtt{id}.P).\text{FULLZ}(\mathtt{id}', \mathtt{in})$ as id *// hand it to $\mathcal{Z}$, caller and all*

Line 3 is the outward direction: the environment names a target interface and an input, and the dummy places exactly that call and returns exactly its answer. Nothing is inspected and nothing remembered — the dummy keeps no state, so it has no INITIALIZE. Line 5 is the inward one: whatever reaches the adversary is handed to the environment together with the identity it came from, and whatever the environment answers is returned to that caller unchanged. The two lines are the same relay read in opposite directions, which is what makes the dummy transparent.

An instruction line 2 cannot parse is answered with $\perp$: the guards answer anything the dummy cannot *place* with REJ, but a payload that names no call at all needs a value of its own, and $\perp$ is what a machine with nothing to say returns.

One detail is worth pausing on. The caller $\mathtt{id}'$ is passed to $\mathcal{Z}$ on line 5 and not merely the payload: without it the environment could not tell which interface is asking, and a relay that loses that cannot be transparent.

Interposing an adversary. Completeness is a statement about moving an adversary from the execution into the machines around it, and needs a name for each move. Both are the same operation seen from two sides: $\mathcal{A}$ is placed between some machine and the dummy interface, so what used to reach $\mathcal{A}$ now reaches it inside, and what $\mathcal{A}$ used to place now leaves as an instruction to $\mathcal{D}$.

Definition 4.26 (Adversary absorbed into the environment)**.** Let $\mathcal{Z}$ be an environment and $\mathcal{A}$ an adversary. The environment $\mathcal{Z}^{\mathcal{A}}$ is the bundle (Definition 4.13) of $\mathcal{Z}$ with one copy of $\mathcal{A}$, the copy *not* keeping its identities; it occupies $\mathrm{IDs}(\mathcal{Z})$ alone and takes its output and *par* from $\mathcal{Z}$. A call $\mathcal{Z}$ places on an A-identity is served internally by $\mathcal{A}$'s interface there, and one $\mathcal{A}$ places on a Z-identity by $\mathcal{Z}$'s. Any other call the hosted $\mathcal{A}$ places, on $\mathtt{id}''.\textsc{FullOp}(x)$ while running at (A, P), is rewritten to the instruction

$$(A, P).\textsc{FullA}\big(\mathtt{id}''.\textsc{Op},\ x\big) \ \text{ as } (Z, P),$$

keyed to the party $\mathcal{A}$ runs at and never its target's, so a call the unabsorbed $\mathcal{A}$ could place only as (A, P) is relayed at that same party and admitted or refused exactly where it was. A call arriving from an A-identity with payload $(\mathtt{id}', \mathtt{in})$ is handed to $\mathcal{A}$'s interface at (A, P), guard included, as a call from $\mathtt{id}'$.

Definition 4.27 (Adversary interposed before a simulator)**.** Let $\mathcal{S}$ be a simulator and $\mathcal{A}$ an adversary. The simulator $\mathcal{S}[\mathcal{A}]$ is the bundle (Definition 4.13) of $\mathcal{S}$ with one copy of $\mathcal{A}$; it occupies $\mathrm{IDs}(\mathcal{A})$ and carries $\mathcal{S}$'s remaining fields — $\mathcal{S}$, not $\mathcal{A}$, being the machine it stands in for, so $\mathcal{S}[\mathcal{A}]$ is a simulator whenever $\mathcal{S}$ is. A call from a Z-identity is served by the hosted $\mathcal{A}$; one from any other process id, a mediation hook in particular, by $\mathcal{S}$. The hosted $\mathcal{A}$'s calls on a Z-identity — its route back to the environment — leave the bundle under (A, P) to the external $\mathcal{Z}$; its other outward calls become the instruction of

Definition 4.26, served by $\mathcal{S}$ at (A, P) as a call from (Z, P); and a call $\mathcal{S}$ places on a Z-identity is handed, past $\text{GUARD}_{\mathcal{Z}}$, to $\mathcal{A}$ at (A, P).

Neither construction is a new kind of machine. The bundle $\mathcal{Z}^{\mathcal{A}}$ occupies what an environment occupies and claims what $\mathcal{Z}$ claimed, so it is admissible exactly when $\mathcal{Z}$ is. It occupies no A-identity, those being the relay's in the right-hand execution, so by Definition 4.13 the hosted $\mathcal{A}$ has no identity of its own to place a call under: hence the rewriting above, and hence the obligation — discharged by Siting below — to check the rewritten claim against $\mathcal{Z}$'s silencer. Absorption in Chapter 4 is the other case of Definition 4.13, which is why a hosted functionality there keeps its claim instead. Meanwhile $\mathcal{S}[\mathcal{A}]$ occupies $\text{IDs}(\mathcal{A})$, which is what standing in the adversary slot means.

The theorem. Emulation *with respect to the dummy alone* is Definition 4.11 with the adversary class cut down to the single machine $\mathcal{D}$. The claim is that nothing is lost.

One mild condition on the adversary is needed, and it is the price of the dummy being a relay.

Definition 4.28 (Refusal-oblivious)**.** An adversary is *refusal-oblivious* if its behaviour is unchanged when any REJ it receives, as the answer to a call it placed, is replaced by $\perp$.

The dummy relays such an answer, and by Section 3.6 its own **return** of a REJ reaches the caller as $\perp$; so where an unabsorbed adversary placing a refused call reads REJ directly, the same adversary driving the dummy reads $\perp$. A refusal-oblivious adversary cannot tell the two apart, which is what the regrouping below needs. The dummy is itself refusal-oblivious — it relays its answer without branching on it — so the hypothesis costs the dummy class nothing, and a real adversary that reads a framework-internal refusal signal is in any case a modelling artefact rather than an attack.

Informally, Theorem 4.29 says that emulation with respect to the dummy alone loses nothing: anything a real adversary achieves, an environment can achieve by driving the dummy, and the simulator for $\mathcal{A}$ is $\mathcal{S}_{\mathcal{D}}[\mathcal{A}]$ — the one simulator the hypothesis supplies, with $\mathcal{A}$ interposed before it — which depends on $\mathcal{A}$ alone and not on $\mathcal{Z}$.

Theorem 4.29 (Completeness of the dummy adversary)**.** *Let π and φ be systems whose cores place no responsive calls (Definition 9.2), let $\varepsilon \geqslant 0$, let $\mathbb{A}$ be a class of refusal-oblivious adversaries (Definition 4.28), let $\mathbb{S}$ be a class of simulators, and let $\mathbb{Z} \subseteq \mathbb{Z}_{\pi,\varphi}$. Suppose*

$$\pi \text{ UC-emulates } \varphi \text{ within } \varepsilon \text{ against } (\{\mathcal{D}\}, \mathbb{S}, \mathbb{Z}).$$

Then

$$\pi \text{ UC-emulates } \varphi \text{ within } \varepsilon \text{ against } (\mathbb{A}, \mathbb{S}', \mathbb{Z}'),$$

where

$$\begin{aligned} \mathbb{S}' &= \{\, \mathcal{S}[\mathcal{A}] \,:\, \mathcal{S} \in \mathbb{S},\ \mathcal{A} \in \mathbb{A} \,\}, \\ \mathbb{Z}' &= \{\, \mathcal{Z} \in \mathbb{Z}_{\pi,\varphi} \,:\, \mathcal{Z}.\mathit{par} \cap \{\mathsf{id} : \mathsf{id}.\mathsf{F} = \mathsf{Z}\} = \varnothing \\ &\qquad \wedge\, \mathcal{Z}^{\mathcal{A}} \in \mathbb{Z} \text{ for every } \mathcal{A} \in \mathbb{A} \,\}. \end{aligned}$$

Explicitly, let $\mathcal{S}_{\mathcal{D}} \in \mathbb{S}$ be the simulator the hypothesis supplies for $\mathcal{D}$. Then

$$\forall \mathcal{A} \in \mathbb{A}\ \ \forall \mathcal{Z} \in \mathbb{Z}' :$$

$$\mathrm{ADV}^{\mathrm{UC}}_{\pi,\varphi}\big(\mathcal{Z}, \mathcal{A}, \mathcal{S}_{\mathcal{D}}[\mathcal{A}]\big) \;\leqslant\; \mathrm{ADV}^{\mathrm{UC}}_{\pi,\varphi}\big(\mathcal{Z}^{\mathcal{A}}, \mathcal{D}, \mathcal{S}_{\mathcal{D}}\big) \;\leqslant\; \varepsilon,$$

so the simulator Definition 4.11 asks for is $\mathcal{S}_{\mathcal{D}}[\mathcal{A}]$, which depends on $\mathcal{A}$ alone and not on $\mathcal{Z}$.

Proof. The whole of the argument is two regroupings, one per world, labelled (R1) and (R2) below. Granting them, the two advantages of the display agree term by term and the hypothesis at $\mathcal{Z}^{\mathcal{A}}$ bounds the second, which is the theorem. Neither is a pure rebracketing — the right-hand executions carry a machine the left-hand ones do not, the relay standing at the ceded identities — so both are the *ceding* half of Lemma 4.18 rather than Proposition 4.19's. (R1) is that lemma applied directly, its three hypotheses being the theorem's three. (R2) is the same argument run again with φ for π and $\mathcal{S}_{\mathcal{D}}$ for $\mathcal{D}$, and it needs an argument of its own because there the adversary moves from one bundle to another rather than out of the execution: what the lemma's proof used of the machine in the slot is recounted and shown to hold of $\mathcal{S}_{\mathcal{D}}[\mathcal{A}]$ as well. The closing paragraph checks the quantifier order — $\mathcal{S}_{\mathcal{D}}[\mathcal{A}]$ depends on $\mathcal{A}$ alone, the hypothesis knowing nothing of $\mathcal{Z}$. Fix $\mathcal{A} \in \mathbb{A}$ and $\mathcal{Z} \in \mathbb{Z}'$, and let $\mathcal{S}_{\mathcal{D}} \in \mathbb{S}$ be the simulator the hypothesis supplies for $\mathcal{D}$. Membership in $\mathbb{Z}'$ gives $\mathcal{Z}^{\mathcal{A}} \in \mathbb{Z}$, so the hypothesis

applies to it, and $\mathcal{Z}^{\mathcal{A}}.par = \mathcal{Z}.par \supseteq \langle\pi\rangle \cup \langle\varphi\rangle$, so the advantage on the right of the display is defined. The conventions of Proposition 4.19 — independent coins, unmerged tapes, call-free INITIALIZE — are in force. The work is two regroupings, one per world:

$$\textsc{Exec}\big(\pi, \mathcal{Z}, \mathcal{A}, \mathcal{C}\big) \;\equiv\; \textsc{Exec}\big(\pi, \mathcal{Z}^{\mathcal{A}}, \mathcal{D}, \mathcal{C}\big), \tag{R1}$$

$$\textsc{Exec}\big(\varphi, \mathcal{Z}, \mathcal{S}_{\mathcal{D}}[\mathcal{A}], \mathcal{C}\big) \;\equiv\; \textsc{Exec}\big(\varphi, \mathcal{Z}^{\mathcal{A}}, \mathcal{S}_{\mathcal{D}}, \mathcal{C}\big), \tag{R2}$$

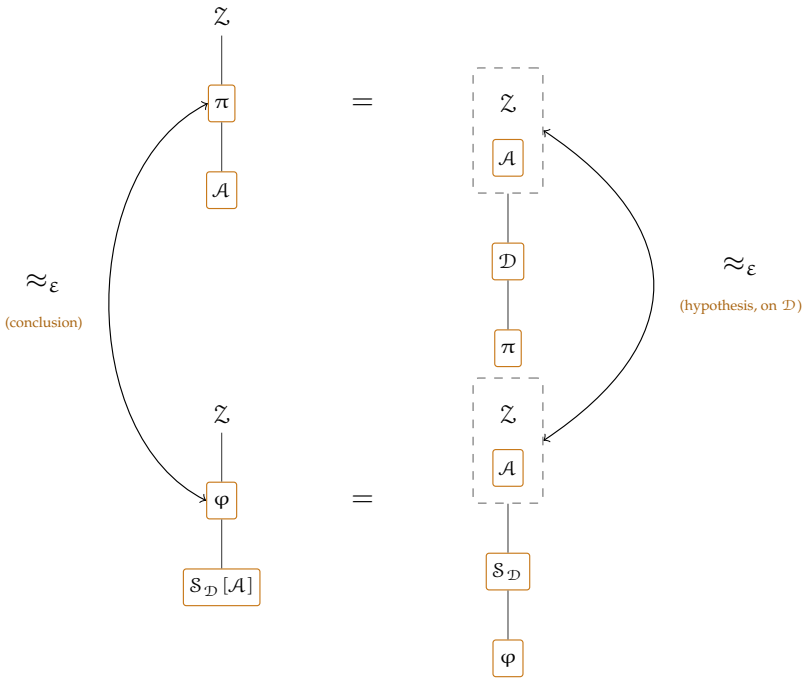
each an equality of distributions; granting them, the two advantages agree term by term and the hypothesis at $\mathcal{Z}^{\mathcal{A}}$ bounds the second by ε, which is the display. Neither is a pure rebracketing — the right-hand executions carry a machine the left-hand ones do not, the relay in the adversary slot — so both are the ceding half of Lemma 4.18 rather than Proposition 4.19's.

Proof of (R1). Lemma 4.18 at $\mathsf{H} = \{\mathcal{A}\}$ with $\mathsf{H}_{\textsc{keep}} = \varnothing$, so that $\mathcal{Z}^{\mathsf{H}}$ is the $\mathcal{Z}^{\mathcal{A}}$ of Definition 4.26 and the relay at the ceded $\textsc{ids}(\mathcal{A})$ is $\mathcal{D}$. Its three hypotheses are the theorem's: $\mathcal{A}$ is refusal-oblivious because $\mathbb{A}$ is; no core of π places a responsive call, which is the hypothesis on the systems; and $\mathcal{Z} \in \mathbb{Z}'$ silences no Z-identity. The occupant families are legal, $\mathcal{Z}^{\mathcal{A}}$ occupying $\textsc{ids}(\mathcal{Z})$ alone and $\mathcal{D}$ the identities $\mathcal{A}$ let go. *Proof of* (R2). Here $\mathcal{A}$ moves from one bundle to another rather than out of the execution, so Lemma 4.18 does not apply on its face and the argument is its ceding half run again with φ for π and $\mathcal{S}_{\mathcal{D}}$ for $\mathcal{D}$. That it transfers is a matter of what the lemma's proof used of the machine in the slot, which is no more than that its guard returns $\textsc{Guard}_{\mathcal{A}}$'s verdicts, that it dispatches on whether the claim has $\mathtt{id}'.\mathsf{F} = \mathsf{Z}$, and — where the hosted $\mathcal{A}$ reads a refused instruction, $\textsc{rej}$ from $\mathcal{S}_{\mathcal{D}}$ inside the bundle on the left against the deemed $\perp$ across the slot on the right — that $\mathbb{A}$ is refusal-oblivious, exactly as in (R1). What differs is which machine holds $\mathcal{A}$ — $\mathcal{S}_{\mathcal{D}}[\mathcal{A}]$ on the left, $\mathcal{Z}^{\mathcal{A}}$ on the right — and Definitions 4.27 and 4.26 make the two perform the same hand-offs, each routed through the receiving machine's guarded interface, the instruction travelling inside a machine on one side and across the execution on the other. The chain $\mathcal{Z} \to \mathcal{A} \to \mathcal{S}_{\mathcal{D}} \to \varphi$ is the same in both, cut at a different point: $\mathcal{S}_{\mathcal{D}}$ meets the same instructions in the same order under the same claims, Siting holds verbatim, and the occupant is judged by the same fields either way — *pid* and **P** by occupancy, and **N** by Definition 4.7, which $\mathcal{S}_{\mathcal{D}}$ meets and $\mathcal{S}_{\mathcal{D}}[\mathcal{A}]$ inherits, carrying $\mathcal{S}_{\mathcal{D}}$'s fields by Definition 4.27; no guard reads the rest. Definition 4.27 routes the left-hand hand-off through the same guarded interface that EXEC serves on the right, and

the hosted $\mathcal{A}$'s own calls back to $\mathcal{Z}$ leave the bundle under (A, P) just as they cross EXEC on the right.

Conclusion. By (R1) and (R2) the two advantages of the display are equal, and the hypothesis at $(\mathcal{D}, \mathcal{Z}^{\mathcal{A}})$ bounds the second by ε. The simulator $\mathcal{S}_{\mathcal{D}}[\mathcal{A}]$ lies in $\mathbf{S}'$ and depends on $\mathcal{A}$ alone, the hypothesis knowing nothing of $\mathcal{Z}$, so the quantifier order of Definition 4.11 is met; $\mathcal{Z} \in \mathbf{Z}'$ was arbitrary. □

Diagrammatically, the proof is the same commuting square as Theorem 4.20's (the figure above its proof), built this time from the *ceding* half of Lemma 4.18 rather than the keeping half: what moves into $\mathcal{Z}$ is not part of the system but the adversary itself, and a relay stands where it stood.



Bottom left is where the theorem starts: π and φ untouched throughout, $\mathcal{A}$ the real-world occupant and $\mathcal{S}_{\mathcal{D}}[\mathcal{A}]$ — $\mathcal{S}_{\mathcal{D}}$ with $\mathcal{A}$ interposed before it — the derived ideal-world one. The top and bottom edges are (R1) and (R2) of the proof: absorbing $\mathcal{A}$ into $\mathcal{Z}$, on whichever side it is

read, is again an *equality* of distributions, exactly as Proposition 4.19's was — but this time it is the ceding half of Lemma 4.18 paying for it, hypotheses (a)–(c) in place of nothing, because what moves is the adversary itself and a relay ($\mathcal{D}$, then $\mathcal{S}_{\mathcal{D}}$) has to stand in its place. The *same* absorbed $\mathcal{Z}^{\mathcal{A}}$ serves both sides, which is why the right edge can compare them at all: it is the theorem's hypothesis, stated only against $\mathcal{D}$. Chasing the square exactly as in Theorem 4.20, the left edge — the theorem's conclusion, now for every refusal-oblivious $\mathcal{A}$ rather than one fixed machine — is forced: the two $\approx_\varepsilon$'s are, by the two equalities, the same quantity.

The three classes move as in Theorem 4.20, and for the same reasons. The adversaries are those the conclusion quantifies over, and the hypothesis had only $\mathcal{D}$; the simulators are computed, one per adversary, by interposing it before the simulator the hypothesis supplies; and the environments are a preimage, an outer $\mathcal{Z}$ counting exactly when what it becomes on absorbing each $\mathcal{A}$ is one the hypothesis covers. Taking $\mathbb{Z}$ to be the largest admissible class makes $\mathbb{Z}'$ the largest one again — less only the environments that silence their own Z-identities, a condition no admissibility imposes and no application wants, excluded because the instruction of Definition 4.26 leaves under a Z-claim, which such an environment would refuse on one side alone — since absorption leaves *par* alone.

Read concretely in the sense of Section 4.5, the two constructions cost what the adversary costs: if $\mathcal{A}$ is $\mathbf{a}$-bounded then $\mathcal{Z}^{\mathcal{A}}$ needs $\mathbf{z} \oplus \mathbf{a}$ where $\mathcal{Z}$ needed $\mathbf{z}$, and $\mathcal{S}_{\mathcal{D}}[\mathcal{A}]$ needs $\mathbf{s} \oplus \mathbf{a}$ where $\mathcal{S}_{\mathcal{D}}$ needed $\mathbf{s}$. The invocation counts add and the per-invocation times take a maximum, both independent of π and φ.

4.5 Concrete Security

Definition 4.11 is stated against arbitrary classes and says nothing about what they contain. Filling them with machines of bounded resources turns each statement of the previous section into a concrete one, and the relations *between* classes into arithmetic on budgets.

Definition 4.30 (Budgets and bounded classes)**.** A *budget* is a pair $\mathbf{a} = (\mathrm{t}, \mathrm{r})$ of naturals. A machine is $\mathbf{a}$-*bounded* if it takes at most t steps in each invocation and answers at most r calls in all; so it

runs for at most t r steps in any execution. A machine meeting both bounds in every execution is counted **a**-bounded whether or not the halting rule is written into it — the bundles built from bounded machines below are bounded in this sense. Write $\mathbb{A}(\mathbf{a})$, $\mathbb{S}(\mathbf{a})$ and $\mathbb{Z}_{\pi,\varphi}(\mathbf{a})$ for the classes of **a**-bounded adversaries, of **a**-bounded simulators, and of **a**-bounded environments lying in $\mathbb{Z}_{\pi,\varphi}$. Budgets are ordered componentwise; machines that share an execution — a bundle of several, or a simulator running an adversary — combine by

$$(t, r) \oplus (t', r') \;:=\; \big(\max(t, t'),\; r + r'\big),$$

one machine running per activation, so the per-invocation time takes the larger bound while the invocation counts add.

The budget is part of the machine, not a promise about it. Bounding the three slot machines is not yet enough to bound an execution — the members of a system are machines too, and a core that recurses on itself, or two members calling one another forever, would carry the token away from every budgeted machine with nothing to reclaim it. Concrete statements therefore read over *budgeted systems*: every member of every system compared carries a budget of its own, in the sense of this definition applied verbatim. An execution then runs finitely many machines, each taking finitely many steps, and exactly one machine steps at any moment, so the execution halts within the total step bound $\sum_i t_i\, r_i$ of its machines and EXEC is defined outright. Halting at a budget is given a value: a machine that halts mid-call resumes each caller suspended on it with $\perp$, and an environment that halts away from its root return is read as returning 0. The classes are monotone in the budget, $\mathbb{A}(\mathbf{a}) \subseteq \mathbb{A}(\mathbf{a}')$ whenever $\mathbf{a} \leqslant \mathbf{a}'$, and likewise for $\mathbb{S}$ and $\mathbb{Z}_{\pi,\varphi}$. That makes Lemma 4.23 a statement about numbers: shrinking the adversary budget, growing the simulator budget, or shrinking the environment budget each preserve emulation.

Definition 4.31 (Concrete UC advantage)**.** Let π and φ be budgeted systems, let **a** and **s** be budgets for the adversary and the simulator, and let $\mathbb{Z} \subseteq \mathbb{Z}_{\pi,\varphi}$ be a class of environments. Set

$$\mathrm{ADV}^{\mathrm{UC}}_{\pi,\varphi}[\mathbf{a}, \mathbf{s}, \mathbb{Z}] \;:=\; \max_{\mathcal{A} \in \mathbb{A}(\mathbf{a})} \; \min_{\mathcal{S} \in \mathbb{S}(\mathbf{s})} \; \max_{\mathcal{Z} \in \mathbb{Z}} \; \mathrm{ADV}^{\mathrm{UC}}_{\pi,\varphi}(\mathcal{Z}, \mathcal{A}, \mathcal{S}),$$

and abbreviate $\mathrm{ADV}^{\mathrm{UC}}_{\pi,\varphi}[\mathbf{a},\mathbf{s},\mathbf{z}]$ for the case $\mathbb{Z} = \mathbb{Z}_{\pi,\varphi}(\mathbf{z})$, the largest class of that budget.

The third argument is a class rather than a budget because the largest class depends on the pair of systems, and the statements below compare advantages taken over *different* pairs. Where a common class is needed the notation must be able to say so.

The three operators are the three quantifiers of Definition 4.11 read as a value, in the same order and for the same reason: the simulator is chosen after the adversary and before the environment. The extrema are attained, though the reason takes a moment, and it is finiteness rather than compactness. An $\mathbf{a}$-bounded machine takes at most $t\,r$ steps in all, so it reads at most $t\,r$ symbols of any input and at most $t\,r$ coins; up to behaviour it is a map from what it read to what it did, and there are finitely many such maps. Its coins weight them with dyadic probabilities of denominator 2^{tr}, so a bounded class induces finitely many behaviour distributions in all. With the members of the systems budgeted as well, $\Pr[\mathrm{EXEC} = 1]$ is a well-defined number at each choice of behaviours, so each advantage of Definition 4.5 takes one of finitely many values, and a maximum or minimum over finitely many values is attained. (Over an arbitrary class $\mathbb{Z}$ the outer max is read as a supremum.) Therefore

$$\pi \text{ UC-emulates } \varphi \text{ within } \varepsilon \text{ against } \big(\mathbb{A}(\mathbf{a}), \mathbb{S}(\mathbf{s}), \mathbb{Z}_{\pi,\varphi}(\mathbf{z})\big)$$

holds exactly when that quantity is at most ε — the attained minimum is what makes the reverse direction exact at the boundary. A concrete security claim is then a single inequality relating three budgets and a probability, with no security parameter and no asymptotics in sight.

CHAPTER 5

Properties

Emulation compares two systems. A *property* asks something of one: that no adversary win a game played against it — which Proposition 5.9 will reduce to the dummy's not winning, the slot absorbing into the environment. Little new machinery is wanted for either. A game is a functionality like any other, the environment is the adversary that plays it, and the plumbing of the previous sections already settles who may call what — so a property is a system, an environment class, and an event, and the fields of Definition 1.1 do most of the work that a game's rules do on paper. What has to be added is small, and is named as it comes: one hygiene condition on the game system, and the three conditions Section 4.4 already asks of anything it moves.

Definition 5.1 (Game, game system)**.** A *game* for $\mathcal{F}$ is a standard functionality $\mathcal{G}\mathfrak{m}$ with

$$\mathcal{G}\mathfrak{m}.\mathit{pid} := (\mathcal{G}\mathfrak{m}.\mathsf{F},\, 0,\, 0), \qquad \mathcal{G}\mathfrak{m}.\mathbf{P} := \mathcal{F}.\mathbf{P} \cup \{\mathsf{Z}\},$$
$$\mathcal{G}\mathfrak{m}.\mathbf{N} := \mathbf{Z}, \qquad \mathcal{G}\mathfrak{m}.\mathbf{U} := \{(\mathcal{F}, \textsc{serves}_{\mathcal{G}\mathfrak{m}})\},$$

carrying beside its own interfaces one or more *finalizations*: interfaces whose names begin with FIN, a prefix reserved for them and borne by no other interface, each served at the root, each returning a bit, and no interface of $\mathcal{G}\mathfrak{m}$ — finalization or otherwise — served once any one of them has been. Here

$$\textsc{serves}_{\mathcal{G}\mathfrak{m}}(\mathcal{G}\mathfrak{m}, \mathcal{F}) \;:\Longleftrightarrow\; \mathcal{F}.\mathbf{P} \supseteq \mathcal{G}\mathfrak{m}.\mathbf{P} \setminus \{\mathsf{Z}\} \;\wedge\; \mathcal{G}\mathfrak{m}.\mathit{pid} \in \mathcal{F}.\mathbf{N}$$

is SERVES with the root exempted. A *game system* is $\gamma := \{\mathcal{G}\mathfrak{m}\} \cup \sigma$ for a system σ containing $\mathcal{F}$ and whatever $\mathcal{F}$ uses and not containing $\mathcal{G}\mathfrak{m}$, with $\textsc{wf}(\gamma)$ and with γ *tight at* $\{\mathcal{G}\mathfrak{m}\}$ in the sense of Remark 6.5.

One game, then, may carry several properties: FINCOR and FINUF over a signature functionality, say, sharing the oracles that record what was signed and differing only in the verdict read off them at the close. The environment chooses which to be judged on by choosing which finalization to call, and that choice is the last thing it does. So a game with several finalizations is not a conjunction of properties but a shell holding several, each with its own advantage below; an environment that maximizes one need not maximize another, and nothing here asks it to.

Game $\mathcal{GM}$ for $\mathcal{F}$: the shell
pid $:= (\mathcal{GM}.\mathrm{F}, 0, 0)$, $\mathbf{P} := \mathcal{F}.\mathbf{P} \cup \{\mathrm{Z}\}$, $\mathbf{N} := \mathbf{Z}$, $\mathbf{U} := \{(\mathcal{F}, \text{SERVES}_{\mathcal{GM}})\}$, *par* $:= \bot$

INITIALIZE():
1: DONE $\leftarrow$ FALSE
2: … the game's own variables

id.OP(in) from id′
3: **require** ¬DONE *// closed once finalized*
4: … the game's own code, which may call $\mathcal{F}$ at party id.P
5: **return** out

id.LEAK() from id′
6: **return** $\bot$ *// a game keeps no party's secrets*

Finalizations of $\mathcal{GM}$, one per property; the shell for $\mathcal{F}$ is shared
each named FIN …, each served at the root, each closing the game

id.FIN() from id′ (the shape of every finalization)
7: **require** ¬DONE *// one finalization only*
8: **require** id.P $=$ Z *// at the root, which stays honest*
9: DONE $\leftarrow$ TRUE
10: $w \leftarrow$ this property's verdict, read off the shell's state
11: **return** w

id.FINCOR(), id.FINUF(), … from id′
12: … the same three lines, then each property's own verdict

Each field is forced, and it is worth saying by what.

The process id. There is one game per experiment, so the session and instance carry no information and we fix both at 0. Session 0 is not a convention but a requirement: the environment reaches $\mathcal{GM}$ by claiming its own Z-identity, which line 2 admits because $\mathbf{Z} \subseteq \mathcal{GM}.\mathbf{N}$, and line 3

then asks $\mathcal{G}\mathfrak{m}.\mathsf{s} = \mathcal{Z}.\mathsf{s} = 0$ — $\mathcal{G}\mathfrak{m}$ not being global. Remark 7.17 is that observation in general. By Definition 1.3 a local $\mathcal{F}$ shares the caller's session, so the whole experiment sits in session 0, which is where the machines the execution supplies live too.

The served parties. A core claims only its own identity (line 5) and GUARD asks $\mathtt{id}'.\mathsf{P} = \mathtt{id}.\mathsf{P}$, so $\mathcal{G}\mathfrak{m}$ at party P reaches $\mathcal{F}$ at party P and at no other: the game exercises $\mathcal{F}$ one party at a time, and the environment chooses the party by choosing which identity of $\mathcal{G}\mathfrak{m}$ to address, its root being free to act for any by line 3. Hence $\mathcal{F}.\mathbf{P}$. The root is added for the finalizations, and the next paragraph says why.

The admitted callers. **Z** and nothing else. Not **Std**, which would make $\mathcal{G}\mathfrak{m}$ global and let any standard functionality finalize the game; not **A**, which would hand it to the adversary. With **Z** alone, line 2 refuses every caller but the environment.

The name. Arbitrary, and Corollary 7.16 says so: renaming the game and $\mathcal{F}.\mathbf{N}$ together changes no probability, so a property proved against a game of one name holds against it under every other. This matters because line 5 lets the game claim only its own identity, so $\mathcal{F}.\mathbf{N}$ has to name the game while the $\mathcal{F}$ that ships inside a protocol names that protocol; the corollary is what makes the two the same property. What is not arbitrary is the caller's *code*, which no renaming touches.

The dependence. $\text{SERVES}_{\mathcal{G}\mathfrak{m}}$ exempts the root because the root is the game's own bookkeeping identity: $\mathcal{F}$ need not serve a party the game never calls it at. A game whose verdict must *query* $\mathcal{F}$ — verifying a claimed forgery, say — has two options, and they are the usual ones: record during the oracle calls whatever the verdict needs, or ask $\mathcal{F}$ to serve the root as well, when $\text{SERVES}_{\mathcal{G}\mathfrak{m}}$ becomes SERVES.

Tightness. The fields above do not suffice on their own, and the gap is Remark 6.5's. An admissible environment is silenced on $\langle\gamma\rangle$, so it may *claim* no identity of a name γ uses — but Definition 4.10 forbids it only from *hosting* at identities γ occupies, and a hosted functionality is a full interface claiming its own identity through its own silencer (Definition 4.13). So an environment may host a machine at an unoccupied process id of a used name: at $(\mathcal{G}\mathfrak{m}.\mathsf{F}, 0, 1)$, say, an invented second instance of the game's own name. A member of σ that admits its caller by *name* — the local pattern of Chapter 1 — admits that machine, and its calls pass every line of GUARD: the name lies in **N**, the parties agree,

the sessions agree, and MEDIATE does not fire at an honest party. The environment would then hold an oracle on $\mathcal{F}$ that the game records nothing of, and $\Pr[\text{WIN}_{\text{FIN}}] = 1$ for any game whose verdict is that the adversary produced something it was not given.

Tightness at $\{\mathcal{G}\mathfrak{m}\}$ refuses that, and refuses it completely. Every member but the game admits whole process ids the system occupies, so an invented instance fails line 2; and Remark 6.5's party condition makes every identity a pinned caller could speak from occupied, where Definition 4.10 forbids hosting. The one route tightness leaves open — to a member of I itself, which here is the game — is closed by $\mathcal{G}\mathfrak{m}.\mathbf{N} = \mathbf{Z}$: a hosted functionality is standard and claims a standard process id, which $\mathbf{Z}$ excludes, while the environment's own Z-identities are not hosted but its own. The game is thus the one member that may admit a bare class of process ids, and it is safe in doing so exactly because Z is no standard name.

Remark 5.2 (Why a finalization is served at the root)**.** Party ids in $\{0,1\}^*$ can be corrupted, and line 4 of the full interface hands a corrupt party's call to the adversary *before* the core runs. A finalization served at such a party would therefore be answered by the adversary whenever that party is corrupt, and the answer to "have I won?" would be whatever the adversary says — Remark 3.1's problem exactly, and no check inside the core can repair it, mediation happening in the wrapper above. The root is the fix, and the framework already supplies it: FULLCORRUPT asks $\mathtt{id}.\mathsf{P} \in \{0,1\}^*$, so Z never enters $\mathbf{C}$, so MEDIATE at $(\mathcal{G}\mathfrak{m}.\mathit{pid}, \mathsf{Z})$ can never fire and the bit line 10 computes is always the game's own. The reserved party ids were introduced to keep the three supplied machines honest; they keep the game honest at no extra cost. Line 8 then confines every finalization to that identity, so no other is a place to win.

Remark 5.3 (One finalization, once, and last)**.** That exactly one finalization is answered, and that it is the last call the game answers, is enforced rather than assumed — and the one flag does both jobs. Line 9 sets a flag the wrapper cannot bypass, line 7 refuses every later finalization, its own name or another's, and line 3 refuses every other interface thereafter; a refused **require** returns REJ by

Section 3.6, so the environment learns that the game is closed and nothing else — at an honest party, at least. At a corrupt one line 4 intercepts before the core is entered, so the caller is answered by the adversary rather than refused; that changes nothing, the slot being unable to write the game's state and the winning event being fixed already. Nothing stops the environment calling a global functionality afterwards either, which is right — the game is over, and what it does then cannot change what it won. The order of the lines matters for the reason $\mathcal{F}_{\mathsf{Sig}}$ re-tests its tables: line 10 may place calls, and Section 3.6 lets a suspended instance be re-entered, so the flag is set before the verdict is computed rather than after. Were it the other way about, two nested finalizations could each find the game open.

That one flag governs them all is what makes several finalizations safe to hang off one shell. Were each to keep a flag of its own, an environment could finalize twice and be judged on two properties in one run — reading, say, a correctness verdict off a state its unforgeability verdict had already been computed from, with the oracles between them. Sharing DONE is what confines a run to one property, and it is why the finalizations sit in a box of their own with the shell's flag rather than each carrying its own bookkeeping.

Remark 5.4 (What corruption buys, and what it costs)**.** At a party $\mathsf{P} \in \{0,1\}^*$ the game's own interfaces are wrapped like any other functionality's, so once $\mathsf{P} \in \mathbf{C}$ line 4 intercepts a call to $\mathcal{G}\mathfrak{m}$ at P: the core does not run, the game records nothing, and the caller is answered by the adversary. That gains an environment nothing — MEDIATE replaces an answer rather than running a core, so the game's state is never written by the adversary — and it costs it the oracle at that party. Since corruption is indexed by party alone (Section 3.1), corrupting a party of $\mathcal{F}$ closes the game's oracle at that party too. A game that wants an oracle at a corrupt party must therefore serve it somewhere the corruption does not reach: at the root, or at a party of its own outside $\mathcal{F}.\mathbf{P}$, which it may do whenever that oracle needs no call on $\mathcal{F}$.

What keeps the adversary out of the game's cores is not either of the two lines one would reach for. Line 3 *admits* $\mathcal{A}$ at a corrupt party,

refusing it only at honest ones; and MEDIATE tests $\mathrm{id}'.\mathsf{F} \neq \mathsf{A}$, so it does not intercept a call the adversary itself places. A call from the slot at a corrupt party would therefore run $\mathcal{G}\mathfrak{m}$'s core, exactly as Section 3.5 says of a corrupt party addressed by $\mathcal{A}$. What refuses it is $\mathcal{G}\mathfrak{m}.\mathbf{N} = \mathbf{Z}$ at line 2, and nothing else does. The narrowness of that field is not tidiness but the only thing between the adversary and the game's own code.

And $\mathcal{G}\mathfrak{m}$'s LEAK returns $\perp$ because leakage does not go through GUARD (Section 3.2): the adversary can read it at a corrupt party whatever $\mathcal{G}\mathfrak{m}.\mathbf{N}$ says, so a game must keep nothing there it is not willing to hand over.

Remark 5.5 (Several instances of $\mathcal{F}$)**.** A game may address as many instances of $\mathcal{F}$ as it likes, and the two ways it can differ. Instances in the game's own session need nothing extra, line 3 comparing sessions and ignoring the instance number, so $(\mathcal{F}.\mathsf{F}, 0, \mathfrak{i})$ is reachable for every $\mathfrak{i}$. Instances in other sessions need $\mathrm{GLOBAL}(\mathcal{F})$, which is what waives that line. Either way the claim is $\mathcal{G}\mathfrak{m}$'s own identity, line 5 silencing its cores on everything else, so every instance sees one and the same caller and access control is decided once. And the instances really are copies: by Chapter 7 they run the same code and their fields differ in the process id alone, while Section 3.6 gives each its own state and its own coins. So a multi-instance property — n keys, n sessions — is written by having the game count them, with no extra convention about what a second instance means.

What counting them does not do is make transfer cheaper. Proposition 5.10 replaces one subsystem, so a game over n instances every one of which is to be replaced costs n applications and $n\varepsilon$, by Theorem 4.24 along a chain of hybrids — exactly as Remark 7.18 records on the relocation side. One application transfers the property for the instances φ contains and says nothing of the others.

Remark 5.6 (A game is not session-uniform, and need not be)**.** $\mathcal{G}\mathfrak{m}.\mathbf{N} = \mathbf{Z}$ meets $\mathbf{Z}$ while $\mathcal{G}\mathfrak{m}$ is not global, so Definition 7.2 fails of $\mathcal{G}\mathfrak{m}$ and the results of Chapter 7 do not apply to a game system. That is as it should be. A game is pinned to session 0 by the paragraph

above, there is one of it per experiment, and relocating it is not something anyone wants; what the chapter applies to is the $\mathcal{F}$ under test, whose instances Remark 5.5 counts.

Which environments may play is settled by the class Chapter 4 already defines, taken at a single system.

Definition 5.7 (Property environments)**.** The environments *admissible for* a game system γ are $\mathbb{Z}_{\gamma,\gamma}$ of Definition 4.10: those with $\mathcal{Z}.par \supseteq \langle\gamma\rangle$ that host at no identity of $\mathrm{IDs}(\gamma)$. Write $\mathbb{Z}^{\circ}_{\gamma}$ for those among them that silence no Z-identity, the ones the absorption of Proposition 5.9 can move.

The two clauses say what a game needs them to say. Silenced on the name closure, an environment may claim no identity of a name γ uses, so it cannot speak as $\mathcal{F}$, as one of $\mathcal{F}$'s subroutines, or as a further instance of any of them; what it may claim is its own Z-identities, which reach $\mathcal{G}\mathfrak{m}$, and fresh names, which reach only what admits all of **Std**. So the environment plays the game through $\mathcal{G}\mathfrak{m}$'s interfaces and touches $\mathcal{F}$ directly only when $\mathcal{F}$ is global — and then it touches it as anyone may, which is what being global means and what Definition 4.10 already grants in the emulation setting. A property of a global $\mathcal{F}$ is therefore a property against an adversary that shares it, and the game must be written knowing that. The hosting clause is the occupancy hygiene of Definition 3.3, and it also stops the environment from installing a second game.

Definition 5.8 (Win, property advantage)**.** Let $\gamma = \{\mathcal{G}\mathfrak{m}\} \cup \sigma$ be a game system, let FIN be one of $\mathcal{G}\mathfrak{m}$'s finalizations, let $\mathcal{Z}$ be admissible for γ (Definition 5.7), and let $\mathcal{X}$ occupy the adversary slot. Write $\mathrm{WIN}_{\mathrm{FIN}}$ for the event that the interface $(\mathcal{G}\mathfrak{m}.pid, \mathsf{Z}).\textsc{FullFin}$ returns 1 in $\textsc{Exec}(\gamma, \mathcal{Z}, \mathcal{X}, \mathcal{C})$. The *advantage* of $\mathcal{Z}$ in the property FIN of σ is one of

$$\mathrm{ADV}^{\mathrm{FIN}}_{\mathcal{G}\mathfrak{m},\sigma,\mathcal{Z},\mathcal{X}} := 2\Pr[\mathrm{WIN}_{\mathrm{FIN}}] - 1,$$

$$\mathrm{ADV}^{\mathrm{FIN}}_{\mathcal{G}\mathfrak{m},\sigma,\mathcal{Z},\mathcal{X}} := \Pr[\mathrm{WIN}_{\mathrm{FIN}}],$$

the *decision* reading and the *search* reading, the probability being over the coins of every machine in the execution. Which of the two

is meant is fixed for each finalization where the game is given, and not by this definition; the paragraph on calibration below says how to choose. We drop $\mathcal{X}$ from the subscript when it is $\mathcal{D}$, and write $\mathcal{F}$ for σ when $\sigma = \{\mathcal{F}\}$. For $\varepsilon \geqslant 0$, a class $\mathbb{Z} \subseteq \mathbb{Z}_{\gamma,\gamma}$ of environments and a class $\mathbb{X}$ of slot occupants, we say σ has the property FIN *within* ε *against* $(\mathbb{Z}, \mathbb{X})$ if $\mathrm{ADV}^{\mathrm{FIN}}_{\mathcal{G}\mathfrak{m},\sigma,\mathcal{Z},\mathcal{X}} \leqslant \varepsilon$ for every $\mathcal{Z} \in \mathbb{Z}$ and every $\mathcal{X} \in \mathbb{X}$, and *within* ε *against* $\mathbb{Z}$ when $\mathbb{X} = \{\mathcal{D}\}$.

Since one flag closes the game (Remark 5.3), at most one $\mathrm{WIN}_{\mathrm{FIN}}$ occurs in any run: the events of distinct finalizations are disjoint, and a game with several of them is several properties measured in several executions rather than several numbers read off one.

Under the decision reading the advantage is signed, where that of Definition 4.5 is not, and the asymmetry is the point: an environment cannot flip $\mathrm{WIN}_{\mathrm{FIN}}$ as it can flip a guess, the event being the game's verdict rather than its own, so a game it cannot win scores -1 and there is nothing to take an absolute value of.

$\mathrm{WIN}_{\mathrm{FIN}}$ is an event of the execution rather than its output, and that is deliberate: an execution's output is whatever the environment returns (Chapter 4), and a property is not a guess to be reported but a thing that happened. Nothing is lost by the distinction, because Remark 5.2 makes the two agree where it matters. The bit line 10 computes is the bit the finalization hands the environment, unmediated, so we may name the environment that reports it: for a fixed FIN, write $\mathcal{Z}^\star$ for $\mathcal{Z}$ with its return at the root replaced by "return 1 if any call I placed on $(\mathcal{G}\mathfrak{m}.\mathit{pid}, \mathsf{Z})$.FULLFIN was answered with 1, and 0 otherwise". Every part of that earns its place. The *identity* is named because a finalization addressed at another party fails line 8 where that party is honest and is answered by the *adversary* where it is corrupt, so the value coming back need not be the game's; at the root neither happens, by Remark 5.2. The *operation* is named because $\mathsf{Z} \in \mathcal{G}\mathfrak{m}.\mathbf{P}$, so line 1 serves every interface of $\mathcal{G}\mathfrak{m}$ at the root, the game's own oracles included — and because with several finalizations the reporting form must say which property it is reporting on, one $\mathcal{Z}^\star$ per FIN. The test is *against* 1 rather than a reading of the answer as a bit, because a later finalization comes back REJ by Remark 5.3, and REJ is no bit (Section 3.6). And it is *any* call rather than the first, because the first one placed may be refused — an environment claiming (Z, P) for $\mathsf{P} \neq \mathsf{Z}$ fails line 2's

party test, leaving DONE unset — while a later one is served; nothing is weakened by quantifying, since at most one call is ever answered with 1, the game serving one finalization and refusing the rest. So EXEC returns 1 exactly when WIN_{FIN} occurs, and $\mathcal{Z}^\star$ can compute it, every call on that interface being one it placed itself: $\mathcal{G}\mathfrak{m}.\mathbf{N} = \mathbf{Z}$ admits no other caller, and a functionality the environment hosts claims a standard id, which $\mathbf{Z}$ excludes. Then

$$\Pr\big[\text{EXEC}(\gamma, \mathcal{Z}^\star, \mathcal{X}, \mathcal{C}) = 1\big] \;=\; \Pr\big[\text{WIN}_{\text{FIN}} \text{ in } \text{EXEC}(\gamma, \mathcal{Z}, \mathcal{X}, \mathcal{C})\big],$$

the two machines placing the same calls and differing only in the value they halt with. That also puts $\mathcal{Z}^\star$ in every class $\mathcal{Z}$ is in whose membership turns on *par* and hosting alone — $\mathbb{Z}_{\gamma,\gamma}$ among them, and the budgeted classes of Definition 4.30 up to the one further step of reading a bit back. Proposition 5.10 is where all of this is used.

Why the dummy. The default slot occupant is the dummy of Section 4.4, and the reason is that a game has one adversary and it is the environment. The dummy relays in both directions and keeps no strategy, so an environment driving it holds everything the slot can do; and where $\mathcal{F}$'s own cores reach the adversary — the $\mathcal{A}(\cdot)$ of Section 3.4, which is how an ideal functionality leaves a value unpinned — it is then the environment that answers, which is exactly the freedom a game means to give it. Dropping $\mathcal{X}$ from the subscript for the dummy's case is thus no loss of generality but a choice of who plays; the longer form is kept because Proposition 5.10 needs a simulator in that slot. That the abbreviation costs nothing is not a matter of taste, and the reason is the construction of Section 4.4: the slot absorbs into the environment.

Proposition 5.9 (The slot absorbs into the environment). *Let $\gamma = \{\mathcal{G}\mathfrak{m}\} \cup \sigma$ be a game system whose cores place no responsive calls (Definition 9.2), let $\mathcal{X}$ be a refusal-oblivious occupant of the adversary slot (Definition 4.28), and let $\mathcal{Z} \in \mathbb{Z}^\circ_\gamma$. Then $\mathcal{Z}^{\mathcal{X}}$ of Definition 4.26 again lies in $\mathbb{Z}^\circ_\gamma$, and for every finalization* FIN *of $\mathcal{G}\mathfrak{m}$*

$$\text{ADV}^{\text{FIN}}_{\mathcal{G}\mathfrak{m},\sigma,\mathcal{Z},\mathcal{X}} \;=\; \text{ADV}^{\text{FIN}}_{\mathcal{G}\mathfrak{m},\sigma,\mathcal{Z}^{\mathcal{X}}}.$$

So if σ has the property FIN *within ε' against $\mathbb{Z}^\circ_\gamma$, it has it within ε' against $(\mathbb{Z}^\circ_\gamma, \mathbb{X})$ for every class $\mathbb{X}$ of refusal-oblivious occupants. Read concretely, an $\mathbf{z}$-bounded environment and an $\mathbf{x}$-bounded occupant give an absorbed environment that is $\mathbf{z} \oplus \mathbf{x}$-bounded.*

Proof. Lemma 4.18 at $\mathsf{H} = \{\mathcal{X}\}$ with $\mathsf{H}_{\text{KEEP}} = \varnothing$ gives

$$\textsc{Exec}(\gamma, \mathcal{Z}, \mathcal{X}, \mathcal{C}) \;\equiv\; \textsc{Exec}(\gamma, \mathcal{Z}^{\mathcal{X}}, \mathcal{D}, \mathcal{C}),$$

its three hypotheses being the three assumed here, one for one. Now WIN_{FIN} is the event that the interface FULLFIN at $(\mathcal{G}\mathfrak{m}.pid, \mathsf{Z})$ returns 1, and absorption touches the adversary slot alone — $\mathcal{G}\mathfrak{m}$ occupies that identity on both sides — so the event has one probability for each FIN alike, and Definition 5.8 gives one number under either reading. Admissibility is Section 4.4's observation that $\mathcal{Z}^{\mathcal{X}}$ occupies what an environment occupies and carries $\mathcal{Z}$'s *par*, so it is admissible exactly when $\mathcal{Z}$ is; the budget is the $\mathbf{z} \oplus \mathbf{a}$ of Chapter 8. The final sentence is the display read at each $\mathcal{X}$ in turn. □

The hypotheses are Theorem 4.29's and no more: a game system placing responsive calls is outside this, by Remark 9.4, and so is an environment that silences its own Z-identities, the instruction of Definition 4.26 leaving under a Z-claim. Neither is a condition on what the occupant *does*, which is the point — a game need not care whether the machine in the slot plays fairly, only that it can be moved.

One scope condition comes with all of this, and it is the same one throughout. The proposition above rests on the regrouping of Theorem 4.29, which holds only for systems whose cores place no responsive calls, Remark 9.4 showing where the relay fails otherwise; the same condition governs Theorem 4.29 itself, which is what supplies the emulation hypothesis Proposition 5.10 wants from a wider adversary class, and Corollary 5.11 inherits it on both sides at once. A game system whose $\mathcal{F}$ reaches the slot through $\mathcal{A}^!$ therefore sits outside all three and must have its dummy-only hypothesis established directly. What a responsive theory of games would look like we leave where Remark 9.4 leaves it.

The calibration. Which of Definition 5.8's two readings a finalization takes is not a matter of taste, and the rule is this. The factor $2\Pr[\text{WIN}_{\text{FIN}}] - 1$ reads the property as a *decision*: an environment that guesses wins with probability $\frac{1}{2}$ and scores 0, and one that always wins scores 1. That is right where guessing is a strategy — where the verdict is a bit the environment could have flipped a coin for. It is wrong for a *search* property — unforgeability, say, where winning is meant to be hard outright — because there the same quantity reads $\Pr[\text{WIN}_{\text{FIN}}] = (1 + \text{ADV}^{\text{FIN}}_{\mathcal{G}\mathfrak{m},\sigma,\mathcal{Z}})/2$, so a bound ε on the advantage is

a bound $(1+\varepsilon)/2$ on the winning probability, which for small ε is the wrong scale: an ε of 0 would permit a forger succeeding half the time. Search properties therefore take $\Pr[\mathrm{WIN}_{\mathrm{FIN}}]$ itself. Declaring the reading per finalization rather than fixing one globally is what lets a single shell carry both kinds at once. Every property in this paper is of the second kind — winning each of the games below is meant to be impossible outright, not merely no better than a guess — so all of them take the search reading; the decision reading is there for properties where guessing is the strategy, an indistinguishability written as a game among them.

The choice is not free, and Proposition 5.10 is where it is paid for. Its two displays are the same bound at the two scales — 2ε for the decision reading, ε for the search reading — so each finalization inherits transfer at the scale its own reading fixes, and a game mixing the two inherits at both.

Finally, the reason to define properties this way rather than beside the framework: emulation carries them.

Proposition 5.10 (Properties transfer along emulation)**.** *Let $\gamma = \{\mathcal{G}\mathfrak{m}\} \cup \sigma$ be a game system, let $\varphi \subseteq \sigma$, let the replacement of φ by π in γ be well-formed in the sense of Definition 4.2, and let $\gamma[\pi/\varphi]$ be a game system as well. Suppose*

$$\pi \text{ UC-emulates } \varphi \text{ within } \varepsilon \text{ against } (\{\mathcal{D}\}, \mathbb{S}, \mathbb{Z}).$$

Then there is $\mathcal{S} \in \mathbb{S}$ such that for every finalization FIN *of $\mathcal{G}\mathfrak{m}$ and every $\mathcal{Z} \in \mathbb{Z}_{\gamma[\pi/\varphi],\gamma}$ whose reporting form has $\mathcal{Z}^{\star\,\gamma\setminus\varphi} \in \mathbb{Z}$ — a condition Proposition 4.16 makes vacuous whenever $\mathbb{Z}$ is the largest admissible class, though not when it is the largest class at a* budget*, absorption costing $\mathbf{c}_{\gamma\setminus\varphi}$ and the reporting form one step more —*

$$\mathrm{ADV}^{\mathrm{FIN}}_{\mathcal{G}\mathfrak{m},\sigma[\pi/\varphi],\mathcal{Z}} \leqslant \mathrm{ADV}^{\mathrm{FIN}}_{\mathcal{G}\mathfrak{m},\sigma,\mathcal{Z},\mathcal{S}} + 2\varepsilon \qquad \textit{(decision reading)},$$

and, writing $\mathrm{WIN}_{\mathrm{FIN}}$ and $\mathrm{WIN}'_{\mathrm{FIN}}$ for the winning events of the two executions the display above compares — the real one and the ideal one —

$$\Pr[\,\mathrm{WIN}_{\mathrm{FIN}}\,] \leqslant \Pr[\,\mathrm{WIN}'_{\mathrm{FIN}}\,] + \varepsilon \qquad \textit{(search reading)}.$$

Proof. Three moves and one bound. First the statement is checked to be about the systems it names, which is where the hypothesis on $\gamma[\pi/\varphi]$

is spent: emulation supplies neither the game's party set at the new occupant nor its process id in that occupant's $\mathbf{N}$, and tightness is part of Definition 5.1 besides. Then the reporting form of Definition 5.8 turns the winning event into an output bit, rewriting what $\mathcal{Z}$ returns at the root and changing no run. Then absorption moves the game itself inside the environment, so that an environment *playing* the game becomes one *distinguishing* the two systems. What is left is Theorem 4.20 applied at that absorbed environment. The search reading is the bound it gives; the decision reading is that bound doubled, and nothing in the argument reads FIN, so it holds of each finalization separately. First, the two advantages are defined and are about the systems the statement names. Since $\mathcal{G}\mathfrak{m} \notin \sigma$ we have $\mathcal{G}\mathfrak{m} \notin \varphi$, so $\gamma[\pi/\varphi] = \{\mathcal{G}\mathfrak{m}\} \cup \sigma[\pi/\varphi]$; and $\mathbb{Z}_{\gamma[\pi/\varphi],\gamma}$ lies in both $\mathbb{Z}_{\gamma,\gamma}$ and $\mathbb{Z}_{\gamma[\pi/\varphi],\gamma[\pi/\varphi]}$, each clause of Definition 4.10 weakening when a union of two closures shrinks to one. That the left-hand side is a property advantage at all is where the hypothesis on $\gamma[\pi/\varphi]$ is spent, and it is not free: emulation supplies neither $\mathcal{G}\mathfrak{m}.\mathbf{P} = \mathcal{F}'.\mathbf{P} \cup \{\mathsf{Z}\}$ at the new occupant $\mathcal{F}'$ nor $\mathcal{G}\mathfrak{m}.\mathit{pid} \in \mathcal{F}'.\mathbf{N}$, the second being hypothesis (ii) of Theorem 4.4 and no part of Theorem 4.20. Nor is that all it buys: tightness is part of Definition 5.1, so the hypothesis asks in addition that π's members pin their callers by process id where φ's did. Emulation supplies that least of all, comparing behaviour at the boundary and saying nothing of the declared fields behind it.

Let $\mathcal{S} \in \mathbb{S}$ be the simulator the hypothesis supplies for $\mathcal{D}$; it depends on $\mathcal{D}$ alone, so one $\mathcal{S}$ serves every $\mathcal{Z}$. Fix such a $\mathcal{Z}$, fix FIN, and take the reporting form $\mathcal{Z}^\star$ for that FIN from the paragraph after Definition 5.8, which gives

$$\Pr\big[\textsc{Exec}(\delta, \mathcal{Z}^\star, \mathcal{X}, \mathcal{C}) = 1\big] \;=\; \Pr\big[\textsc{Win}_{\textsc{Fin}} \text{ in } \textsc{Exec}(\delta, \mathcal{Z}, \mathcal{X}, \mathcal{C})\big]$$

for either game system δ and either occupant $\mathcal{X}$; the hypothesis places $\mathcal{Z}^\star$ in the class $\mathbb{Z}'$ that Theorem 4.20 attaches to this replacement, and $\mathcal{Z}^\star \in \mathbb{Z}_{\gamma[\pi/\varphi],\gamma}$ because $\mathcal{Z}$ is and the two agree on *par* and on what they host. Theorem 4.20 applied to the replacement of φ by π in γ, at the adversary $\mathcal{D}$ and the environment $\mathcal{Z}^\star$, bounds the difference of the two left-hand probabilities — with $\delta = \gamma[\pi/\varphi]$ and $\mathcal{X} = \mathcal{D}$ against $\delta = \gamma$ and $\mathcal{X} = \mathcal{S}$ — by ε. Hence $\Pr[\textsc{Win}_{\textsc{Fin}}]$ differs by at most ε between the two, which is the second display; the advantage under the decision reading, being $2\Pr[\textsc{Win}_{\textsc{Fin}}] - 1$, then differs by at most 2ε, which is the first. The order matters for use rather than for proof: the probability

bound is what the argument gives, and the decision-reading bound is it doubled. Under the search reading the advantage *is* the probability, so the second display is already the transfer statement and the factor of two never arises. Nothing in the argument reads FIN, so it holds of each finalization separately. □

Two things about that proof. The reduction is the textbook one, and absorption is what makes it so: the absorbed environment $\mathcal{Z}^{\star\,\gamma\setminus\varphi}$ of Definition 4.15 hosts the game itself along with the rest of $\gamma \setminus \varphi$, so "an environment playing the game against π" becomes "an environment distinguishing π from φ" by moving the game inside it, which is what Theorem 4.20 was proved about. And the property on the right is stated against $\mathcal{S}$ in the slot rather than against the dummy, which is why Definition 5.8 keeps the occupant as an argument — but it is not an obligation anyone need meet in that form, because the slot absorbs into the environment as well.

The reduction is worth drawing, because it is one move and the move is easy to read past. It is the commuting square of Figure 1 again, in the same conventions and with the dotted box again marking the pair being exchanged — but this time it is the *game* that moves:

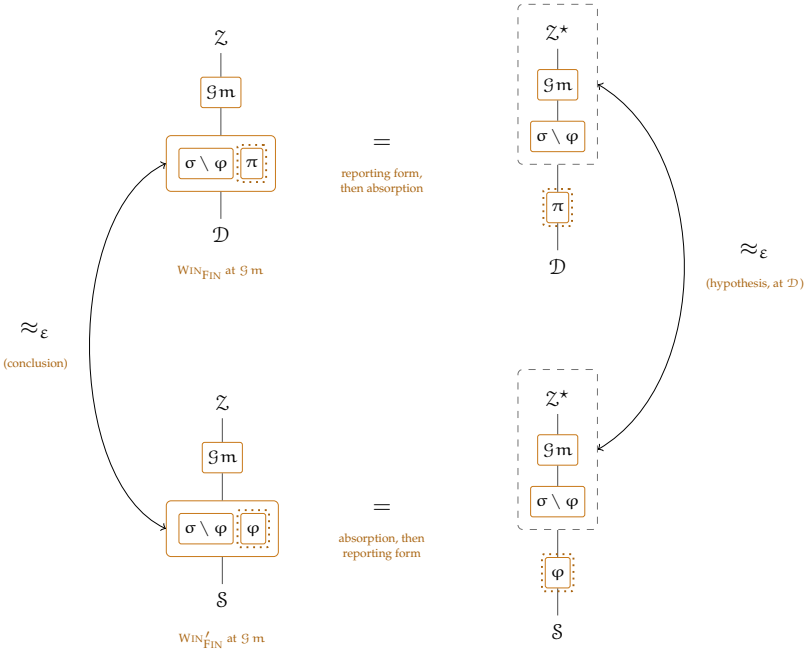


Figure 2: Property transfer, as the same square with the game moving instead of the protocol. In the left column $\mathcal{G}\mathfrak{m}$ sits inside the system, where a game belongs; in the right it sits inside the dashed box, part of the machine doing the distinguishing. Crossing that boundary is the whole reduction. (Proposition 5.10.)

Read it by watching $\mathcal{G}\mathfrak{m}$. In the left column it sits inside the system, where a game belongs: the environment plays it, and WIN_{FIN} is the verdict it returns. In the right column it sits inside the dashed box, part of the machine doing the distinguishing. Crossing that boundary is the whole reduction — an environment *playing the game* against π becomes an environment *telling π from φ* — and once the game is across, what is left is a bare emulation question, which is what Theorem 4.20 was proved about.

The two horizontal edges are free. The reporting form only rewrites what $\mathcal{Z}$ returns at the root, leaving every call it places untouched, so it changes no run; and absorption is Lemma 4.18, an identity of executions rather than a bound. So the right edge is the only thing paid for, and it is the hypothesis applied at $\mathcal{D}$ — which is why the bottom row carries $\mathcal{S}$ in the slot and not the dummy, and so why Definition 5.8 keeps the occupant as an argument at all. The left edge is then forced, exactly as

in Figure 1: chasing the square makes the two $\approx_\varepsilon$'s the same quantity rather than two bounds that happen to share an ε.

One reading of that quantity is the proposition's, the other is its double. Under the search reading the property advantage *is* $\Pr[\mathrm{WIN}_{\mathrm{FIN}}]$, so the left edge is already the transfer statement and no factor of two appears; under the decision reading the advantage is $2\Pr[\mathrm{WIN}_{\mathrm{FIN}}] - 1$, which doubles it. Every per-functionality property proof in the paper is an instance of this one square, which is why it earns a diagram where the instances do not.

Corollary 5.11 (Transfer, against every adversary)**.** *In the setting of Proposition 5.10, suppose in addition that the cores of γ and of $\gamma[\pi/\varphi]$ place no responsive calls (Definition 9.2), that the simulator $\mathcal{S}$ that proposition supplies and the occupant $\mathcal{A}$ below are refusal-oblivious (Definition 4.28), and that the environments named lie in $\mathbb{Z}^\circ_\gamma$. Fix a finalization* FIN *of $\mathcal{G}\mathfrak{m}$. If σ has the property* FIN *within ε' against $\mathbb{Z}_{\gamma,\gamma}$ in the sense of Definition 5.8, then*

$$\mathrm{ADV}^{\mathrm{FIN}}_{\mathcal{G}\mathfrak{m},\sigma[\pi/\varphi],\mathcal{Z},\mathcal{A}} \;\leqslant\; \varepsilon' + 2\varepsilon$$

for every occupant $\mathcal{A}$ of the adversary slot and every $\mathcal{Z}$ the proposition admits with $\mathcal{Z}^{\mathcal{A}}$ among them. If in place of the hypothesis on advantages the ideal side satisfies $\Pr[\mathrm{WIN}'_{\mathrm{FIN}}] \leqslant p$ under the same quantification, then $\Pr[\mathrm{WIN}_{\mathrm{FIN}}] \leqslant p + \varepsilon$, with $\mathrm{WIN}_{\mathrm{FIN}}$ and $\mathrm{WIN}'_{\mathrm{FIN}}$ naming the real and ideal events as in Proposition 5.10 — which is the form a search property inherits in, the first display being the decision reading's.

Proof. Three steps, of which the outer two are absorption. By Proposition 5.9 at the real game system, $\mathrm{ADV}^{\mathrm{FIN}}_{\mathcal{G}\mathfrak{m},\sigma[\pi/\varphi],\mathcal{Z},\mathcal{A}} = \mathrm{ADV}^{\mathrm{FIN}}_{\mathcal{G}\mathfrak{m},\sigma[\pi/\varphi],\mathcal{Z}^{\mathcal{A}}}$, the right-hand side being the dummy's case. Proposition 5.10 at $\mathcal{Z}^{\mathcal{A}}$ bounds it by $\mathrm{ADV}^{\mathrm{FIN}}_{\mathcal{G}\mathfrak{m},\sigma,\mathcal{Z}^{\mathcal{A}},\mathcal{S}} + 2\varepsilon$. And Proposition 5.9 again, now at the ideal game system, rewrites the first term as $\mathrm{ADV}^{\mathrm{FIN}}_{\mathcal{G}\mathfrak{m},\sigma,(\mathcal{Z}^{\mathcal{A}})^{\mathcal{S}}}$, which the hypothesis on σ bounds by ε', the doubly absorbed machine being an admissible environment by two applications of the same proposition. The probability form is the same three steps read on the second display of Proposition 5.10 rather than the first, absorption changing no probability at all. □

Two things this buys. The hypothesis on the ideal side is now the property against the *dummy* alone — one statement about σ, quantified over environments and nothing else — where Proposition 5.10 on its own asked for the property against the particular simulator the emulation supplied. Nothing need be assumed about how that simulator corrupts or leaks — only that it does not branch on a framework-internal refusal, which Section 4.4 asks of every machine it moves: whatever else it does, absorbing it into the environment turns it into part of a machine the ideal-side hypothesis already covers. And the conclusion is the property of the real system against *every* adversary, not only against the dummy, by the same move applied on the other side. Read concretely, the two absorptions are the whole price: for an **a**-bounded $\mathcal{A}$ and an **s**-bounded $\mathcal{S}$ the innermost statement is reached at environment budget $\mathbf{z} \oplus \mathbf{a} \oplus \mathbf{s}$, one hosting cost per machine moved, and Chapter 8 charges nothing else.

Remark 5.12 (What the occupant of the slot does on the ideal side)**.** Nothing here is needed for the proof above, which absorbs whatever the slot does into the ε Theorem 4.20 supplies. What is at stake is instead the cost of the *hypothesis* on the right: the two executions the proposition compares differ in the slot as well as in the system. Corruption divides in two, and only half of it is at issue. The half that is not is what the environment does for itself: line 3 admits $\mathrm{id}'.\mathsf{F} \in \{\mathsf{A}, \mathsf{Z}\}$, so an environment needs nobody's cooperation for the corruptions it wants, and claiming (Z, P) from its root it makes them identically on both sides. The half that is at issue is the occupant's own initiative: $\mathcal{S}$ may corrupt parties nobody asked it to, or decline what it is asked through the slot, where $\mathcal{D}$ would relay. That is why the right-hand side of Proposition 5.10 is the property against $\mathcal{S}$ and not against the dummy, and why Definition 5.8 keeps the occupant as an argument. What an ideal σ has to satisfy *in that form* is the game against every simulator the emulation may produce; Corollary 5.11 reduces it to the dummy again, and does so by moving the simulator into the environment rather than by asking anything of it.

Nothing escapes the ε, and the register is why. The environment reads **C** through FULLSTATUS, which mediates nothing and returns the whole corrupted set (Remark 3.1), so it may read **C** at any point

and compare it against the parties it corrupted itself together with whatever it instructed the slot to add. Let $\mathcal{Z}$ be an environment the class admits, and let $\mathcal{Z}'$ be $\mathcal{Z}$ with that comparison added and a return of 0 where it fails. On the real side it never fails, $\mathcal{D}$ corrupting what it is told and nothing besides; on the ideal side it fails with whatever probability δ the simulator strays with, in either direction — declining what it is asked, or corrupting unbidden. So $\mathcal{Z}'$ distinguishes with advantage δ, and the emulation hypothesis gives $\delta \leqslant \varepsilon$. One may therefore reason as though $\mathcal{S}$ echoes the corruptions it is instructed to make, at a price already paid — which is Section 3.1's commensurateness, read for properties. It is worth having, but it is not what discharges the hypothesis on the right of Proposition 5.10; absorbing the slot does, and it needs nothing at all of the simulator's conduct.

Remark 5.13 (Leakage on the ideal side)**.** Leakage divides between the sides as corruption does, and needs one check of its own because it does not go through GUARD. Only the slot can read a LEAK: line 2 asks $\mathrm{id}'.\mathsf{F} = \mathsf{A}$, and the environment cannot claim such an identity. So $\mathcal{G}\mathfrak{m}.\mathbf{N}$ does not protect the game there, and something else must: line 4's gate together with the party, since the game's leakage is readable only at a corrupt party, the root is never corrupt, and at every other party $\mathcal{G}\mathfrak{m}$'s LEAK returns $\perp$ by Remark 5.4. The game's state therefore reaches the adversary by that route on neither side. What does differ between the sides is φ's leakage against π's, which $\mathcal{S}$ must produce; that is an ordinary part of emulation, bounded by the same ε, and a property inherits the bound with nothing further to check.

CHAPTER 6

Blockers

Definition 4.10 of Chapter 4 silences an environment on one and the same name closure in both worlds, so the claims open to it do not depend on which world it faces — and Proposition 4.16 and Theorem 4.20 use no relation between the two identity sets at all. What blocking buys is narrower: it makes that closure attainable as an exact silenced set, and so delivers the least restricted admissible class, as the discussion of Definition 4.10 in Chapter 4 records. A *blocker* occupies identities and nothing else. It admits no caller, $\mathbf{N} := \varnothing$, so the conjunct on line 2 of GUARD is false for every call and the caller receives REJ. Its BLOCK core is therefore never entered and its body is a formality. Leakage does not go through GUARD, so a corrupt party's LEAK core is reachable; it returns $\perp$, all a machine with no state has to give. Both are cores, wrapped as Definition 1.2 requires of any standard functionality, and it is the wrapper that makes the first unreachable.

Blocker $\mathcal{B}[\mathcal{F}]$
pid $:= \mathcal{F}.pid$, $\mathbf{P} := \mathcal{F}.\mathbf{P}$, $\mathbf{N} := \varnothing$, $\mathbf{U} := \varnothing$, *par* $:= \perp$

id.BLOCK() from id′
1: **return** $\perp$ // *unreachable due to* GUARD
id.LEAK() from id′
2: **return** $\perp$ // *reachable, but there is no state*

The blocker $\mathcal{B}[\mathcal{F}]$ is indexed by the functionality it stands in for, from which it copies the two fields that fix an identity set and nothing else. By construction $\text{IDs}(\mathcal{B}[\mathcal{F}]) = \text{IDs}(\mathcal{F})$: it occupies precisely the identities of $\mathcal{F}$, session and instance included, and nothing besides. Being unreachable, it stands in for $\mathcal{F}$ as an occupant of those identities and in no other respect.

Write $\mathrm{I}_\pi := \{\mathcal{F}.pid : \mathcal{F} \in \pi\}$ for the process ids a system uses. By Definition 1.2 each $\mathrm{q} \in \mathrm{I}_\pi$ is carried by exactly one member, which we

write π_q, so $\pi = \{\pi_q : q \in I_\pi\}$ and $\mathrm{IDs}(\pi) = \bigcup_{q \in I_\pi} \mathrm{IDs}(\pi_q)$. Blockers are indexed by *ids* and not by members, and the distinction matters: a process id may occur in both worlds carrying different code, in which case φ_q is not a member of π although π uses q, and a blocker for such a q would put two machines at the same id.

Indexing by $I_\varphi \setminus I_\pi$ instead keeps a blocker clear of what is already there. No member of π carries such a q, hence no identity of $\mathrm{IDs}(\pi)$ has q in its first component and $\mathrm{IDs}(\varphi_q) \cap \mathrm{IDs}(\pi) = \varnothing$. A blocker built from φ_q thus lands neither on an identity π already serves nor on the id of an existing member. Distinct ids give pairwise disjoint blockers, and all of them are new to π, so the enlarged system again has distinct ids throughout.

The pair of systems being compared must in addition agree wherever they overlap.

Definition 6.1 (Compatibility)**.** Systems π and φ are *compatible* if, whenever a process id occurs in both, the two functionalities sitting at it agree on their served parties and on whom they admit; the remaining fields — what they use, their parameters, and the code they run — may differ.

Compatibility is used at exactly one point: the ids common to both worlds. No blocker is added for such an id, so whatever it contributes must already agree on the two sides. Without it a shared id could serve different parties in π and in φ, and the difference would go uncovered. Requiring **N** to agree as well is more than the argument needs, since only **P** enters an identity set once the process id is fixed; but it costs nothing, a shared id being carried by the same functionality on both sides in every case we use.

Definition 6.2 (Blocker system, blocked systems)**.** For systems π and φ, the *blocker system* $\textsc{Blockers}(\varphi \setminus \pi)$ is

$$\textsc{Blockers}(\varphi \setminus \pi) \;:=\; \left\{\, \mathcal{B}[\varphi_q] \;:\; q \in I_\varphi \setminus I_\pi \,\right\},$$

and the *blocked systems* of the pair are

$$\overline{\pi} \;:=\; \pi \cup \textsc{Blockers}(\varphi \setminus \pi), \qquad \overline{\varphi} \;:=\; \varphi \cup \textsc{Blockers}(\pi \setminus \varphi).$$

The argument of BLOCKERS is notation rather than an input: the construction reads the ids $I_\varphi \setminus I_\pi$ and φ's occupants there, not the member set the difference denotes — at a shared id carrying different code, $\varphi_q \in \varphi \setminus \pi$ and yet no blocker is built. Each id in $I_\varphi \setminus I_\pi$ is thus covered by the blocker of the functionality living there, with that functionality's party set. Each world gains exactly the process ids the other has and it lacks, all blocking has to achieve. Ids common to both worlds get no blocker, which is where Definition 6.1 does its work.

Informally, Theorem 6.3 says blocking equalises the two worlds: each blocked system is again a system, and for a compatible pair the two occupy one and the same identity set, $\text{IDS}(\pi) \cup \text{IDS}(\varphi)$ — each world having gained exactly the process ids the other has and it lacks.

Theorem 6.3 (Equalisation). *Let π and φ be systems. Then the blocked systems $\overline{\pi}$ and $\overline{\varphi}$ of Definition 6.2 are systems. If π and φ are moreover compatible, then*

$$\text{IDS}(\overline{\pi}) \;=\; \text{IDS}(\overline{\varphi}) \;=\; \text{IDS}(\pi) \cup \text{IDS}(\varphi).$$

Proof. Two claims, and only the second needs compatibility. The first asks the three things Definition 1.2 asks of a system — finiteness, standardness, distinct process ids — of each blocked system, and gets them from the indexing: blockers are indexed by the ids the other world has and this one lacks, so they land on no existing member. The second compares the two identity sets index by index over $I_\pi \cup I_\varphi$. At an id only one world uses, both sides contribute the same set, a blocker copying the two fields that fix an identity set. At a shared id no blocker is built, so the two contribute $\text{IDS}(\pi_q)$ and $\text{IDS}(\varphi_q)$ as they stand — and compatibility supplies the one thing the indexing does not, that the served party sets agree. That is the only place it is used. For the first claim, being a system asks three things of $\overline{\pi}$: that it be finite, that its members be standard, and that they carry distinct process ids. Finiteness is immediate, the added members being indexed by $I_\varphi \setminus I_\pi \subseteq I_\varphi$. Standardness is inherited: a blocker carries the wrappers Definition 1.2 asks of a standard functionality, BLOCK as in Section 3.5 and LEAK as in Section 3.2, and its process id is copied from a member of φ, so its name is none of A, Z, C. For distinctness, the members of $\text{BLOCKERS}(\varphi \setminus \pi)$ are the $\mathcal{B}[\varphi_q]$ with $q \in I_\varphi \setminus I_\pi$, and $\mathcal{B}[\varphi_q].\mathit{pid} = \varphi_q.\mathit{pid} = q$. These

ids are pairwise distinct and none lies in I_π, so the members of $\overline{\pi}$ carry distinct process ids and $I_{\overline{\pi}} = I_\pi \cup I_\varphi$. The same argument applies to $\overline{\varphi}$.

For the second, $\mathrm{IDs}(\mathcal{B}[\mathcal{F}]) = \mathrm{IDs}(\mathcal{F})$ holds for every $\mathcal{F}$, since a blocker copies the two fields that determine an identity set. Hence

$$\mathrm{IDs}(\overline{\pi}) = \bigcup_{q \in I_\pi} \mathrm{IDs}(\pi_q) \cup \bigcup_{q \in I_\varphi \setminus I_\pi} \mathrm{IDs}(\varphi_q),$$
$$\mathrm{IDs}(\overline{\varphi}) = \bigcup_{q \in I_\varphi} \mathrm{IDs}(\varphi_q) \cup \bigcup_{q \in I_\pi \setminus I_\varphi} \mathrm{IDs}(\pi_q),$$

and both unions are indexed by $I_\pi \cup I_\varphi$. Compare them at each q in that set. If $q \in I_\pi \setminus I_\varphi$ then both contribute $\mathrm{IDs}(\pi_q)$, and if $q \in I_\varphi \setminus I_\pi$ then both contribute $\mathrm{IDs}(\varphi_q)$. If q lies in both, the first contributes $\mathrm{IDs}(\pi_q)$ and the second $\mathrm{IDs}(\varphi_q)$. Both are carried at the index q, so $\pi_q.\mathit{pid} = \varphi_q.\mathit{pid} = q$ outright, and compatibility supplies the one thing that does not follow from the indexing, $\pi_q.\mathbf{P} = \varphi_q.\mathbf{P}$. An identity set is fixed by those two fields, so $\mathrm{IDs}(\pi_q) = \mathrm{IDs}(\varphi_q)$. The two unions therefore agree term by term, and each equals

$$\bigcup_{q \in I_\pi} \mathrm{IDs}(\pi_q) \cup \bigcup_{q \in I_\varphi} \mathrm{IDs}(\varphi_q) = \mathrm{IDs}(\pi) \cup \mathrm{IDs}(\varphi).$$

□

The second claim is a table. A process id falls in one of three places, and the theorem is what sits at each:

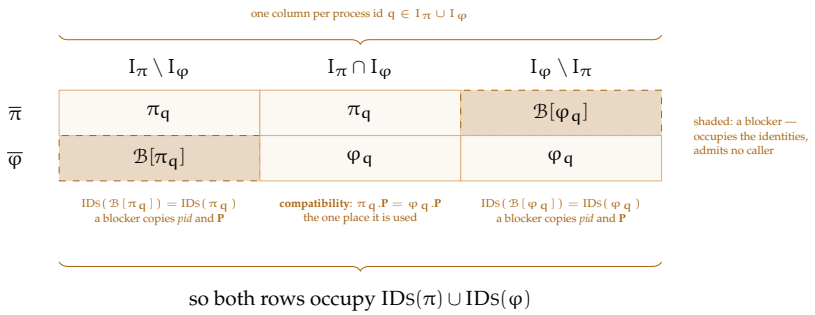


Figure 3: Where a process id can fall, and what blocking does at each place. The three columns are the three cases of the second claim of Theorem 6.3.

In the outer columns the world that lacks the id gains a blocker, and a blocker occupies exactly the identities of the functionality it stands

in for, having copied the only two fields that fix an identity set. So the two rows contribute the same identities there — although one contributes a working functionality and the other a machine that answers nothing, which is the point of shading them differently. Equalising identity sets is all blocking has to achieve, and it had better be all it achieves: a blocker that answered anything would change what an environment could learn, and the comparison it was built to enable would be worthless.

The middle column is where the hypothesis lives. No blocker is built for a shared id, so the two rows contribute $\mathrm{IDs}(\pi_q)$ and $\mathrm{IDs}(\varphi_q)$ as they stand, and nothing in the indexing makes those equal — both are carried at q, so the process ids agree outright, but the served party sets need not. Compatibility is exactly the assumption that they do, and this column is the only place it is used. Read the other way, it is also what the assumption costs: a shared id serving different parties in the two worlds is a difference blocking cannot cover, because the one device it has — adding an occupant — is unavailable where an occupant is already present.

Only the second claim needs any of this. That $\overline{\pi}$ and $\overline{\varphi}$ are systems at all asks nothing of compatibility, which is why the theorem states the two separately and why Proposition 6.4 below may lean on the first alone.

Blocking enlarges a system, and enlarging a system is one of the two ways Definition 1.3 could come undone.

Proposition 6.4 (Blocking preserves well-formedness)**.** *Let π and φ be systems with $\mathrm{WF}(\pi)$ and $\mathrm{WF}(\varphi)$. Then $\mathrm{WF}(\overline{\pi})$ and $\mathrm{WF}(\overline{\varphi})$.*

Proof. $\overline{\pi}$ is a system by the first claim of Theorem 6.3, which asks no compatibility, so $\mathrm{WF}(\overline{\pi})$ is a legal question. We argue for $\overline{\pi}$; the argument for $\overline{\varphi}$ is the same with $\mathrm{WF}(\varphi)$ in place of $\mathrm{WF}(\pi)$. Take $\mathcal{F} \in \overline{\pi}^{+}$ and $(\mathcal{F}', \mathsf{R}) \in \mathcal{F}.\mathbf{U}$.

If $\mathcal{F}$ is one of the added blockers there is nothing to check: a blocker has $\mathbf{U} = \varnothing$ by construction, so it contributes no entry. Otherwise $\mathcal{F}$ lies in π^{+}, and $\mathrm{WF}(\pi)$ supplies a witness $\mathcal{G} \in \pi^{+}$ for that entry. That witness is still present, since $\pi^{+} \subseteq \overline{\pi}^{+}$, and blocking alters no member's fields, so $\mathcal{G}.\mathsf{F}$, $\mathcal{G}.\mathsf{s}$ and everything R reads are what they were. The three conditions therefore still hold of the same $\mathcal{G}$.

The argument uses nothing about blockers beyond their empty **U**: enlarging a system to another system preserves well-formedness whenever the members added carry no entries of their own, because a witness once present is never removed and the fields it is judged on never change. Neither compatibility nor Theorem 6.3's second claim is used. □

Remark 6.5 (Tight realizations)**.** The closure keeps the environment's own claims off the used names; hosting is the one route around it. An admissible environment may host a machine at an *unoccupied* identity of a used name — an invented second instance of a caller, say — and a hosted machine is a full interface, claiming its own identity through its own silencer rather than through *par* (Definition 4.13). Against a pair whose members admit that name, the two worlds can then answer differently at an identity only one of them occupies, and no condition on the class can refuse the intruder without refusing absorption: the absorbed environment of Definition 4.15 hosts machines of exactly the same shape — outside occupants at admitted names — and differs from the intruder only in the code it runs, which admissibility rightly does not read. Part of the repair belongs to the realization. Call a system *tight at* a subset I of its members if every member outside I admits whole process ids occupied by the system, rather than names, and serves only parties each of its admitted callers serves — with SERVES this makes a private member's party set exactly its callers', which is what a private subroutine is. Tightness closes the host route *to the members outside* I: an unoccupied process id lies outside every pinned **N** and fails line 2, and the party condition makes every identity a pinned caller could speak from occupied, where Definition 4.10 forbids hosting; so a call an intruder addresses to a member outside I is refused, on the ideal side too, by the blocker's empty **N** or by EXEC finding no occupant. What tightness does *not* close is the route to a member of I itself: an interface member that admits a bare used name — the local pattern of Chapter 1 — is reachable by a hosted fake instance exactly as a private member would be, and tightness constrains no member of I. Pinning those too, or admitting only $\mathbf{Z} \cup \mathbf{A}$ where the environment reaches them directly, would close it; we do not pursue a general sufficient condition here. Tightness is a hygiene

condition in the spirit of a subroutine-respecting protocol, necessary for the private members not to leak; a full characterization of when comparisons against blocked pairs are faithful we leave open.

CHAPTER 7

Relocation and Renaming

Chapter 1 records that a functionality's code hardly reads its own session and never its instance number: s occurs at line 3 of GUARD, in the wrappers that invoke it, and in Definition 1.3, and i occurs nowhere. This chapter turns that observation into a statement. Renaming sessions and instances is an isomorphism of executions, so an emulation proved at one process id holds at every other, within the same ε and at the same budgets. One hygiene condition is needed and no more — that a functionality reachable from outside every session be reachable from every session, which every machine the paper specifies satisfies — and isolating it is half the work. That is what lets a realization be written for a single session, as $\mathcal{F}_{\mathsf{Sig}}$ of Chapter 15 is, and read as covering the rest.

Definition 7.1 (Relocation)**.** A *relocation* is a permutation θ of the process ids such that

(i) $\theta(q).F = q.F$ for every q;

(ii) $q.s = q'.s \iff \theta(q).s = \theta(q').s$ for all q, q' with $q.F, q'.F \in \{0,1\}^*$;

(iii) $\theta(q) = q$ for each of the three ids $q \in \{(A,0,0),(Z,0,0),(C,0,0)\}$ the execution supplies.

Conditions (i) and (ii) say exactly that on the standard names

$$\theta(F,s,i) \;=\; \big(F,\ \hat{\theta}(s),\ \theta_{F,s}(i)\big)$$

for a permutation $\hat{\theta}$ of session ids and, for each name and session, a permutation $\theta_{F,s}$ of instance numbers — we write the two components of a relocation this way rather than with letters of their own, σ and ι being spoken for by Proposition 4.19 and Definition 4.17, whose notation the proofs below import. Names are kept, because GUARD and WF match by name and nothing may become a different functionality;

sessions are permuted *as blocks*, because line 3 asks whether two ids share a session and not which; and instances are permuted freely within a block, that component being read nowhere.

Condition (iii) pins the three process ids that occur as *literals* in code: A, in the hook $(A, \mathrm{id.P})$ of MEDIATE; C, in the register read $(C, \mathrm{id.P})$ of lines 2 and 6; and Z, at line 2 and in the relay of line 5. Every identity named outright in the paper is one of these three paired with a party in scope, so pinning the three pins them all — and it pins those three and no more. It must not be read as pinning every id of a reserved name, and the difference is not cosmetic: by Chapter 1 the set $\mathbf{A}$ holds a triple $(A, s, \mathfrak{i})$ for every s and $\mathfrak{i}$, only $(A, 0, 0)$ ever occurring, so a condition fixing all of them and read together with (ii) unrestricted would give $\hat{\theta}(s) = s$ at every s and leave no relocation that moves a session at all. Restricting (ii) to the standard names is the other half of the same care: $\hat{\theta}$ is determined there, and the ids of a reserved name that never occur are left to move harmlessly inside their own class, which Proposition 7.5 shows is where they stay.

Nothing here pins session 0, and $\hat{\theta}$ is free to move it. That freedom has a price, and it is worth seeing exactly where, since this is the only point at which the framework tells one session from another. Line 3 compares the *caller's* session while GLOBAL is tested of the callee, and the three machines the execution supplies sit in session 0; so a call from $\mathcal{A}$ or $\mathcal{Z}$ to a callee that is not global passes that line exactly when the callee's own session is 0. A callee of that shape would tell a relocation moving session 0 from one leaving it alone. Nothing else can: for every other callee the supplied machines are either waived by GLOBAL or refused at line 2 before the session is looked at. So one condition on the callee closes the gap, and it is a hygiene condition rather than a restriction on θ.

Definition 7.2 (Session-uniform)**.** A functionality $\mathcal{F}$ is *session-uniform* if

$$\mathcal{F}.\mathbf{N} \cap (\mathbf{A} \cup \mathbf{Z}) \neq \varnothing \quad \Longrightarrow \quad \mathrm{GLOBAL}(\mathcal{F}),$$

so that it admits a caller from outside every session only if it is reachable from every session. A system is session-uniform if each of its members is, and an environment if each functionality it hosts is.

The sessions a session-uniform functionality can be reached from do not depend on the session it sits in, which is what relocation needs of it. The three supplied machines meet the condition outright, all three being global — $\mathcal{A}$ and $\mathcal{C}$ by the first disjunct of GLOBAL, $\mathcal{Z}$ by the second — and so does every occupant of the adversary slot, global by that same second disjunct, and so does a blocker, whose **N** is empty and which line 2 therefore refuses whatever the session. Those are every **N** the paper declares; $\mathcal{F}_{\mathsf{Sig}}$ takes its as a parameter, where the condition is one line to check. What the condition excludes is the shape Chapter 1 leaves open in saying that a local $\mathcal{F}$ may itself declare whether **A** and **Z** may call it: a local functionality admitting the adversary is reachable from the slot in session 0 and in no other, and that is the one thing a relocation cannot carry. Remark 7.17 records the alternative, a one-line widening of line 3 that would dispense with the condition altogether.

Definition 7.3 (Action of a relocation)**.** A relocation θ acts on identities by $\theta(\mathit{pid}, \mathsf{P}) := (\theta(\mathit{pid}), \mathsf{P})$, leaving the party component alone, and on identity sets pointwise. It sends a functionality $\mathcal{F}$ to $\theta\mathcal{F}$ with

$$\theta\mathcal{F}.\mathit{pid} := \theta(\mathcal{F}.\mathit{pid}), \quad \theta\mathcal{F}.\mathbf{P} := \mathcal{F}.\mathbf{P}, \quad \theta\mathcal{F}.\mathbf{N} := \theta(\mathcal{F}.\mathbf{N}),$$
$$\theta\mathcal{F}.\mathbf{U} := \theta(\mathcal{F}.\mathbf{U}), \quad \theta\mathcal{F}.\mathit{par} := \theta(\mathcal{F}.\mathit{par}),$$

where θ acts on a **U** entry by $\theta(\mathcal{F}', \mathsf{R}) := (\theta\mathcal{F}', \mathsf{R})$ and on *par* through whatever identities and process ids it contains; the code of $\theta\mathcal{F}$ is the code of $\mathcal{F}$ with every process id occurring in it — in a call target, a claimed identity, a test, or a stored value — replaced by its image; and every identity set a machine *declares* rather than computes is replaced by its image too, which for a bundle (Definition 4.13) means its occupied set, its outward set and any routing it fixes by hand. It acts on a system memberwise, and on any other machine of the execution the same way, a bundle's hosted members included, so that $\theta\mathcal{Z}$ hosts the images of what $\mathcal{Z}$ hosts. We require the requirement predicates to be equivariant, $\mathsf{R}(\mathcal{F}, \mathcal{G}) \iff \mathsf{R}(\theta\mathcal{F}, \theta\mathcal{G})$, as SERVES is, it reading only **P** and a membership of **N**. Every clause above relabels pointwise, so the action is one: ID acts as the identity and $(\theta'\theta)\mathcal{F} = \theta'(\theta\mathcal{F})$. In particular θ^{-1} undoes θ on machines as it does on process ids, which is what lets the results below argue in both directions.

Relabelling the process ids a program *names* is not by itself relabelling what it does, and the gap between the two is a condition on the machine model. We state it, because Lemma 7.9 is false without it.

Definition 7.4 (Identity-atomic code)**.** Code is *identity-atomic* if it handles identities and process ids as opaque tokens. It may project one to its name or party component; test two of them, or two of their name, party, session or instance components, for equality; test one for membership in a set of them the code has been given, as line 2 tests $\mathcal{F}.\mathbf{N}$ and SILENCE tests $\mathcal{I}$; form a set of them by those tests, as lines 2 and 3 form $\bar{\mathcal{I}}$ and line 5 forms $\{\mathtt{id}'' \neq \mathtt{id}\}$; name one outright; and destructure a value that carries one. It may not compute with a session or an instance number: no arithmetic, no ordering, and no session or instance written down on its own, apart from inside an identity or process id named outright. Each of these counts as one step, whatever the identities involved, as Definition 4.30 counts a call as one answer.

Every machine in this paper is identity-atomic. The predicate GUARD compares sessions for equality and nothing else, no core reads $\mathtt{i}$ at all, and the only process ids named outright are the three of condition (iii). We take the model to be this throughout, and it is what makes Definition 7.3 sound: on identity-atomic code the substitution there does not merely rewrite literals but computes the relabelled behaviour, each permitted operation being equivariant — projections because θ leaves names and parties alone, equality and membership because θ is injective and the sets tested and formed are relabelled with the code, a named identity because substitution replaces it by its image. It is also what makes a machine and its image take the same steps: they perform the same operations in the same order, the control flow being the same on both, and the last clause of the definition charges one step for each however the ids inside it have been renamed. So the budgets of Definition 4.30 carry over unchanged — which a cost model charging by the symbol would not grant, a relocation being free to lengthen the encoding of a session.

Without the restriction both fail. A core returning $\mathtt{id.i} \bmod 2$ names no process id, so substitution leaves it untouched; carried to another instance number it answers differently, and the relabelled machine is not the original relabelled. A core calling $(\mathtt{F}, \mathtt{id.s}, \mathtt{id.i} + 1)$ misses its

target in the same way. Neither is a machine anyone would write — Definition 4.6 permits an *arbitrary* core, which is why the exclusion has to be stated rather than assumed.

Proposition 7.5 (Basic invariances)**.** *Let θ be a relocation, $\mathcal{F}$ a functionality and π a system. Then*

(i) $\text{IDs}(\theta\mathcal{F}) = \theta\big(\text{IDs}(\mathcal{F})\big)$*, and occupants carrying pairwise disjoint identity sets keep them;*

(ii) θ *carries each name class onto itself, so* $\theta(\mathbf{Std}) = \mathbf{Std}$, $\theta(\mathbf{A}) = \mathbf{A}$ *and* $\theta(\mathbf{Z}) = \mathbf{Z}$*, and every* $\mathbf{N}$ *declared by names — a local* $\{pid : pid.\mathsf{F} = \mathsf{F}^{\uparrow}\}$ *among them — is its own image;*

(iii) $\text{GLOBAL}(\theta\mathcal{F}) \iff \text{GLOBAL}(\mathcal{F})$*;*

(iv) $\langle\theta\pi\rangle = \theta\big(\langle\pi\rangle\big) = \langle\pi\rangle$*;*

(v) $\theta\pi$ *is a system, and* $\text{WF}(\pi) \iff \text{WF}(\theta\pi)$*; and*

(vi) θ^{-1} *is again a relocation.*

Proof. Six items, and the useful thing to see is which of Definition 7.1's three conditions each one spends. (i) is conditions (i) and injectivity: an identity set is fixed by two fields, one relabelled and one untouched. (ii) is condition (i) alone — name classes map into themselves, and being a bijection on a partition, onto them as well, which is what makes the three reserved id sets invariant. (iii) then follows from (ii), GLOBAL reading only a name and a subset of **Std**. (iv) is condition (i) again, name closures being specified by names. (v) is the substantial one: well-formedness transports because names match, the session conjunct is either waived by GLOBAL or read between two standard ids where condition (ii) preserves it, and the relation R is equivariant by Definition 7.3; the converse runs the same argument on θ^{-1}. (vi) is why that is legitimate — conditions (i) and (ii) are an identity and an equivalence, so they hold of the inverse, and (iii) holds because θ fixes those three ids. (i) An identity set is fixed by *pid* and **P**, and a relocation relabels the first while leaving the second alone; disjointness survives because θ is injective on identities.

(ii) By (i) of Definition 7.1 each name class maps into itself. The classes partition the process ids, so for x in the class of F the point $\theta^{-1}(x)$ lies in some class, which maps into itself and therefore is the class of F; so the map is onto that class as well. The three displayed sets

are unions of name classes, and a set of ids specified by a name is a union of them.

(iii) GLOBAL($\mathcal{F}$) reads **Std** $\subseteq \mathcal{F}$.**N** or $\mathcal{F}$.F $\in \{A, Z, C\}$. The second is untouched by (i) of Definition 7.1; for the first,

$$\textbf{Std} \subseteq \theta(\mathcal{F}.\textbf{N}) \iff \theta^{-1}(\textbf{Std}) \subseteq \mathcal{F}.\textbf{N},$$

and $\theta^{-1}(\textbf{Std}) = \textbf{Std}$ by item (ii) above, θ being a bijection.

(iv) A name closure is specified by the names its system uses; θ keeps those, so $\theta\pi$ uses the names π does and $\langle\theta\pi\rangle = \langle\pi\rangle$. And $\langle\pi\rangle$ is a union of name classes crossed with party ids, so item (ii) gives $\theta(\langle\pi\rangle) = \langle\pi\rangle$ as well.

(v) The members of $\theta\pi$ carry distinct process ids, θ being injective, and are standard, their names being unchanged and their wrappers untouched by relabelling; the family is finite. For WF, take $\mathcal{F} \in \pi^+$ and an entry $(\mathcal{F}', R)$ with witness $\mathcal{G}$. Names match on both sides. For the session conjunct $\mathcal{G}.s = \mathcal{F}.s$, either GLOBAL($\mathcal{G}$) and item (iii) above waives the conjunct on both sides, or $\mathcal{G}$ is standard — and then so is $\mathcal{F}$, the entries of the three supplied machines naming only each other and $\mathcal{C}$.**U** being empty — so that both ids are standard and (ii) of Definition 7.1 preserves the test. Finally $R(\mathcal{F}, \mathcal{G}) \iff R(\theta\mathcal{F}, \theta\mathcal{G})$ by the equivariance Definition 7.3 requires. The entries of the three supplied machines are settled identically on both sides, every field SERVES reads of them being fixed by items (ii) above and (iii) of Definition 7.1. For the converse, run the same argument on θ^{-1}, a relocation by item (vi), using $\theta^{-1}(\theta\pi) = \pi$ from Definition 7.3.

(vi) Conditions (i) and (ii) of Definition 7.1 are stated as an identity of components and as an equivalence, and so hold of the inverse; (iii) does because θ fixes those three ids. □

Two of these deserve a word. By (ii), only the pinned **N** declarations of Remark 6.5 move at all, and they move with the members they pin. And by (iv) name closures are invariant, which is what makes Proposition 7.10 come out against an environment class constrained exactly as before rather than one merely corresponding to it.

Definition 7.6 (Pid-blind core)**.** A core is *pid-blind* if $\theta\text{OP} = \text{OP}$ for every relocation θ: it names no process id that a relocation moves. It may compare the sessions of two identities in scope, as GUARD

does, and it may name one of the three process ids that (iii) of Definition 7.1 fixes — but not some other id of a reserved name, which (iii) leaves free to move.

Every core in this paper is pid-blind. The wrappers of Chapter 2 read names and parties only; MEDIATE hooks to $(\mathtt{A}, \mathtt{id.P})$ and lines 2 and 6 to $(\mathtt{C}, \mathtt{id.P})$, both reserved; the $\mathcal{Z}$ box computes $\bar{\jmath}$ from *par* by tests on names and parties; $\mathcal{D}$ places the call its payload names; and $\mathcal{F}_{\mathsf{Sig}}$ reads $\mathtt{id.P}$ and its own **P** and nothing else of an identity.

That is a stronger property than Lemma 7.9 needs, and a different one. The lemma needs Definition 7.4, which every machine of an execution must meet, the environment and the occupant of the adversary slot included — and neither of those can be pid-blind, each having to name the process ids it drives. What pid-blindness buys is the reading of the conclusion: for a system of pid-blind cores, $\theta\pi$ is π with five fields relabelled and not a line of code rewritten, so a relocated protocol is the protocol, run elsewhere.

Before the lemma, the other half of the question the definition answers: which fields an execution reads at all.

Proposition 7.7 (Inert fields)**.** *Let $\mathcal{F}$ and $\mathcal{F}'$ be standard functionalities agreeing on pid,* **P***, par and code.*

(i) If they agree on **N** *as well, so that they differ in* **U** *alone, then no execution distinguishes them: in the two systems got by putting one or the other at that process id, every activation produces the same configuration.*

(ii) If they differ in **N** *alone, then from the same state and on the same call* $\mathtt{id.FULLOP(in)}$ *from* $\mathtt{id}'$*, either both wrappers refuse at line 1 with* REJ*, or both admit and enter the same core at the same point in the same state.*

Proof. By inspection of where the two fields are read during an execution. The field **U** is read only by the static check of Definition 1.3 and the results about it, never by EXEC or by any interface, which gives (i). The readers of **N** inside an execution are two: line 2 of GUARD, and the predicate GLOBAL and hence line 3. Both sit inside $\text{GUARD}_{\mathcal{F}}$, which line 1 of the full interface **require**s before anything else. (The field is read outside an execution too — by Definition 1.3 through those

two, by Definition 6.1, and by the typing conditions of Definition 4.7 and Remark 6.5 — but those are conditions on a system, not steps of one.) So the field decides that verdict and is read nowhere else in the wrapper: the lines between line 1 and the core — the register read, the corruption gate, the first MEDIATE hook, the silenced set $\{\mathtt{id}'' \neq \mathtt{id}\}$ — are functions of *pid*, **P**, *par*, the code and the call, so two wrappers that both admit reach line 5 together and enter the same core at the same point in the same state. That is (ii). □

So **N** is access control and nothing else, and **U** is inert in every execution, as Remark 4.8 already records for the adversary slot. Inertness past the guard is a statement about one step, and it cannot be strengthened to one about the value returned. The obvious reason is that a call one admits and the other refuses sends the two states apart. The less obvious one is that even a call both admit can end differently, because the core entered at line 5 may re-enter this very wrapper — Section 3.6 permits an instance to be re-entered, and line 5 lets a core claim its own identity — and the inner call meets $\text{GUARD}_{\mathcal{F}}$ again, where the field differs. A core that places $\mathtt{id}.\text{FULLOP}'()$ as $\mathtt{id}$ and returns 1 just if the answer is not REJ separates an $\mathcal{F}$ admitting its own process id from an $\mathcal{F}'$ that does not, on a call both admitted. What survives is the exact claim: the two differ in guard verdicts alone, and a single differing verdict, outer or inner, is enough to separate them.

Remark 7.8 (The two fields that are read)**.** Neither **P** nor *par* admits a statement of this kind, and for different reasons. The field **P** fixes $\text{IDS}(\mathcal{F})$, hence which calls EXEC dispatches to $\mathcal{F}$ at all and the second conjunct of line 1; and cores read it directly — INITIALIZE of $\mathcal{F}_{\mathsf{Sig}}$ types $\mathtt{VK} : \mathcal{F}_{\mathsf{Sig}}.\mathbf{P} \to \mathcal{K} \cup \{\square\}$, and the test $\exists\, \mathrm{P}' \notin \mathbf{C}$ of VERIFY quantifies over that domain. So enlarging or shrinking a party set is not a symmetry. What *is* one is *renaming*: a permutation β of $\{0,1\}^*$ extended to fix A and Z, acting on identities, on **P**, on **C** and on the party-indexed state, gives an isomorphism of executions by the argument of Lemma 7.9, provided cores read party ids only through equality and indexing — as $\mathcal{F}_{\mathsf{Sig}}$ does, its $\exists \mathrm{P}' \notin \mathbf{C} : vk = \mathtt{VK}[\mathrm{P}']$ being equivariant. The two places a party id enters otherwise survive it as well: line 3 names the party Z, which β fixes, and FULLCORRUPT tests $\mathtt{id}.\mathrm{P} \in \{0,1\}^*$, a set β permutes.

The field *par* is read by the code by definition of the field, and $\mathcal{Z}.par$ is read by Definition 4.10 besides; there is nothing to be invariant in. A relocation moves it, and moves it to itself whenever it is declared by names, which $\mathcal{Z}.par := \langle\overline{\pi}\rangle$ is.

Informally, Lemma 7.9 says that relabelling sessions and instances throughout an execution changes nothing observable: it is Proposition 4.19's induction with θ in place of the absorption bijection, and every check is the equivariance of one line of one wrapper.

Lemma 7.9 (Relocation)**.** *Let θ be a relocation, π a session-uniform system, $\mathcal{X}$ an occupant of the adversary slot (Definition 3.3), and $\mathcal{Z}$ a session-uniform environment hosting at no identity of* $\text{IDs}(\pi)$*, so that* EXEC *is defined at the four. Then*

$$\text{EXEC}(\theta\pi, \theta\mathcal{Z}, \theta\mathcal{X}, \mathcal{C}) \qquad \textit{and} \qquad \text{EXEC}(\pi, \mathcal{Z}, \mathcal{X}, \mathcal{C})$$

are identically distributed.

Proof. The correspondence of Definition 4.17 at $\iota := \theta$, read up to θ: two states, program points or caller chains correspond when one is the other's image. Since θ is a bijection the relation is symmetric, and the proof is again an induction on activations, whose step the four paragraphs below discharge. *Machines* checks that occupancy is relabelled bijectively, so both executions are defined and start in correspondence. *Guards* takes the three conjuncts of GUARD in turn; the first two are injectivity, and the third is the one paragraph that does real work, since line 3 is the only place the framework tells one session from another — this is where session-uniformity is spent, and without it a relocation moving session 0 would change the verdict. *Silencers, mediation, the register* checks the wrappers, and observes that the register needs no image at all, corruption being indexed by party alone. *Routing, claims, token* closes the step, identity-atomic code taking the same branches on both sides. The correspondence is that of Definition 4.17 with $\iota := \theta$ — each machine paired with its image, $\mathcal{C}$ with itself — and with (C1), (C3) and (C4) read up to θ: two states, two program points, two chains of suspended callers correspond when one is the θ-image of the other. Since θ is a bijection on identities the relation is symmetric, and as

before the proof is an induction on activations. The paragraphs below discharge the step.

Machines. $\theta\pi$ is a system and occupancy is relabelled bijectively, by (i) and (v) of Proposition 7.5; since $\mathcal{Z}$ hosts clear of IDs(π) and $\theta\mathcal{Z}$ hosts at the images of what $\mathcal{Z}$ hosts, the occupants are pairwise disjoint on the right exactly where they are on the left, so both executions are defined. Line 1 initialises $\mathcal{C}$ and every member on both sides, and a member's initial state is its image's relabelled, giving (C1) at the outset.

Guards. Take a call $\mathtt{id}.\textsc{FullOp}(\mathtt{in})$ from $\mathtt{id}'$ at $\mathcal{F}$ and its image at $\theta\mathcal{F}$. Line 1: $\mathtt{id}.\mathit{pid} = \mathcal{F}.\mathit{pid}$ iff $\theta(\mathtt{id}.\mathit{pid}) = \theta(\mathcal{F}.\mathit{pid})$ by injectivity, and $\mathtt{id}.\mathrm{P} \in \mathbf{P}$ is untouched. Line 2: $\mathtt{id}'.\mathit{pid} \in \mathcal{F}.\mathbf{N}$ iff $\theta(\mathtt{id}'.\mathit{pid}) \in \theta(\mathcal{F}.\mathbf{N})$, again by injectivity, and the party test reads components θ leaves alone. Line 3 is a disjunction whose second disjunct agrees on the two sides by (iii) of Proposition 7.5, so suppose GLOBAL$(\mathcal{F})$ fails — it fails on both sides together or on neither — and consider the first. Every machine of the execution is session-uniform: π and $\mathcal{Z}$'s hosted copies by hypothesis, and $\mathcal{X}$, $\mathcal{C}$ and $\mathcal{Z}$'s own interfaces because their process ids are A, C and Z, which GLOBAL admits by its second disjunct. So $\mathcal{F}.\mathbf{N}$ meets neither **A** nor **Z**, and a claim carrying the id of $\mathcal{A}$ or of $\mathcal{Z}$ is refused at line 2 on both sides — (iii) fixing those ids, and $\theta(\mathcal{F}.\mathbf{N})$ missing **A** and **Z** just as $\mathcal{F}.\mathbf{N}$ does, those two sets being θ-invariant by (ii) of Proposition 7.5. The third supplied id cannot arrive at all, $\mathcal{C}$ placing no calls. The claims that reach line 3 therefore carry standard process ids, and so does $\mathtt{id}$, the name of $\mathcal{F}$ lying in $\{0,1\}^*$ once GLOBAL$(\mathcal{F})$ fails; so $\mathtt{id}'.\mathrm{s} = \mathtt{id}.\mathrm{s}$ holds of the images exactly when it holds of the originals, by (ii) of Definition 7.1. This is the one paragraph session-uniformity is for; without it a relocation moving session 0 would change the verdict here. So the verdict is the same on both sides, at every callee, including the leakage interface of Section 3.2, whose three requirements read only what these lines read.

Silencers, mediation, the register. The silenced set of line 5 is $\{\mathtt{id}'' \neq \mathtt{id}\}$, whose image is $\{\mathtt{id}'' \neq \theta\mathtt{id}\}$, so a core claims its own identity on both sides and nothing else. The set $\bar{\jmath}$ of the $\mathcal{Z}$ box is computed from $\theta(\mathit{par})$ by tests on names and parties, so it is the image of the set computed on the left, and admits exactly the image of what that admits. The wrapper MEDIATE reads **C**, $\mathtt{id}.\mathrm{P}$ and $\mathtt{id}'.\mathrm{F}$ and hooks to $(\mathrm{A}, \mathtt{id}.\mathrm{P})$; lines 2 and 6 address $(\mathrm{C}, \mathtt{id}.\mathrm{P})$. Every id named outright here is one of the three that (iii) of Definition 7.1 fixes. And the register itself needs no image: **C** collects *party* ids, which a relocation does not touch, so $\mathcal{C}$ is literally the

same machine holding the same set at corresponding configurations. Corruption being indexed by party alone (Section 3.1) is what buys this.

Routing, claims, token. EXEC dispatches a call to the occupant of its target identity, and targets correspond under θ while occupancy does by the first paragraph; a call addressed to an unoccupied identity is answered REJ on the left exactly when its image is on the right. Claims correspond by the previous paragraph. The code being identity-atomic (Definition 7.4), a machine and its image take the same branches and place calls that are θ-images of one another, so they reach the same program point with values related by θ; and each reads the same coins in the same order, which is (C2). The token discipline is unchanged, EXEC having one operation on both sides and the same chain of suspensions arising, which is (C3) and (C4).

Corresponding halting configurations output alike, the output being $\mathcal{Z}$'s and $\theta\mathcal{Z}$'s return at (Z, Z), an identity (iii) of Definition 7.1 fixes. So the two executions agree as distributions, in particular on the probability of returning 1. □

The lemma is easier to see than to state. One execution, and the same execution with every process id renamed, side by side:

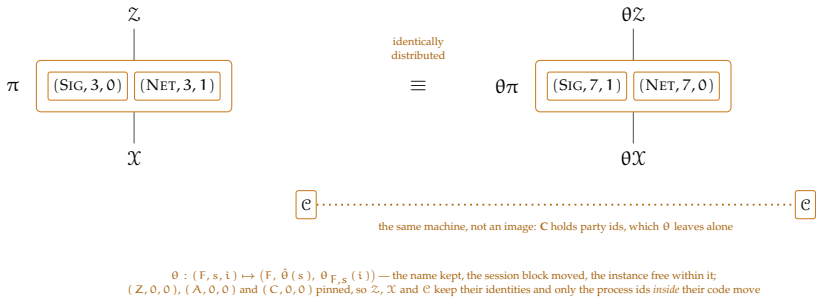


Figure 4: One execution beside the same execution with every process id renamed. Lemma 7.9 is that the two agree as distributions.

Everything a relocation may do is legible in the two systems. The *name* is kept, because GUARD and WF match by name and nothing may quietly become a different functionality. The *session* moves as a block — both members go from 3 to 7 together — because line 3 asks whether caller and callee *share* a session and never which one they are in. And the *instance* is free to move within that block, 0 and 1 here changing

places, because no line of any code reads it. The three ids that occur as literals are pinned, so $\mathcal{Z}$, $\mathcal{X}$ and $\mathcal{C}$ sit at the same identities on both sides and only the process ids *inside* their code are rewritten.

$\mathcal{C}$ is not even that: it is the same machine holding the same set, because $\mathbf{C}$ collects party ids and a relocation does not touch a party. Indexing corruption by party alone (Section 3.1) is what buys this, and it is worth noticing how much it buys — a register indexed by process id would have to be carried across by θ like everything else, and the corruption pattern would then be a thing a relocation could disturb.

One warning the picture cannot give. Drawn this way the lemma looks free, and it nearly is: every guard conjunct, every silencer, every mediation hook reads only names, parties and the three pinned ids, all of which either survive θ or are fixed by it. The exception is the single session test, and it is the reason session-uniformity is a hypothesis rather than a remark — a callee reachable from outside every session but sitting in one of them would answer the supplied machines differently before and after a θ that moves session 0, and the two executions would come apart at that one line. That paragraph of the proof is the whole of the work; the rest is bookkeeping the picture already shows.

Proposition 7.10 (Emulation transports)**.** *Let θ be a relocation and π, φ systems; for (ii) and (iii) let the two of them and every environment of the classes named be session-uniform, which (i) does not need. Then*

(i) $\mathbb{Z}_{\theta\pi,\theta\varphi} = \theta(\mathbb{Z}_{\pi,\varphi})$*, and θ carries adversaries to adversaries and simulators to simulators, fixing the dummy of Section 4.4;*

(ii) if π UC-emulates φ within ε against $(\mathbb{A}, \mathbb{S}, \mathbb{Z})$, then $\theta\pi$ UC-emulates $\theta\varphi$ within ε against $(\theta\mathbb{A}, \theta\mathbb{S}, \theta\mathbb{Z})$; and

(iii) for budgeted systems and every $\mathbf{a}, \mathbf{s}, \mathbf{z}$, $\mathrm{ADV}^{\mathrm{UC}}_{\theta\pi,\theta\varphi}[\mathbf{a},\mathbf{s},\mathbf{z}] = \mathrm{ADV}^{\mathrm{UC}}_{\pi,\varphi}[\mathbf{a},\mathbf{s},\mathbf{z}]$.

Proof. Three claims, and each rests on a different part of Proposition 7.5. (i) is bookkeeping about the classes: both clauses of Definition 4.10 transport, the *par* clause because name closures are θ-invariant and the hosting clause by injectivity, with the containment going both ways because θ^{-1} is a relocation too; adversaries go to adversaries and simulators to simulators because the fields those definitions read are the ones θ fixes, and the dummy is fixed outright, its core being pid-blind. (ii) is then immediate from Lemma 7.9 applied to each world,

the returned simulator being $\theta\mathcal{S}$, which depends on $\theta\mathcal{A}$ alone. (iii) is the concrete form: θ is a bijection of each of the three budgeted classes onto the class the conclusion quantifies over, and reindexing a function along bijections changes neither a supremum nor an infimum. Session-uniformity is preserved throughout, which is checked once rather than in each part. For (i), Definition 4.10 asks two things of $\mathcal{Z}$. The *par* clause is θ-invariant outright: $\theta\mathcal{Z}.\mathit{par} = \theta(\mathcal{Z}.\mathit{par})$ contains $\theta(\langle\pi\rangle \cup \langle\varphi\rangle) = \langle\theta\pi\rangle \cup \langle\theta\varphi\rangle$ exactly when $\mathcal{Z}.\mathit{par}$ contains $\langle\pi\rangle \cup \langle\varphi\rangle$, name closures being invariant by (iv) of Proposition 7.5. The hosting clause transports by injectivity, $\theta\mathcal{Z}$ hosting at the images of what $\mathcal{Z}$ hosts and $\mathrm{IDs}(\theta\pi) \cup \mathrm{IDs}(\theta\varphi)$ being the image of $\mathrm{IDs}(\pi) \cup \mathrm{IDs}(\varphi)$ by (i) of that proposition. The containment goes both ways, θ^{-1} being a relocation by (vi). For the slot, an adversary occupies $\mathrm{IDs}(\mathcal{A})$, whose identities carry the process id (iii) of Definition 7.1 fixes and party components θ leaves alone, and it carries $\mathcal{A}$'s wrapper, which is pid-blind; so θ relabels the core alone and $\theta\mathcal{A}$ is an adversary of the same shape. Definition 4.7 asks $\mathbf{N} \supseteq \mathbf{Std} \cup \mathbf{Z}$, a set θ fixes by (ii), so simulators go to simulators. The dummy's core is pid-blind — line 3 places the call its payload names — so $\theta\mathcal{D} = \mathcal{D}$. Nothing of the kind is claimed of the $\mathcal{A}$ of Section 3.4, whose core Definition 4.6 leaves arbitrary, and nothing below needs it.

For (ii), fix $\mathcal{A} \in \mathbb{A}$ and let $\mathcal{S} \in \mathbb{S}$ be the simulator the hypothesis supplies for it. Given $\theta\mathcal{A} \in \theta\mathbb{A}$, return $\theta\mathcal{S} \in \theta\mathbb{S}$; it depends on $\theta\mathcal{A}$ alone, so the quantifier order of Definition 4.11 is met. For any $\theta\mathcal{Z} \in \theta\mathbb{Z}$, Lemma 7.9 applied to each world gives

$$\mathrm{ADV}^{\mathrm{UC}}_{\theta\pi,\theta\varphi}\big(\theta\mathcal{Z}, \theta\mathcal{A}, \theta\mathcal{S}\big) \;=\; \mathrm{ADV}^{\mathrm{UC}}_{\pi,\varphi}\big(\mathcal{Z}, \mathcal{A}, \mathcal{S}\big) \;\leqslant\; \varepsilon,$$

the two executions on the left being the θ-images of the two on the right. By (i) the conclusion is a legal statement of emulation, $\theta\mathbb{Z} \subseteq \mathbb{Z}_{\theta\pi,\theta\varphi}$.

Session-uniformity is preserved throughout: θ fixes $\mathbf{A}$ and $\mathbf{Z}$ by (ii) of Proposition 7.5 and GLOBAL by its (iii), so $\theta\mathcal{F}$ is session-uniform exactly when $\mathcal{F}$ is, and the same of a system and of an environment memberwise. Every class below is therefore closed under θ in this respect as well.

For (iii), a machine and its image take the same steps and answer the same calls, the code being identity-atomic and its control flow therefore the same on both (Definition 7.4); a member of a budgeted system and its image carry the same budget for the same reason. So θ is a bijection of each of the three classes onto the class the conclusion

quantifies over: $\mathbb{A}(\mathbf{a})$ and $\mathbb{S}(\mathbf{s})$ onto themselves, and $\mathbb{Z}_{\pi,\varphi}(\mathbf{z})$ onto $\mathbb{Z}_{\theta\pi,\theta\varphi}(\mathbf{z})$ by (i), each with inverse induced by θ^{-1}. Lemma 7.9 makes the advantage of Definition 4.5 equal at corresponding triples, so the two nested extrema of Definition 4.31 are the same three operators applied to one function reindexed along bijections, and reindexing changes neither a supremum nor an infimum. No finiteness is needed here; what the finiteness of Definition 4.31 settles is that the extrema are *attained*, which is a separate matter and holds on both sides alike. □

Emulation is thus a property of a system up to relocation, and a proof may fix the session and the instance at the outset. To say so for a family of instances, one needs to know when two lists of process ids are related by a relocation at all, and the answer is three conditions.

Proposition 7.11 (When one tuple relocates to another)**.** *Let* $\vec{q} = (q_1, \ldots, q_m)$ *and* $\vec{q}\,' = (q'_1, \ldots, q'_m)$ *be tuples of process ids of standard name. There is a relocation* θ *with* $\theta(q_j) = q'_j$ *for every* j *if and only if, for all* j *and* k*,*

$$q_j.F = q'_j.F, \qquad (q_j.s = q_k.s \iff q'_j.s = q'_k.s),$$
$$(q_j = q_k \iff q'_j = q'_k).$$

In particular any two process ids of one name are related by a relocation.

Proof. Necessity is the three conditions of Definition 7.1 read off in order. Sufficiency is a construction: build $\hat{\theta}$ from the session assignment, which the second condition makes a well-defined bijection between the sessions occurring on the two sides, and extend it to a permutation because what is left over is of one size on each side; then, at each name and session, build $\theta_{F,s}$ from the instance assignment, which the third and fourth conditions make injective and hence again extensible. Taking the identity everywhere else assembles a relocation carrying $\vec{q}$ to $\vec{q}\,'$, and (iii) holds because the reserved ids were never touched. Necessity is the conditions of Definition 7.1 read off in order: names by (i), sessions shared or not by (ii), and equality by θ being injective.

For sufficiency, let S and S' be the finitely many sessions occurring in $\vec{q}$ and in $\vec{q}\,'$. The second condition makes $q_j.s \mapsto q'_j.s$ a well-defined bijection $S \to S'$ — onto, every $q'_j.s$ being an image — so what is left over on the two sides is of one size, finite or infinite alike, and it

extends to a permutation $\hat{\theta}$ of session ids. Now fix a name F and a session $s \in S$ and consider the indices j with $q_j.F = F$ and $q_j.s = s$. Their instance assignment $q_j.i \mapsto q'_j.i$ is well defined and injective: if $q_j.i = q_k.i$ then $q_j = q_k$, so $q'_j = q'_k$ by the fourth condition and the instances agree; and if $q_j.i \neq q_k.i$ then $q_j \neq q_k$, so $q'_j \neq q'_k$ while their names and sessions agree, and the instances differ. Each is again a bijection between the instance numbers occurring on the two sides, so it extends to a permutation $\theta_{F,s}$ as $\hat{\theta}$ did; take the identity at every other name and session, and let θ act as $(F, s, i) \mapsto (F, \hat{\theta}(s), \theta_{F,s}(i))$ on the standard names and as the identity on the reserved ones. Then θ meets Definition 7.1, (iii) because the reserved ids are fixed outright, and it carries $\vec{q}$ to $\vec{q}\,'$. For the last claim, a single pair q, q' of one name meets the three conditions vacuously. □

Corollary 7.12 (One instance suffices). *Call a pair of maps $\vec{q} \mapsto \pi(\vec{q})$ and $\vec{q} \mapsto \varphi(\vec{q})$ an* instance family *if $\pi(\theta\vec{q}) = \theta\big(\pi(\vec{q})\big)$ and $\varphi(\theta\vec{q}) = \theta\big(\varphi(\vec{q})\big)$ for every relocation θ, where $\vec{q}$ lists the process ids occupied by $\pi(\vec{q})$ and $\varphi(\vec{q})$ together — a protocol, its private subroutines and the specification it is measured against indexed by one tuple and moving with it. Suppose $\pi(\vec{q})$ and $\varphi(\vec{q})$ are session-uniform and that $\pi(\vec{q})$ UC-emulates $\varphi(\vec{q})$ within ε against $\big(\mathbb{A}(\mathbf{a}), \mathbb{S}(\mathbf{s}), \mathbb{Z}_{\pi(\vec{q}),\varphi(\vec{q})}(\mathbf{z})\big)$ cut down to the session-uniform environments, for a single $\vec{q}$. Then at every $\vec{q}\,'$ meeting the three conditions of Proposition 7.11, $\pi(\vec{q}\,')$ UC-emulates $\varphi(\vec{q}\,')$ within the same ε against $\mathbb{A}(\mathbf{a})$, $\mathbb{S}(\mathbf{s})$ and the same cut-down class for that pair. In particular, for a family occupying one process id, emulation at one id gives emulation at every id of that name.*

Proof. Proposition 7.11 supplies a relocation carrying $\vec{q}$ to $\vec{q}\,'$, equivariance turns it into one carrying the pair of systems to the pair at $\vec{q}\,'$, and Proposition 7.10 transports the statement — session-uniformity of the relocated pair coming from that of the original, as the proof of that proposition records. The last claim is the final clause of Proposition 7.11: a one-id tuple meets the three conditions against any other id of its name. □

Indexing by the whole tuple is what makes the condition satisfiable, and the point is worth recording, since indexing by the principal process id alone — the first thing one writes — does not. A relocation may fix a given q and still move a private member sitting at another id, so

equivariance in q alone would force every member of $\pi(q)$ to have an id fixed by the whole stabiliser of q; and the only such ids are q itself and the reserved three, every other being movable by some relocation that fixes q. Equivariance in q alone therefore admits only systems with no private members at all, which is not the case anyone wants.

Written for a pair of systems instead, and asking what else has to move, the statement splits in two. A new *instance* of a name can be taken up by itself; a new *session* cannot, and the reason is condition (ii) of Definition 7.1: a relocation permutes sessions as blocks, so nothing can carry one member out of session s while leaving another behind in it.

Corollary 7.13 (Moving one instance, moving one session)**.** *Let π and φ be session-uniform systems each carrying a member at one and the same process id q_0, and let q be a process id with $q.F = q_0.F$ occupied by neither system. Suppose π UC-emulates φ within ε against $\big(\mathbb{A}(\mathbf{a}), \mathbb{S}(\mathbf{s}), \mathbb{Z}_{\pi,\varphi}(\mathbf{z})\big)$. In each case below the conclusion is that the moved pair UC-emulates within the same ε against $\mathbb{A}(\mathbf{a})$, $\mathbb{S}(\mathbf{s})$ and the largest admissible class of that pair at $\mathbf{z}$.*

(i) If $q.s = q_0.s$, the transposition $\theta := (q_0\ q)$ of instance numbers at the name $q_0.F$ in that session is a relocation, and it fixes the process id of every other member of either system. So $\theta\pi$ and $\theta\varphi$ are π and φ with the instance at q_0 carried to q and nothing else disturbed — save that a member naming the moved instance in a field, by pinning it among its callers in the manner of Remark 6.5 or otherwise, has that mention carried along with it.

(ii) If $q.s \neq q_0.s$, no relocation carries q_0 to q while fixing a process id of standard name in session $q_0.s$ — so no member of a system lying there is fixed, and the block moves entire. One carrying q_0 to q exists; and if in addition neither system has a member in session $q.s$, one may be chosen that carries each member of π and of φ lying in session $q_0.s$ into session $q.s$ and fixes the process id of every member lying elsewhere. Without that further hypothesis the members already in session $q.s$ move too — by the same argument applied at q, no relocation carrying q_0 to q can fix one of them — so the conclusion is then about a pair with two blocks moved rather than one.

Proof. Both parts are Proposition 7.10 once a relocation has been exhibited, so the work is exhibiting one. For (i) the transposition of two ids

of a common session does it, and the check is that it moves nothing else occupied. For (ii) the negative half comes first — a relocation carrying q_0 to an id of a different session moves *every* standard id of that session, by condition (ii) of Definition 7.1 — and the positive half then constructs the relocation that moves session $q_0.s$ and fixes every third session. The last sentence applies the negative half to θ^{-1}, which is a relocation by (vi) of Proposition 7.5. Both parts are Proposition 7.10 once the relocation is exhibited, its part (iii) supplying the classes. For (i), the transposition has $\hat{\theta} = \mathrm{ID}$, so conditions (i) and (ii) of Definition 7.1 hold, and (iii) does because $q_0.F$ is the name of a member of a system and so none of A, Z, C. It moves only the two ids q_0 and q, and q is occupied by neither system, so every other occupied id is fixed; what is not fixed is a field naming q_0, which Definition 7.3 relabels.

For (ii), suppose $\theta(q_0) = q$ and let p be any process id of standard name with $p.s = q_0.s$; by condition (ii) of Definition 7.1, $\theta(p).s = \theta(q_0).s = q.s \neq p.s$, so $\theta(p) \neq p$. For existence, take $\hat{\theta}$ the transposition of the two sessions and instance permutations carrying $q_0.i$ to $q.i$ at the name $q_0.F$ in session $q_0.s$, the identity elsewhere. That θ carries session $q_0.s$ into session $q.s$ and fixes every id of a third session; if no member lies in session $q.s$, the members it moves are exactly those of session $q_0.s$. For the final sentence, let θ carry q_0 to q and let m be a member lying in session $q.s$. Then θ^{-1} is a relocation by (vi) of Proposition 7.5 and carries q to q_0 with $q_0.s \neq q.s$, so the negative claim applied to it moves every standard id of session $q.s$, m among them; and $\theta(m) = m$ would give $\theta^{-1}(m) = m$, so θ moves m too. □

Sessions and instances are not the only components a permutation can move, and the third is what settles a question Chapter 5 raises: a game exercises $\mathcal{F}$ under a claim carrying the game's own name (line 5), so $\mathcal{F}.\mathbf{N}$ must name the game, while the $\mathcal{F}$ that ships inside a protocol admits that protocol instead. The two differ in one field, and it is the field GUARD reads.

Definition 7.14 (Renaming)**.** A *renaming* is a permutation ν of the process ids of the form

$$\nu(F, s, i) \;=\; \big(\bar{\nu}(F),\; s,\; i\big)$$

for a permutation $\bar{\nu}$ of the names fixing A, Z and C. It acts on identities, fields, code and declared sets as Definition 7.3 prescribes,

and in addition replaces every name occurring outright in code by its image.

The extra clause is needed and was not before. Definition 7.3 substitutes process ids, which for a relocation is enough, names being what a relocation keeps; a renaming moves them, so a test against a bare name — $\mathrm{id}'.\mathrm{F} \neq \mathrm{A}$ in MEDIATE, or the $\mathrm{id}'.\mathrm{F} = \mathrm{Z}$ the dummy of Section 4.4 dispatches on — has to travel too. For the machines of this paper it travels nowhere: the only names they mention outright are the three $\bar{\nu}$ fixes, so a renaming changes no line of code at all, only fields.

Proposition 7.15 (Renaming). *Let ν be a renaming. Then*

(i) $\mathrm{IDS}(\nu\mathcal{F}) = \nu(\mathrm{IDS}(\mathcal{F}))$; $\nu(\mathbf{Std}) = \mathbf{Std}$, $\nu(\mathbf{A}) = \mathbf{A}$ *and* $\nu(\mathbf{Z}) = \mathbf{Z}$; $\mathrm{GLOBAL}(\nu\mathcal{F}) \iff \mathrm{GLOBAL}(\mathcal{F})$; $\nu\mathcal{F}$ *is session-uniform exactly when* $\mathcal{F}$ *is;* $\langle\nu\pi\rangle = \nu(\langle\pi\rangle)$; $\nu\pi$ *is a system with* $\mathrm{WF}(\pi) \iff \mathrm{WF}(\nu\pi)$; *and* ν^{-1} *is a renaming;*

(ii) for every system π, every occupant $\mathcal{X}$ of the adversary slot and every environment $\mathcal{Z}$ hosting at no identity of $\mathrm{IDS}(\pi)$ *— with no session-uniformity asked of any of them —*

$$\mathrm{EXEC}(\nu\pi, \nu\mathcal{Z}, \nu\mathcal{X}, \mathcal{C}) \quad \textit{and} \quad \mathrm{EXEC}(\pi, \mathcal{Z}, \mathcal{X}, \mathcal{C})$$

are identically distributed; and

(iii) $\mathbb{Z}_{\nu\pi,\nu\varphi} = \nu(\mathbb{Z}_{\pi,\varphi})$; *if π UC-emulates φ within ε against $(\mathbb{A}, \mathbb{S}, \mathbb{Z})$ then $\nu\pi$ UC-emulates $\nu\varphi$ within ε against $(\nu\mathbb{A}, \nu\mathbb{S}, \nu\mathbb{Z})$; and the concrete advantages of Definition 4.31 agree at every budget.*

Proof. All three parts are the corresponding relocation proofs re-run, and the overview is what changes. Names are now *permuted* rather than carried onto themselves, which is the point of the definition; the reserved classes and **Std** stay fixed, and with them GLOBAL and session-uniformity, while a name closure is carried rather than fixed. In (ii) the one hard paragraph of Lemma 7.9 — the session conjunct — becomes trivial, a renaming moving no session, which is why no session-uniformity hypothesis appears here at all. In (iii) what is lost is the reading, not the result: the environment class of the conclusion corresponds to the hypothesis's rather than being it. (i) is the proof of Proposition 7.5 with two changes of bookkeeping. Name classes are

now permuted rather than carried onto themselves, so an **N** declared by names becomes the one declared by the images — which is the whole point of the definition — while **Std**, being the union of every standard class, and **A** and **Z**, being single reserved classes, are fixed, and with them GLOBAL, Definition 7.2 and the reserved ids of (iii) of Definition 7.1. And a name closure is carried rather than fixed, $\langle\rangle$ being specified by the names a system uses. For WF the session conjunct is untouched, ν moving no session, and the name match holds on one side exactly when on the other.

(ii) is the proof of Lemma 7.9 verbatim but for its one hard paragraph, which becomes trivial: line 3's first disjunct is literally unchanged, ν leaving every session alone, and its second is preserved by (i). Nothing needs excluding, which is why no session-uniformity appears here. Every other check reads a name for equality or a membership in a set that travels with the code, and Definition 7.4 together with the extra clause of Definition 7.14 makes the substitution compute the relabelled behaviour as before.

(iii) is the proof of Proposition 7.10, which used of the closures only that $\nu(\langle\pi\rangle \cup \langle\varphi\rangle) = \langle\nu\pi\rangle \cup \langle\nu\varphi\rangle$ and never that they stay put. What is lost is the reading recorded after Proposition 7.5: the environment class of the conclusion corresponds to the one of the hypothesis rather than being constrained identically, since *par* must now name the images. □

Corollary 7.16 (The caller's name is immaterial)**.** *Let ν be a renaming and let $\gamma = \{\mathcal{G}\mathfrak{m}\} \cup \sigma$ be a game system for $\mathcal{F}$ (Definition 5.1). Then $\nu\gamma = \{\nu\mathcal{G}\mathfrak{m}\} \cup \nu\sigma$ is a game system, $\nu\mathcal{G}\mathfrak{m}$ is a game for $\nu\mathcal{F}$, and for every finalization* FIN *of $\mathcal{G}\mathfrak{m}$*

$$\mathrm{ADV}^{\mathrm{FIN}}_{\nu\mathcal{G}\mathfrak{m},\nu\sigma,\nu\mathcal{Z},\nu\mathcal{X}} \;=\; \mathrm{ADV}^{\mathrm{FIN}}_{\mathcal{G}\mathfrak{m},\sigma,\mathcal{Z},\mathcal{X}}$$

for every $\mathcal{Z} \in \mathbb{Z}_{\gamma,\gamma}$ and every occupant $\mathcal{X}$ of the adversary slot. Since $\nu\mathcal{D} = \mathcal{D}$, the abbreviated form reads

$$\mathrm{ADV}^{\mathrm{FIN}}_{\nu\mathcal{G}\mathfrak{m},\nu\sigma,\nu\mathcal{Z}} \;=\; \mathrm{ADV}^{\mathrm{FIN}}_{\mathcal{G}\mathfrak{m},\sigma,\mathcal{Z}}.$$

So σ has the property FIN *within ε against a class of environments exactly when $\nu\sigma$ has it within ε against the image of that class.*

Proof. That $\nu\gamma$ is a system with $\mathrm{WF}(\nu\gamma)$ is (i). For $\nu\mathcal{G}\mathfrak{m}$ being a game, check the four fields of Definition 5.1: its process id is $(\bar{\nu}(\mathcal{G}\mathfrak{m}.\mathrm{F}), 0, 0)$, ν

moving neither session nor instance; its $\mathbf{P}$ is untouched and $\nu\mathcal{F}.\mathbf{P} = \mathcal{F}.\mathbf{P}$, so it is $\nu\mathcal{F}.\mathbf{P} \cup \{Z\}$ as required; its $\mathbf{N}$ is $\nu(\mathbf{Z}) = \mathbf{Z}$ by (i); its entry is $(\nu\mathcal{F}, \text{SERVES}_{\mathcal{G}\mathfrak{m}})$, and $\text{SERVES}_{\mathcal{G}\mathfrak{m}}$ is equivariant, reading $\mathbf{P}$ and a membership of $\mathbf{N}$. Line 8 still serves every finalization at the root, party ids being no part of a process id, and a renaming moves no interface name, so FIN names the same one on both sides. For the advantage, WIN_{FIN} is the event that the interface FULLFIN at $(\mathcal{G}\mathfrak{m}.pid, Z)$ returns 1, and ν carries that identity to $(\nu\mathcal{G}\mathfrak{m}.pid, Z)$; by (ii) the two executions are identically distributed under the correspondence, so the event has the same probability on both sides and Definition 5.8 gives the same number. That $\nu\mathcal{Z}$ is admissible for $\nu\gamma$ is (iii). For the abbreviated form, $\nu\mathcal{D} = \mathcal{D}$: the dummy's fields are $pid := A$ and sets of ids that (i) fixes, and its core dispatches on $\mathtt{id}'.\mathsf{F} = Z$ and places the call its payload names, so no name it mentions moves. □

Read at the transposition of $\mathcal{G}\mathfrak{m}.\mathsf{F}$ with a name γ does not use, and with $\mathcal{G}\mathfrak{m}.\mathsf{F}$ carried by $\mathcal{G}\mathfrak{m}$ alone, this says what one wants of Definition 5.1. The game may be given any name and $\mathcal{F}.\mathbf{N}$ renamed to match; no other field moves, and by Definition 7.14 no line of code does either. A property is therefore not tied to what its game is called, and the $\mathcal{F}$ whose property one proves may admit whatever caller name the deployment will use.

What no renaming gives is a different caller *programme*. It carries a game to a game and a protocol to a protocol, never a game to the protocol that will use $\mathcal{F}$ in earnest — and a protocol that hands $\mathcal{F}$'s answers to the adversary satisfies no property of $\mathcal{F}$, so nothing could. Proposition 7.7 narrows the gap from the other side, $\mathbf{N}$ being read nowhere but inside $\text{GUARD}_{\mathcal{F}}$: a property is a statement about $\mathcal{F}$'s code and the access control around it, and about nothing else that field could carry. Crossing the rest is what emulation is for, and Proposition 5.10 is the crossing. Relocations and renamings compose, each being an isomorphism of executions, so a system may be moved in name, session and instance at once.

Remark 7.17 (Session 0 is distinguished)**.** Session-uniformity is not an artefact of the proofs, and the functionality it excludes is a real one. Line 3 compares the *caller's* session, and $\mathcal{A}$ and $\mathcal{Z}$ sit in session 0, so a functionality that is *local* in the sense of Chapter 1 yet admits $\mathbf{A}$ is addressable from the adversary slot exactly when

its own session is 0: with $\mathcal{F}.s \neq 0$ the conjunct $\mathrm{id}'.s = \mathrm{id}.s$ fails and $\textsc{global}(\mathcal{F})$ does not save it. The difference is observable — it is how $\mathcal{D}$ carries out an instruction of Section 4.4 at a corrupt party — so for such a pair no relocation moving session 0 is an isomorphism, and Definition 7.2 is necessary rather than convenient. It is also the least one can exclude: for every other callee the supplied machines are waived by GLOBAL or refused at line 2, so no smaller condition would do and no larger one is needed.

The shape it excludes is not exotic, and one place the paper itself reaches for it should be said plainly. Remark 6.5 offers, as a way of closing the hosting route to an *interface* member, that such a member admit only $\mathbf{Z} \cup \mathbf{A}$ — the environment and the adversary and no one else. A member of that shape is not global, **Std** lying outside its **N**, so it is not session-uniform. That is no defect of Definition 7.2 but the artefact showing through at the place it costs most: line 3 lets the environment reach such a member in session 0 and in no other, so a protocol whose interface members admit exactly $\mathbf{Z} \cup \mathbf{A}$ is drivable only in session 0, whatever relocation has to say about it. The two routes an admissible environment has into a protocol divide accordingly. It may claim a fresh name, which line 2 admits only at a member admitting all of **Std** — global, hence shared across sessions, hence not one run of one protocol. Or it may claim its own Z-identity, which line 2 admits at a member admitting **Z** — and then line 3 pins that member to session 0. Neither route reaches a session-local protocol interface in a session of its own choosing.

That the environment can claim its own Z-identity at all is worth recalling, since it is what makes the second route available: the name Z is used by no system, so Z-identities lie outside $\langle\pi\rangle \cup \langle\varphi\rangle$ and Definition 4.10 leaves them claimable. A member whose **N** is only its parent's name class is reachable by neither route, and there the artefact shows through the adversary's slot alone.

Reading line 3 as

$$\big(\mathrm{id}'.s = \mathrm{id}.s \;\vee\; \textsc{global}(\mathcal{F}) \;\vee\; \mathrm{id}'.\mathrm{F} \in \{A, Z\}\big)$$

would dispense with the condition. Session 0 would then confer nothing, a local functionality admitting the adversary would be reachable from the slot in every session alike, and Definition 7.2 could be dropped from every statement above: the results would

hold of all systems rather than the session-uniform ones. Nothing else in the chapter would change — condition (iii) stays, the reserved ids being named outright in code whatever the session rule is, and the tuple conditions of Proposition 7.11 are already free of session 0. The widening is small and in the spirit of the rest: the two machines it names are global by GLOBAL's second disjunct wherever GLOBAL is tested of them, the corruption gate of line 3 still confines $\mathcal{A}$ to corrupt parties, and $\mathcal{Z}$ stays out by **N** and by its own *par*. What it would buy is more than relocation: it is what would let a protocol be local to its session and driven by the environment at the same time, which is the pairing the two routes above deny and which Remark 6.5's suggestion silently assumes. We keep the narrower rule, which is Canetti's [7], and record both the option and its price here.

Remark 7.18 (Relocation is not a many-instance theorem)**.** Proposition 7.10 moves one copy and charges nothing. It does not follow, and is not true, that n copies at n indices cost ε rather than $n\varepsilon$: the executions of $\pi(\vec{q}_1), \ldots, \pi(\vec{q}_n)$ and of $\varphi(\vec{q}_1), \ldots, \varphi(\vec{q}_n)$ are not each other's relabellings — no relocation carries a system to n copies of itself — and the only route between them is the hybrid chain of Theorem 4.24, which adds. What the two results give together is the statement one wants from a single proof: prove $\pi(\vec{q})$ emulates $\varphi(\vec{q})$ at one $\vec{q}$; Corollary 7.12 gives it at every other within ε; and Theorem 4.24, whose disjointness hypotheses come to the $\vec{q}_k$ being pairwise disjoint and clear of the surrounding shell, replaces all n at once within $n\varepsilon$, at the budgets Chapter 8 charges. Relocation is also functorial in the constructions of Section 4.1 and Chapter 6, each of which reads only names, ids and identity sets: $\theta\big(\rho[\pi/\varphi]\big) = (\theta\rho)[\theta\pi/\theta\varphi]$ since an injective action commutes with the set difference, $\theta\overline{\pi} = \overline{\theta\pi}$ — the right-hand side read, as Definition 6.2 requires, as the blocked system of the pair $(\theta\pi, \theta\varphi)$ — since a blocker copies the two fields θ relabels and $I_{\theta\varphi} \setminus I_{\theta\pi} = \theta(I_\varphi \setminus I_\pi)$, and $\theta\big(\mathcal{Z}^{\rho\setminus\varphi}\big) = (\theta\mathcal{Z})^{\theta\rho\setminus\theta\varphi}$ since Definition 7.3 relabels a bundle's declared outward set, whose reserved half $\{\mathrm{id} : \mathrm{id}.F \in \{A, C\}\}$ is fixed by (ii) of Proposition 7.5. So a relocated proof of composition is the composition of the relocated proof.

CHAPTER 8

Concrete Costs

Two threads of concrete cost sit outside the main line and are collected here: the cost of a system in itself, read off its members, and the cost the composition results of Section 4.5 charge, arithmetic on the budgets of Definition 4.31.

Budgeted member by member, a system also has a cost as a whole, and it is a functionality's cost read one level up.

Definition 8.1 (Cost of a system)**.** An *invocation* of a system π is a call reaching one of its members from outside π; π services it by a cascade of internal activations before control returns outside. The *cost* of π is the pair (T_π, R_π): the *per-invocation time* T_π, the largest total number of steps in one such cascade — summed over the activations it triggers, one machine running at a time — and the *invocation count* R_π, the number of invocations π answers. So π runs in at most $T_\pi R_\pi$ steps.

Proposition 8.2 (System cost from member costs)**.** *Let* $\pi = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$ *with each* $\mathcal{F}_i$ *of cost* (t_i, r_i)*, and let* k_i *bound the activations of* $\mathcal{F}_i$ *within one invocation of* π*. Then*

$$T_\pi \leqslant \sum_i k_i t_i, \qquad R_\pi \leqslant \sum_i r_i,$$

and π *runs in at most* $\sum_i t_i r_i$ *steps. If every invocation activates each member at most once —* π *subroutine-respecting,* $k_i \leqslant 1$ *— then* $T_\pi \leqslant \sum_i t_i$*; if moreover each invocation drives all members,* $R_\pi \leqslant \min_i r_i$*.*

Proof. An invocation activates $\mathcal{F}_i$ at most k_i times, each for at most t_i steps, so it spends at most $\sum_i k_i t_i$. The activations of $\mathcal{F}_i$ over the

whole execution number at most r_i, and every invocation is at least one activation, so the invocations number at most $\sum_i r_i$; the total steps are $\sum_i (\text{activations of } \mathcal{F}_i)\, t_i \leqslant \sum_i r_i t_i$. When each invocation drives all members, R_π invocations activate each $\mathcal{F}_i$ exactly R_π times, so $R_\pi \leqslant r_i$ for every i. □

Corollary 8.3 (Cost of a union). *Let π_1 and π_2 be systems with disjoint process ids and costs (T_1, R_1) and (T_2, R_2). If no member of one invokes a member of the other, then*

$$\begin{aligned}\big(T_{\pi_1 \cup \pi_2},\ R_{\pi_1 \cup \pi_2}\big) &= (T_1, R_1) \oplus (T_2, R_2) \\ &= \big(\max(T_1, T_2),\ R_1 + R_2\big).\end{aligned}$$

If instead each invocation of π_1 makes at most k calls into π_2 and the cross-call graph is acyclic, then $T_{\pi_1 \cup \pi_2} \leqslant T_1 + k\,T_2$ from a π_1 entry, and symmetrically, with $R_{\pi_1 \cup \pi_2} \leqslant R_1 + R_2$; with the cross-calls unbounded the summary costs no longer determine the union's.

Proof. With no cross-calls a call from outside $\pi_1 \cup \pi_2$ enters π_1 or π_2 and its cascade stays there, costing at most T_1 or T_2 and belonging to that subsystem's invocation count; so the per-invocation time is $\max(T_1, T_2)$ and the counts add. A cross-call from π_1 into π_2 is internal to the union and extends the cascade by one invocation of π_2, at most T_2 steps; at most k of them, acyclically, give $T_1 + kT_2$, while a call once external to π_2 but placed from π_1 is no longer counted, so the invocations still number at most $R_1 + R_2$. □

The subroutine-respecting proviso is not idle: without it one call can be driven around the system without end, and T_π degenerates towards the whole of $\sum_i t_i r_i$ — the same discipline that keeps a private member's callers pinned in Remark 6.5. This system cost is dual to the $\oplus$ of Definition 4.30: hosting a system as a bundle, one member per activation, combines its members by $\oplus$ into $(\max_i t_i,\ \sum_i r_i)$, the cost $\mathbf{c}_{\rho \setminus \varphi}$ pays in the composition costs below; calling it as a subsystem sums the times over a cascade and counts external calls, $(\sum_i t_i,\ \min_i r_i)$ in the synchronous case. Both bound the same total $\sum_i t_i r_i$, the second more tightly when every invocation is a full pass. And $\oplus$ recurs at the system level itself: by Corollary 8.3 non-interacting subsystems compose by it, one operator at both scales.

When the members of π call one another the per-invocation time turns on the shape of that calling, and one case is clean.

Proposition 8.4 (DAG systems). *Suppose the call graph of $\pi = \{\mathcal{F}_1, \dots, \mathcal{F}_n\}$ is acyclic and every invocation activates each internal member $\mathcal{F}_i$ at most $\max_{j:\,\mathcal{F}_j \to \mathcal{F}_i} t_j$ times — the largest per-invocation time among the members that call it. Then*

$$T_\pi \;\leqslant\; \sum_i \Big(\max_{j:\,\mathcal{F}_j \to \mathcal{F}_i} t_j \Big) t_i \;\leqslant\; \Big(\max_i t_i \Big) \sum_i t_i,$$

a per-invocation time independent of the depth of the graph.

Proof. An invocation entering a source activates each $\mathcal{F}_i$ some a_i times and spends $\sum_i a_i t_i$ steps. A source runs once and each internal $\mathcal{F}_i$ at most $\max_{j \to i} t_j$ times by hypothesis, so $a_i \leqslant \max_i t_i$ throughout, whence $T_\pi \leqslant \sum_i (\max_{j \to i} t_j)\, t_i \leqslant (\max_i t_i) \sum_i t_i$. □

The hypothesis earns its place, for acyclicity alone does not bound T_π. Writing n_{ji} for the calls $\mathcal{F}_j$ makes to $\mathcal{F}_i$ in one activation, the per-invocation activation counts solve $a_i = \sum_{j \to i} n_{ji}\, a_j$, so a_i counts the weighted source-to-$\mathcal{F}_i$ paths, and a reconvergent graph multiplies them: a chain of N diamonds — $s \to \{A_k, B_k\} \to C_k \to \{A_{k+1}, B_{k+1}\} \to \cdots$ — doubles the count at each level, so a at the foot is 2^N and the per-invocation time is exponential in the depth. Capping each node by the *maximum* over its callers rather than by their sum is exactly what stops the paths accumulating — the $\oplus$-over-summation theme once more, and the DAG counterpart of the subroutine-respecting case $k_i \leqslant 1$ of Proposition 8.2, one activation relaxed to $\max_{j \to i} t_j$. The invocation caps bound $a_i \leqslant r_i$ in any case; the point here is a bound in the times alone, independent of the caps and of the depth.

That last remark is the whole of the general statement: the caps alone already close the cost class, with no condition on the call structure.

Corollary 8.5 (Cost composition). *A system $\pi = \{\mathcal{F}_1, \dots, \mathcal{F}_n\}$ whose members have costs (t_i, r_i) runs in at most $\sum_i t_i r_i$ steps over at most $\sum_i r_i$ invocations, so $T_\pi \leqslant \sum_i t_i r_i$ and $R_\pi \leqslant \sum_i r_i$ whatever its call structure. Hence if each t_i and r_i is polynomial in a parameter λ and n is polynomially bounded, (T_π, R_π) is polynomial: the polynomially bounded*

systems are closed under formation.

Proof. The step and invocation bounds are Proposition 8.2, and one invocation's cascade is part of the whole execution, so $T_\pi \leqslant \sum_i t_i r_i$ too. A sum of polynomially many polynomials is a polynomial. □

No condition on the call graph is needed here, unlike the depth-free *time* bound of Proposition 8.4, because the caps carry it. A reconvergent cascade that would activate a member 2^N times can do so only if that member answers 2^N calls, so $r_i \geqslant 2^N$ — a member already outside the polynomial class; within the class the cap refuses the excess and truncates the cascade at $\sum_i r_i$ activations. This is the efficiency counterpart of Theorem 4.20: where composition preserves *security* — $\rho[\pi/\varphi]$ emulates ρ once π emulates φ — cost composition preserves *efficiency*, so a protocol assembled from polynomial-time functionalities is itself polynomial-time. Together the two are what the concrete apparatus is for: polynomial-time UC-secure protocols compose into polynomial-time UC-secure protocols, which is what lets the asymptotic reading below be a corollary rather than a theory of its own.

The same closure survives the operation the composition theorem runs.

Proposition 8.6 (Replacement preserves polynomial boundedness)**.** *Let ρ and π be systems whose members are polynomial and polynomially many, and let the replacement of φ by π in ρ be well-formed. Then $\rho[\pi/\varphi]$ is polynomially bounded.*

Proof. The members of $\rho[\pi/\varphi] = (\rho \setminus \varphi) \cup \pi$ (Definition 4.1) are those of ρ outside φ, each polynomial as a member of ρ, together with those of π, again polynomial; they number at most $|\rho| + |\pi|$, polynomially many. Corollary 8.5 applies. □

So the system that runs after a replacement is efficient whenever the specification ρ and the realization π are — Theorem 4.20 read for cost rather than security, and on the same operation.

Composition, numerically. In Theorem 4.20 the adversaries and the simulators carry over unchanged, so their budgets do. The environments are the one place something is computed, and what absorption costs is the cost of hosting. Systems are budgeted member by member,

so the cost can be looked up: write $\mathbf{c}_{\rho\setminus\varphi}$ for the $\oplus$-combination of the budgets of the members of $\rho \setminus \varphi$ — the largest of their per-invocation times, the sum of their invocation counts, the hosted-bundle cost dual to the called-subsystem cost of Proposition 8.2 — which bounds them in any execution, whoever is driving them. The absorbed environment of Definition 4.15 runs $\mathcal{Z}$ and those functionalities and nothing else, so

$$\mathcal{Z} \in \mathbb{Z}_{\rho[\pi/\varphi],\,\rho}(\mathbf{z}) \qquad \Longrightarrow \qquad \mathcal{Z}^{\rho\setminus\varphi} \in \mathbb{Z}_{\pi,\varphi}\big(\mathbf{z} \oplus \mathbf{c}_{\rho\setminus\varphi}\big),$$

the containment in $\mathbb{Z}_{\pi,\varphi}$ being Proposition 4.16. Feeding this into the theorem, the conclusion is

$$\mathrm{ADV}^{\mathrm{UC}}_{\rho[\pi/\varphi],\,\rho}[\mathbf{a},\mathbf{s},\mathbf{z}] \;\leqslant\; \mathrm{ADV}^{\mathrm{UC}}_{\pi,\varphi}\big[\mathbf{a},\ \mathbf{s},\ \mathbf{z}\oplus\mathbf{c}_{\rho\setminus\varphi}\big].$$

Composition is free in the adversary and in the simulator, and costs the surrounding system once, on the environment side.

Transitivity, numerically. Proposition 4.12 asks that the first step's simulator class be the second step's adversary class, which becomes an inequality between budgets: $\mathbb{S}(\mathbf{s}) \subseteq \mathbb{A}(\mathbf{a}')$ holds as soon as $\mathbf{s} \leqslant \mathbf{a}'$. A simulator one is willing to build must be an adversary the next step is willing to face. The proposition also asks for one class admissible for both steps, and that hypothesis must be carried here: for $\mathbb{Z} \subseteq \mathbb{Z}_{\pi,\varphi} \cap \mathbb{Z}_{\varphi,\psi}$, which by the proof of Proposition 4.12 lies in $\mathbb{Z}_{\pi,\psi}$ as well,

$$\mathrm{ADV}^{\mathrm{UC}}_{\pi,\psi}[\mathbf{a},\mathbf{s}',\mathbb{Z}] \;\leqslant\; \mathrm{ADV}^{\mathrm{UC}}_{\pi,\varphi}[\mathbf{a},\mathbf{s},\mathbb{Z}] \;+\; \mathrm{ADV}^{\mathrm{UC}}_{\varphi,\psi}[\mathbf{s},\mathbf{s}',\mathbb{Z}].$$

Budgets alone will not do here. Written with $\mathbf{z}$ in all three places the brackets would range over $\mathbb{Z}_{\pi,\psi}(\mathbf{z})$, $\mathbb{Z}_{\pi,\varphi}(\mathbf{z})$ and $\mathbb{Z}_{\varphi,\psi}(\mathbf{z})$, and the first is contained in neither of the others — an environment admissible for the outer pair is told nothing about $\mathrm{IDs}(\varphi)$ — so no hypothesis would cover the middle probability that the triangle inequality cancels. The composition display above has no such trouble: absorption maps its one class into the other. Corollary 4.21 is the two displays put together. Along the chain of Theorem 4.24 the same two effects accumulate: the errors add, each hop widens the environment budget by the cost of hosting what it absorbs at that step, and the class relations of Remark 4.25 become the inequalities $\mathbf{s}_{k+1} \leqslant \mathbf{a}_k$.

Nesting, and whether the simulator blows up. Say that an emulation statement has *overhead* f if it holds against $\big(\mathbb{A}(\mathbf{a}),\mathbb{S}(\mathrm{f}(\mathbf{a})),\cdot\big)$ for every

adversary budget **a**: whatever the adversary is given, the simulator one builds costs f of it. In the per-invocation budget the two components behave differently,

$$f(t, r) \;=\; \big(\max(t, \tau),\;\; \gamma r + \delta\big),$$

the query count affine — γ the calls the simulator places for each call of the adversary it runs, δ its own — and the per-invocation time a maximum, τ the simulator's own per-activation work, since one machine runs per activation and the simulator adds only bounded work to each.

Theorem 4.20 does not touch f at all. Its adversary and simulator classes are those of the hypothesis, so a single replacement returns the very machine the hypothesis supplied and the whole price is paid on the environment side. Overheads compose in one place only: where a simulator is re-read as an *adversary*, which is Proposition 4.12. There $\mathcal{S}'$ is built from $\mathcal{S}$, itself built from $\mathcal{A}$, so the composite has overhead $f_2 \circ f_1$. Applying the results k times, the query budget obeys $r_j = \gamma\, r_{j-1} + \delta$, that is

$$r_k \;=\; \gamma^k r_0 \;+\; \delta\,\frac{\gamma^k - 1}{\gamma - 1} \quad (\gamma \neq 1), \qquad r_k \;=\; r_0 + k\delta \quad (\gamma = 1),$$

while the per-invocation time saturates, $t_k = \max(t_0, \tau_1, \ldots, \tau_k)$, bounded by the largest level's own work and constant in depth.

The blow-up therefore lives entirely in the query count, a dichotomy turning on γ alone. A simulator that must place more than one call for some call of the adversary it runs has $\gamma > 1$, and k applications cost γ^k: genuinely exponential, and at $\gamma = 2$ a depth of 20 already costs a factor of 2^{20}, about a million, on the query count. A simulator that relays each call once and adds bounded traffic of its own has $\gamma = 1$, and k applications cost $r_0 + k\delta$: linear, δ paid once per level. This is the property to check when a simulator is built; nothing proved here implies it. It is also why straight-line simulators matter beyond the usual reasons, rewinding being the standard way to acquire $\gamma > 1$: one rewinding simulator anywhere in the tower puts a factor γ on every level below it.

The same formula governs breadth as well as depth. Theorem 4.24 builds $\mathcal{S}_n, \ldots, \mathcal{S}_1$ by handing each simulator to the next step as its adversary, so n parallel instances compose n overheads exactly as an n-deep tower does, and Remark 4.25 read with budgets asks $\mathbf{a}_k \geqslant$

$f_{k+1}(\mathbf{a}_{k+1})$ at every link. Composing n instances of a $\gamma = 1$ protocol costs $n\delta$ extra queries; composing n instances of a $\gamma = 2$ one is hopeless for even moderate n.

Nothing comparable happens on the environment side, or in the per-invocation time. Each level adds the cost of hosting the shell it absorbs, and a tower's shells are disjoint, so a depth-k nesting reaches the innermost statement at budget $\mathbf{z} \oplus \bigoplus_{i \leqslant k} \mathbf{c}_i$ — its invocation count that of running the ambient system once, its per-invocation time the largest shell's, however deep the tower. The errors behave likewise, k applications of Proposition 4.12 giving $\sum_i \varepsilon_i$. Where a security claim degrades exponentially under nesting, the simulator's query overhead $\gamma > 1$ is the only possible source: one machine runs per activation, so neither the per-invocation time nor the environment budget can multiply.

Recovering the asymptotic reading. Nothing above mentions a security parameter. To get the familiar statement, index everything by λ, let the budgets be polynomials in λ, and ask that the advantage be negligible in it; the classes then become the polynomially bounded machines and Definition 4.11 the usual definition of UC emulation. The concrete form is the one the composition results are proved in, and the asymptotic form is a corollary of it, not the other way round — the ordering practice-oriented provable security has long argued for [2].

CHAPTER 9

Responsive Calls

Chapter 15 needs a promise the framework has so far had no way to write down. When SIGN asks the adversary for a signature it must get one back, and get it back *now*: were the adversary free to go off and drive the rest of the execution before answering, the party would be left suspended mid-signature and locality lost — the same immediacy synchronous UC asks of a responsive environment [11]. The text there settles for restricting the simulator class. This chapter says the thing in code instead.

What has to be forbidden. Not that the adversary answer helpfully — it may return anything, and Chapter 12 deals with a bad answer. What has to be forbidden is the *token* leaving: between receiving the call and returning, the responder must place no call at all. That is a restriction on the calls a machine makes, and the paper already has a wrapper for exactly that.

Definition 9.1 (Responsive answering)**.** RESPOND is SILENCE at its maximum: with $\mathcal{I}_{\text{ALL}}$ the set of all identities,

$$\begin{aligned} &\textsc{Respond}\big(\mathtt{id}.\textsc{Op}(\mathtt{in}) \text{ from } \mathtt{id}'\big) \\ :=\ &\textsc{Silence}\big(\mathcal{I}_{\text{ALL}},\ \mathtt{id}.\textsc{Op}(\mathtt{in}) \text{ from } \mathtt{id}'\big). \end{aligned}$$

Every claim lies in $\mathcal{I}_{\text{ALL}}$, so by Chapter 2 every call the wrapped programme issues is refused with REJ *without ever being placed*. The programme may compute as long as it likes and must still return a value; what it cannot do is hand the token on.

Read as a handler. Written out, RESPOND is SILENCE with its last clause deleted. The wrapper SILENCE forwards a call whose claim it permits, so its operation clause is partly a re-raise; RESPOND has no

Wrapper RESPOND
programme id.OP(in) from id′

RESPOND(id.OP(in) from id′):
1: **handle** id.OP(in) from id′ **with**
2: **return** out ↦ out
3: id″.FULLOP(in″) as id‴; k ↦ k(REJ)

re-raise, only a constant. In the vocabulary of Chapter 2 it *discharges* the ambient call effect rather than forwarding it: inside its scope the effect is handled locally and completely, so the wrapped programme is pure with respect to it. It is also the definition of *locality*, the property Chapter 15 wanted: a call on the slot is served within the responder's own activation, the token never leaving it. That is the crisp statement of responsiveness — *a responsive core is one whose call effect does not escape to* EXEC — and the operational reading follows, since EXEC dispatches exactly the calls that do escape. A responsive call is one activation of the responder and no more.

Definition 9.2 (Responsive interface, responsive call)**.** The adversary carries, beside FULLA, a second interface FULLA$^!$ with the same guard and the same core but RESPOND in place of SILENCE:

$$\mathrm{id}.\textsc{FullA}^!(\mathrm{in})\ \text{from}\ \mathrm{id}' : \quad \textbf{require}\ \textsc{Guard}_{\mathcal{A}}(\mathrm{id}, \mathrm{id}');\ \textbf{return}\ \textsc{Respond}\big(\mathrm{id}.\mathrm{A}(\mathrm{in})\ \text{from}\ \mathrm{id}'\big).$$

A *responsive call* is a call on FULLA$^!$. Inside an interface with identity id we write $\mathcal{A}^!(\mathrm{in})$ for $(\mathrm{A}, \mathrm{id}.\mathrm{P}).\textsc{FullA}^!(\mathrm{in})$ as id, as $\mathcal{A}(\mathrm{in})$ abbreviates the ordinary one.

Three things follow, and they are the price.

The responder cannot read the register. The set **C** is reached by calling FULLSTATUS, and that call is refused, so a responsive answer is computed from what the responder already knows. It cannot delegate: no subroutine, no other party, no environment. And it cannot corrupt, CORRUPT being a call like any other. The last deserves stating on its own, because Section 3.5 reads the register twice and the gap between the two reads is exactly where a corruption can slip in.

Proposition 9.3 (A responsive call cannot corrupt)**.** *In any execution, the register's state* **C** *when a responsive call returns is its state when the call was placed.*

Proof. By Section 3.1, **C** changes only on line 5 of FULLCORRUPT, and only when that interface is reached by a call. Between the placing and the return of a responsive call the responder holds the token throughout, and by Definition 9.1 every call it issues is refused before being placed, so FULLCORRUPT is not reached. No other machine holds the token in that interval, so none can reach it either. □

So if a core reaches the adversary only through $\mathcal{A}^!$, the party it serves is corrupt at line 7 exactly when it was corrupt at line 3, and the second mediation hook cannot fire on account of that call. Remark 3.2 describes what happens when it does; responsiveness is the condition under which it does not.

This is what Chapter 15 wanted. Writing $\mathcal{A}^!$ in place of $\mathcal{A}$ in GEN, SIGN and VERIFY makes locality a property of the code rather than a condition on the simulator class, and the wrapper enforces it, so it holds of *every* occupant of the adversary slot — the simulator included, Definition 4.7 giving a simulator the adversary's interfaces, FULLA$^!$ among them. What it does not do is make the answer good: a responsive adversary may still return $\perp$, and SAN repairs that. Responsiveness buys the token back; sanitization buys the value.

Remark 9.4 (Responsiveness and the dummy)**.** The two chapters pull against each other, and it is worth being explicit about where. The dummy of Section 4.4 answers a call from a functionality by relaying it to the environment on line 5 — and that is a call, which RESPOND refuses. So $\mathcal{D}$ cannot serve a responsive call: the relay is refused, and the rule of Section 3.6 delivers its answer as $\perp$ where a real adversary would answer with substance. The regrouping (R1) would therefore fail for a system whose cores use $\mathcal{A}^!$, at the first responsive call — which is why Theorem 4.29 hypothesises systems whose cores place none. Recovering completeness for responsive systems needs a matching notion on the environment's side, so that the relayed question can be answered without the token moving on; we leave that open.

CHAPTER 10

Core Language

Chapters 13 to 21 give the code of this book's functionalities, protocols and games, and that code is meant to be read as a program, not as an illustration of one — a promise that has to be kept by a stated convention rather than by the reader's good will. [5] keep exactly this promise for game-based proofs with a small, strongly typed, terminating language they call $\mathcal{L}$, given by a context-free grammar (their Figure 15) and a short named list of the sugars they allow on top of it. This chapter takes the same two steps here: a closed grammar for what every chapter from here on calls a *core* — an operation's body, already this book's own word for it — and this book's own short list of departures from that grammar, each named once so that every core still to come can be checked against it rather than taken on faith.

Everything before this chapter is exempt, and deliberately so. GUARD, SILENCE and MEDIATE (Chapter 2), EXEC (Section 3.6), the dummy adversary (Section 4.4) and RESPOND (Chapter 9) do not run inside a core: they are what runs a *system* of cores, dispatching calls, evaluating guards, handing the token from one activation to the next. A definition of execution is not itself a program in the language it executes — exactly as [5]'s own account of a game meeting an adversary sits in their informal Section 3, outside the grammar of their Figure 15 — and the one notation that could be mistaken for a core's own, the continuation form $\mathtt{id}''.\textsc{FullOp}(\mathtt{in}'')$ as $\mathtt{id}'''$; $\mathtt{k}$ carrying an explicit, unbound resumption $\mathtt{k}$, belongs entirely to that outer layer: it never once occurs inside an operation's own code from Chapter 13 on. A plainer relative of it does, constantly — an ordinary call to another functionality's operation, resumed under the caller's own identity rather than handed a fresh continuation — and that one is a core-language matter in its own right, below.

The core grammar. A core is a sequence of statements, and every

operation this book writes is exactly one core. Scoped to what this book's cores actually use, rather than to $\mathcal{L}$'s full grammar:

$$
\begin{aligned}
\mathit{core} &::= \varepsilon \mid \mathit{stmt}\,;\,\mathit{core} \\
\mathit{stmt} &::= \mathit{lv} \leftarrow \mathbf{e} \mid \mathit{lv} \leftarrow_{\$} \mathrm{S} \\
&\quad \mid \textbf{if } e \textbf{ then } \mathit{core}\ [\,\textbf{else } \mathit{core}\,] \\
&\quad \mid \textbf{for } x \in \mathrm{S} \textbf{ do } \mathit{core} \\
&\quad \mid \textbf{return } e \\
\mathit{lv} &::= \mathtt{x} \mid \mathtt{x}[e] \\
\mathbf{e} &::= \ldots \mid \mathit{call} \\
\mathit{call} &::= \mathtt{id}''.\textsc{Op}(e_1, \ldots, e_k)\ \textbf{as } \mathtt{id}
\end{aligned}
$$

Figure 5: The core grammar, scoped to this book's own usage. $\mathcal{L}$'s grammar ([5], their Figure 15) with the sub-grammar of expressions left to ordinary mathematical notation, since every expression this book writes is already standard over the six types below; call is $\mathcal{L}$'s own, generalized past their Adversary-only restriction (below). (Chapter 10.)

Six base types, not $\mathcal{L}$'s five: integer, boolean, string, set, table — $\mathcal{L}$'s array, kept as a function's type throughout, $\mathtt{L} : \mathbb{N} \to \{0,1\}^n \cup \{\square\}$ reading exactly as $A : \{0,1\}^* \to \{0,1\}^* \cup \{\mathsf{undefined}\}$ does there, $\square$ this book's undefined — and identity, this book's own addition, below. Only one **for**-form is attested, **for** $x \in S$ **do**, and the one instance of it binds a pair directly, **for** $(m, y) \in N$ **do** (Chapter 20, lines 17 and 16) — an obvious sugar for binding one variable and destructuring it. The bounded integer-range form $\mathcal{L}$ also allows is in reserve, unused so far, and licensed the moment a core needs it.

The caller is implicit. Every operation's core is headed $\mathtt{id}.\textsc{Op}(\mathtt{in})$ **from** $\mathtt{id}'$, and $\mathtt{id}'$ — the caller's identity — is readable anywhere in the core though it names no argument of Op. A plain oracle in $\mathcal{L}$ carries nothing like it; this book's cores need it because so many begin by asking exactly who is calling before anything else runs, $\mathcal{G}_{\mathsf{Clock}}$'s operations being representative (the guard at line 13 tests $\mathtt{id}'.\mathrm{F}$ and $\mathtt{id}.\mathrm{P}$'s corruption before the core does anything else).

A core may call another core. $\mathcal{L}$ reserves the *call* expression for the Adversary alone; an oracle inside one of their games may not place one. Every functionality, protocol and game in this book does — the protocol of Chapter 20 reads $\mathcal{F}_{\mathsf{PKI}}$ and $\mathcal{F}_{\mathsf{Sig}}$ mid-operation (lines 22 and neighbours), and $\mathcal{F}_{\mathsf{Rand}}$'s own unpredictability game calls back into $\mathcal{F}_{\mathsf{Rand}}$ itself (line 5) — because this book's functionalities and protocols are built from others by declaration, **U** naming exactly what a core may

call (Chapter 1), and a game's shell is a functionality like any other. The call itself is $\mathcal{L}$'s own, unrelaxed: $\texttt{id}''.\textsc{Op}(e_1, \ldots)$ **as** id evaluates like any other expression and resumes the calling core at the same line, under its own identity. What stays strictly outside a core is the *general* form, with an unbound continuation k standing for an arbitrary resumption point rather than "the next line of this same core" — that is the execution model's own notation for what a call means in general, never a core's, and Section 3.6 is where it is defined.

Require is sugar. A core's own **require** e line, used throughout the execution-model chapters and inside plenty of the cores to come, is sugar for **if not** e **then return** $\perp$ — not literally REJ, even though a failed guard elsewhere returns exactly that. Section 3.6 makes REJ the one value no core may return as its own output, laundering a core-level REJ into $\perp$ before delivery; a core's own **require** merely names the common case compactly. $\mathcal{G}_{\mathsf{Clock}}$'s operations spell the same guard out by hand, with an ordinary **if . . . then return** (line 13 again): both are the one core-language statement, sugared or not.

Table indices need no new rule. reg[id.P], L[ctr]: neither is string-indexed, and $\mathcal{L}$ already has the answer, its own third enhancement ([5]) — a non-string index is passed through encode$(\cdot)$ first, silently. Every table this book writes that is not string-indexed already reads this way; there is nothing to add here beyond citing them for it.

Identity is a sixth type. One departure is not sugar. Definition 1.1 makes an identity a pair $\texttt{id} = (\mathit{pid}, \mathsf{P})$ of a process id and a party, a process id itself a triple $(\mathsf{F}, \mathsf{s}, \mathsf{i})$ — and a core reads through both levels at once, id.F, id.s, id.P named projections, id.i the one field no core ever reads (Chapter 1 says so outright). None of $\mathcal{L}$'s five types carries a projection, and encode$(\cdot)$ gives only equality-preservation, by [5]'s own account of it — an identity is read apart far too often for that to serve. So identity joins the type list as a sixth base type, structured rather than atomic, with projection and componentwise equality as its only operations — the one place this book's cores ask $\mathcal{L}$ for something it does not already have.

Reading the code. The grammar above is what a core may compute; how it is set on the page is a separate promise, and one already mostly kept. Keywords are bold, algorithmicx's default: **if**, **then**, **else**, **for**, **do**, **return**, and **require**. An operation's name carries a link wherever it is

declared and wherever it is called back to, both rendered in accentsoft small caps — one macro anchors the declaration, two more point back to it from every call site. State lives in `x`, typewriter, reserved for a core's own mutable fields; `id`, `in`, `out` stay italic, bound parameters rather than state, and the two must not be run together — a future reader deciding what a core's own record type looks like needs exactly this distinction and no other. Comments print in commenttint, italic, off to the side, never load-bearing. That copy-editing pass over the chapters already written is done, not new notation but a uniformity sweep: a first prose mention of an already-declared operation is linked back to its declaration throughout, and the constant atoms — OK, REJ, TRUE, FALSE, DONE, together with every trace tag and storage key elsewhere in this book — carry commenttint rather than an operation name's own colour, so bare colour now does separate "this is a name" from "this is a value."

What the grammar and the departures above buy, together, is a single test: from Chapter 13 on, a line of this book's math is a core's own code exactly when it type-checks against the table above plus the departures named here, and it is a specification — a game's verdict, a property's quantifier, SAN's own pick — exactly when it does not. Closing the gap to an actual language, OCaml or otherwise, is then a matter of compiling six types and a handful of statement forms once, not of first deciding, chapter by chapter, what a given display of symbols was ever claiming to compute.

CHAPTER 11

Identical Until Bad

Code-based proofs advance by *hops* [5, 13]: two games that are the same program except where a flag has been raised, and a bound on the probability of the raising. The fundamental lemma of game playing is the licence — if the games are identical until bad, their outcomes differ with probability at most Pr[BAD] — and its proof is a coupling: run both games on the same coins and they cannot part company before the flag goes up. In this paper a game is not one program. An execution runs an environment, a slot occupant, a register and a system; the token wanders among them; the occupant may be handed the token mid-operation and drive the rest of the execution before answering — the freedom Chapter 9 exists to discipline — and a hop must survive all of it. This chapter replays the lemma over executions, and the finding is that the wandering costs nothing: the lemma below has no hypothesis at all.

Where the flag lives. bad is an ordinary state variable of an ordinary machine: declared bad ← FALSE in INITIALIZE, never assigned any value but TRUE afterwards, owned like the rest of a machine's state by the machine that declares it. The framework already keeps every other hand off it — the one thing a machine can do to state it does not own is provoke calls that read or extend it, and at a corrupt party MEDIATE replaces *answers*, never state (Remark 5.4) — so a flag is raised by its owner's own code or not at all. Whether anyone may *read* it is immaterial, through an interface or through LEAK, which bypasses GUARD: until the flag is raised the two machines below answer any read alike, and after it is raised the lemma has already paid. Several machines may carry flags at once — members sharing a name included, each owning its own — and BAD below is the event that *some* machine raises *some* flag.

Definition 11.1 (Identical until bad)**.** Standard functionalities $\mathcal{F}$ and $\mathcal{F}'$ are *identical until bad* if their five fields agree; their INITIALIZEs agree line for line, declarations common to both sides and `bad` ← FALSE among them; and their remaining code agrees except inside *bad-guarded* blocks, of two shapes: the continuation of an assignment `bad` ← TRUE, from just past the assignment to the end of its *enclosing block* — the branch, loop or operation body the assignment stands in, so that everything in the continuation is reached only across the assignment — and the body of a conditional whose test is `bad`. Erasing the bad-guarded blocks from both sides leaves one functionality, operation for operation and line for line, LEAK included; in particular the assignments themselves are common code, each heading its own divergent continuation, and neither side assigns `bad` any value but TRUE. Equal functionalities are identical until bad, with nothing to erase. Two occupants of the adversary slot are identical until bad when their INITIALIZEs and cores are, read the same way. Systems π and π' are identical until bad if some bijection pairs every member of one with a member of the other identical until bad with it.

Two things about the shape, both [5]'s. Membership is checked by inspection — it is a comparison of code, not a claim about behaviour — which is what keeps hops cheap to audit. And it is built so that a divergent statement cannot run early: the only ways into a bad-guarded block are its head, an assignment that raises the flag as it is crossed, and a test of the flag itself, and the flag is monotone — so nothing bad-guarded runs while its owner's flag is down. Because paired members carry equal fields, the two systems occupy the same identities and use the same names, $\langle\pi\rangle = \langle\pi'\rangle$, and one class of environments serves both: $\mathbb{Z}_{\pi,\pi'} = \mathbb{Z}_{\pi,\pi} = \mathbb{Z}_{\pi',\pi'}$.

Lemma 11.2 (The fundamental lemma, over systems)**.** *Let systems π and π' be identical until bad, let occupants $\mathcal{X}$ and $\mathcal{X}'$ of the adversary slot be identical until bad, and let $\mathcal{Z} \in \mathbb{Z}_{\pi,\pi'}$. Write* BAD *for the event that some machine executes an assignment* `bad` ← TRUE*, in whichever execution is named. Then*

$$\Pr\big[\text{BAD } \textit{in } \text{EXEC}(\pi, \mathcal{Z}, \mathcal{X}, \mathcal{C})\big] \;=\; \Pr\big[\text{BAD } \textit{in } \text{EXEC}(\pi', \mathcal{Z}, \mathcal{X}', \mathcal{C})\big],$$

and, writing $\Pr[\textsc{Bad}]$ *for the common value,*

$$\big|\Pr\big[\textsc{Exec}(\pi,\mathcal{Z},\mathcal{X},\mathcal{C})=1\big] - \Pr\big[\textsc{Exec}(\pi',\mathcal{Z},\mathcal{X}',\mathcal{C})=1\big]\big| \leqslant \Pr[\textsc{Bad}].$$

If moreover the two systems are game systems — the pairing then matches game to game, fields being preserved — the same holds of every finalization $\textsc{Fin}$*, which the two games share:*

$$\big|\Pr\big[\textsc{Win}_{\textsc{Fin}} \textit{ in } \textsc{Exec}(\pi,\mathcal{Z},\mathcal{X},\mathcal{C})\big] - \Pr\big[\textsc{Win}_{\textsc{Fin}} \textit{ in } \textsc{Exec}(\pi',\mathcal{Z},\mathcal{X}',\mathcal{C})\big]\big| \leqslant \Pr[\textsc{Bad}].$$

Proof. One coupling, read pointwise. Fixing every tape fixes both executions outright, so the two may be taken from one sample and compared step by step. The induction shows that while no flag is up the two correspond strictly and the token stands on *common* code — which is exactly what Definition 11.1's two bad-guarded shapes buy, neither being reachable except across the assignment that raises the flag or through a test that fails while it is down. The first raising is therefore the same step on both sides, and that is the first display: $\Pr[\textsc{Bad}]$ is one number rather than two that happen to share a bound. Off $\textsc{Bad}$ the two runs are equal outright, so every fact of the run has a single indicator there, and each difference of probabilities is at most the measure of the set where indicators may differ. Two conventions of Chapters 1 and 3.6 are used, the ones the absorption lemma restates: each machine draws from its own coin tape, uniform and independent of every other's, read only when its code samples — an environment's hosted copies keeping tapes of their own, bundling merging nothing — and $\textsc{Initialize}$ places no calls. Fixing every tape therefore fixes an execution outright: which machine holds the token, at which line of which operation, with what state, and what its next statement does are functions of the configuration, coins entering only as the next unread positions of the holder's own tape.

Couple the two executions. The pairing matches machine to machine — member to member, occupant to occupant, the environment and the register each to themselves — and paired machines occupy the same identities, so give paired machines equal tapes. Each execution

separately keeps its distribution, tapes on either side being uniform and independent, so both may be read off one sample of tapes and compared pointwise. Say the two *correspond strictly* at a configuration if (C1)–(C4) of Definition 4.17 hold under the pairing — equal states, equal unread tapes, the token in paired hands at the same line with the same local values, suspended callers matching frame for frame — with no relay anywhere and nothing elided.

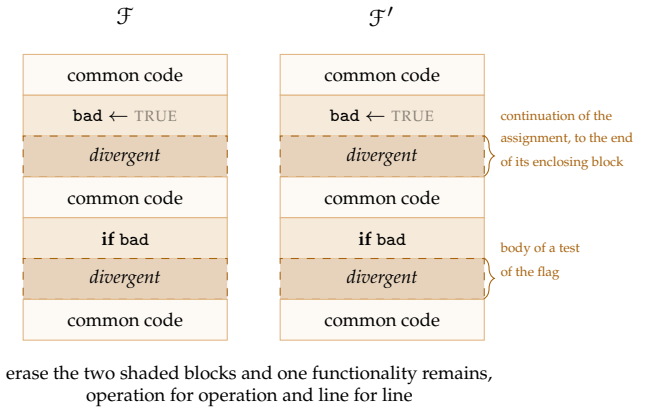
At the start they correspond strictly. Line 1 initialises the register and every member on both sides, in whatever order — INITIALIZE places no calls and touches no state but its owner's, so the order leaves no trace in the configuration; paired members run INITIALIZEs that agree line for line, `bad` ← FALSE among them, placing no calls and drawing, where they draw, the same positions of equal tapes; and line 2 hands both tokens to the one environment at the same identity.

Suppose the two correspond strictly, no flag yet raised on either side, and let one statement run. The token stands in paired machines at the same line, and that line is *common* code: a bad-guarded statement is reached only through its head or through a test of the flag, the head raises the flag as it is crossed, the test fails while the flag is down, and no flag is up. So the statement is the same statement, read over the same state and the same next coins, histories having consumed tape for tape alike. If it writes, both sides write alike; if it returns, both resume the same frame with the same value; if it places a call, both place the same call — target, payload and claimed identity — met by the same wrappers over equal fields and an equal register state, landing at paired machines, or at no machine and answered REJ alike; and every delivery Section 3.6 fixes — REJ for a refused **require**, $\perp$ where a core would return REJ or a mediated slot answers one — is a rule, not a choice. Strict correspondence survives the step. And if the statement is an assignment `bad` ← TRUE — common code, standing at the head of its divergent continuation — both sides execute it, and the step is the first raising in both executions at once.

Pointwise on the coupled space, then: either no flag is ever raised and the two executions correspond strictly at every step — so one halts exactly when the other does, with the same output, the environment being one machine resumed alike — or there is a first raising, it is simultaneous, and the executions agree up to and through it. The event BAD therefore has one indicator on the coupled space, which is the first display, with equality. Off BAD the runs are equal outright, so

any fact of the run has one indicator there too: the output, and, for game systems, the event WIN_{FIN} that the interface $(\mathcal{G}\text{m}.pid, \mathsf{Z}).\text{FULLFIN}$ returns 1 (Definition 5.8). Each difference of probabilities is at most the measure of the set where its indicators may differ, which is $\Pr[\text{BAD}]$. □

The definition and the proof have a picture each, and they are worth seeing together, since one is exactly what the other consumes. Above is what Definition 11.1 asks of the code; below is what the coupling then does with it. Neither panel uses the box-and-wire vocabulary of the earlier figures, and that is the point: nothing here is moved from one machine to another, so there is nothing to draw a wire between.



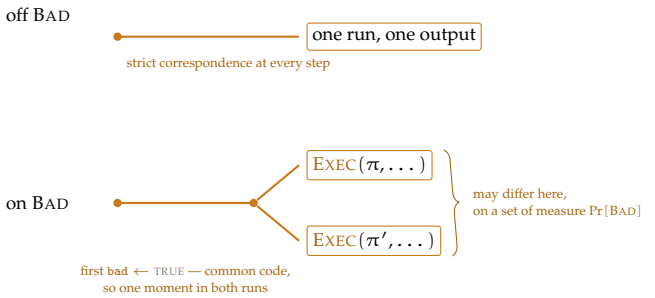


Figure 6: Identical until bad, in two panels. Above, what Definition 11.1 asks of the code; below, what the coupling does with it. Neither panel uses the box-and-wire vocabulary of the earlier figures, because nothing here moves from one machine to another. (Definition 11.1 and Lemma 11.2.)

The upper panel is checkable by inspection, which is what keeps a hop cheap to audit: it is a comparison of code, not a claim about behaviour. The two shaded shapes are the only ones allowed, and between them they are why nothing shaded can run while the flag is down — the only ways in are the head, which raises the flag as it is crossed, and a test of the flag, which fails while it is down. So the shading above is exactly what runs beyond the fork below, and nowhere else.

The fork is a single point, and that is the whole of the argument. The statement heading each divergent block is itself common code, so both runs execute it at the same step of the same coupled sample — which gives the lemma's *first* display, that $\Pr[\text{BAD}]$ is the same number in the two executions rather than two numbers with one bound. Off BAD there is no fork at all: the two runs are one run, so every fact of it has a single indicator there — the output, and, for game systems, WIN_{FIN} — and each difference of probabilities is at most the measure of the set where indicators may differ. That set is the lower branch, and its measure is $\Pr[\text{BAD}]$.

Remark 11.3 (Nothing is assumed)**.** The lemma asks nothing of the systems or of the slot: not well-formedness, not tightness, not refusal-obliviousness, and — worth savouring — not the absence of responsive calls. Every result that moves a machine pays a hypothesis for the move: Theorem 4.29 and Proposition 5.9 exclude responsive cores, Corollary 5.11 inherits the exclusion on both sides at once, and the notifying clock, the network and the channel sit outside all three for exactly that reason. The lemma above is indifferent — their responsive notifications hop as cheaply as anything else — because nothing is moved. Like the regrouping it is an identity about runs, resting on nothing but the determinism of an execution once its tapes are fixed, not a comparison of what two runs achieve. No budget appears for the same reason: exactness needs none, and Definition 4.30 enters only when an application sets about bounding $\Pr[\text{BAD}]$.

Remark 11.4 (Why not by absorption)**.** There is a tempting shorter route: absorb the whole execution into the environment

— Lemma 4.18 exists to move machines — until each side is a single program, and quote [5] on the pair. It works where it works, and it is instructive to see why it is the worse lemma. Absorption's hypotheses come along: the slot occupant moved must be refusal-oblivious, no core may place a responsive call on a ceded identity (Remark 9.4), and the environment's *par* must hold no Z-identity — so the reduction proves the lemma only for the systems the dummy theorem already covers, and the notifying clock, for one, falls outside it. The direct coupling is no longer than the reduction and carries nothing: the machinery of Definition 4.17 was built for one execution rebracketed, and two executions coupled on equal tapes is the same induction with an easier case analysis — no relay, no rewriting of claims, no refusal to smooth over.

What the lemma is for, concretely. Emulation arguments hop between neighbouring *ideal* objects: a functionality and a lazier variant, a simulator and the same simulator with one check removed. Each hop is a pair identical until bad, the flags raised where the two part company, and the lemma prices the hop at Pr[BAD] — in the same execution, against the same environment, with the occupants hopping alongside where a proof needs it. What remains is to bound Pr[BAD], and that is generally a *property*: a finalization whose verdict is precisely that the raising condition occurred, priced by Definition 5.8 and inherited, where transfer applies, through Corollary 5.11. The canonical instance sits at the signature functionality of Chapter 15. Instrument $\mathcal{F}_{\mathsf{Sig}}$ to raise bad where VERIFY's honest-key test fires — an addition no caller can see, the flag being written and never read — and set beside it the variant that then lets the forgery stand. The two are identical until bad, and the hop between them costs the probability that a forgery is ever submitted — the quantity FINUF exists to police.

CHAPTER 12

Sanitization

We will often need to *sanitize* simulator outputs, so attacks do not affect ideal functionalities too adversely. In honest-majority settings, for instance, we typically sanitize a $\perp$ output to a non-$\perp$ one to preserve liveness. We write such a procedure $\textsc{San}[\textsc{Clean}](v; c)$, where $\textsc{Clean}$ is a predicate on a value together with whatever context it consults, and replace a simulator output v by $v' \leftarrow \textsc{San}[\textsc{Clean}](v; c)$, which operates as below unless stated otherwise. The context c after the semicolon lists the functionality state the predicate reads; it is read-only, and different predicates take different contexts.

Sanitizer $\textsc{San}[\textsc{Clean}]$
predicate $\textsc{Clean}$, value v, context c

$\textsc{San}[\textsc{Clean}](v; c)$:
1: **if** $\textsc{Clean}(v; c)$ **then**
2: **return** v
3: pick $v' \in \{ v' : \textsc{Clean}(v'; c) \}$
4: **return** v'

So sanitization acts only when $\textsc{Clean}(v; c)$ fails, and what it returns then satisfies $\textsc{Clean}$.[1] Our sanitizers need no more than *read-only* access to the functionality state. How line 3 picks is part of the specification — the picked value reaches the caller, so different rules are observably different functionalities — and we fix the lexicographically first good value throughout. The choice matters less than it looks: the

[1] A good value exists whenever the space $\textsc{Clean}$ draws from is infinite and everything it excludes beyond non-membership is a value named in the context c. Both halves are needed for the predicates of Chapter 15: membership in an infinite $\mathcal{K}$ or Σ is the first conjunct of each, and everything else they exclude is an entry of a table passed in c. Under Definition 4.30 the context is finitely supported in any execution — a bounded run records finitely many entries — so the excluded set is finite inside an infinite space.

pick surfaces only when the slot's own answer fails CLEAN, and an adversary preferring some particular good value may simply submit it.

CHAPTER 13

Randomness

13.1 Functionality

$\mathcal{F}_{\mathsf{Rand}}$ is the smallest functionality in this paper and the right one to read first. It hands out uniform n-bit strings, remembers what it handed out, lets a party forget an entry, and leaks what it still remembers. Chapter 1 named it and owed no code; here is the code.

Two things make it worth its own chapter rather than a footnote. It places no call at all — not on the clock, not on the adversary — so it is the one example where nothing at all is left to the slot, and Chapter 9 has nothing to price. And its property, unlike every other in this paper, is not attained at 0: randomness is unpredictable up to a probability the bound has to name, which is where the ε of Definition 5.8 stops being decorative.

The fields. $\mathcal{F}_{\mathsf{Sig}}$'s pattern. *pid*, **P** and **N** are parameters, one instance per session, **N** naming the protocol served in the local pattern of Chapter 1. $\mathbf{U} := \varnothing$, there being no call to make. $par := n$, the width of the strings, which RND reads at line 5.

Erasure is what the leak is for. LEAK returns L entire, so a corrupt party's whole history of draws is the adversary's. That is not a defect but the point of having ERASE beside it: a protocol that draws randomness it must not retain erases the entry, and what has been erased cannot leak. Read that way $\mathcal{F}_{\mathsf{Rand}}$ is a model of secure erasure as much as of sampling, and the pair is why the store of Chapter 14 is shaped the same way. One consequence is worth stating plainly, since it is easy to state wrongly: while nothing is erased, the number of entries in L is the number of times RND was called, so the leak reports the count as well as the values; erasing is exactly what takes an entry out of that count.

Functionality $\mathcal{F}_{\mathsf{Rand}}$
pid, **P**, **N**, **U** := ∅, *par* := n

INITIALIZE():
1: `ctr` ← 0
2: `L` : $\mathbb{N} \to \{0,1\}^n \cup \{\square\}$
3: `L`[∗] ← □

id.RND() from `id'`
4: `ctr` ← `ctr` + 1
5: `r` ←$ $\{0,1\}^n$ *// the one sampling in the paper*
6: `L`[`ctr`] ← `r`
7: **return** (`ctr`, `r`) *// the index, so a caller may erase*

id.ERASE(i) from `id'`
8: `L`[`i`] ← □
9: **return** OK

id.LEAK() from `id'`
10: **return** `L`

Three remarks on the code. The operation RND answers with the index it wrote as well as the string. A caller that means to erase a draw must be able to name it, and `ctr` is the functionality's state and not the caller's, so without line 7 the ERASE beside it could only be used by a caller willing to guess — the protocol of Section 21.2 erases every coin it draws, and this is what lets it. Entries are indexed by a counter and never rewritten, so ERASE needs no companion to $\mathcal{F}_{\mathsf{Store}}$'s distinction between never-written and erased: an index RND has passed is never reused, and □ at an index below `ctr` says the entry was erased while □ above it says the counter has not reached there. And the counter is shared across parties, as any functionality's state is: two parties drawing from one instance draw distinct entries, which is what makes $\mathcal{F}_{\mathsf{Rand}}$ a source rather than a per-party generator.

13.2 Properties

One property, FINUNP, *unpredictability*: no environment names in advance a value a later draw returns. It takes the search reading of Definition 5.8. The game is played over $\gamma := \{\mathcal{G}\mathfrak{m}\} \cup \{\mathcal{F}_{\mathsf{Rand}}\}$ with $\mathcal{F}_{\mathsf{Rand}}.\mathbf{N} := \{\mathcal{G}\mathfrak{m}.pid\}$ pinned and $\mathcal{F}_{\mathsf{Rand}}.\mathbf{P} \subseteq \mathcal{G}\mathfrak{m}.\mathbf{P}$, Section 15.2's tightness argument once more, and for its reason: a draw the game does not see is a draw the verdict cannot rule on.

The shell carries one interface that is not a relay. The operation GUESS places no call and touches $\mathcal{F}_{\mathsf{Rand}}$ not at all; it exists so that the environment can go on record, and the trace being a sequence is what lets the verdict ask whether it went on record *first*. That is Section 17.2's use of order, borrowed for a different purpose: steadiness read the

order of answers, unpredictability reads the order of a claim against an answer.

Game shell over $\mathcal{F}_{\mathsf{Rand}}$; the finalization follows

pid $:=$ $(\mathcal{G}\mathfrak{m}.\mathsf{F}, 0, 0)$, $\mathbf{P} := \mathcal{F}_{\mathsf{Rand}}.\mathbf{P} \cup \{Z\}$, $\mathbf{N} := \mathbf{Z}$, $\mathbf{U} := \{(\mathcal{F}_{\mathsf{Rand}}, \text{SERVES}_{\mathcal{G}\mathfrak{m}})\}$, *par* $:= \bot$

INITIALIZE():
1: DONE $\leftarrow$ FALSE
2: tr $\leftarrow$ ( ) *// the trace, a sequence*

id.RND() from id′
3: **require** ¬DONE
4: **require** id.P $\neq Z$ *// the root only finalizes*
5: $(i, r) \leftarrow \mathtt{id}_{\mathcal{F}_{\mathsf{Rand}}}$.FULLRND() as id *// the index is not recorded*
6: tr $\leftarrow$ tr $\cdot$ (RND, id.P, r)
7: **return** r

id.ERASE(i) from id′
8: **require** ¬DONE
9: **require** id.P $\neq Z$
10: $\mathtt{id}_{\mathcal{F}_{\mathsf{Rand}}}$.FULLERASE(i) as id
11: tr $\leftarrow$ tr $\cdot$ (ERASE, id.P, i)
12: **return** OK

id.GUESS(y) from id′
13: **require** ¬DONE
14: tr $\leftarrow$ tr $\cdot$ (GUESS, id.P, y) *// no call; the root may guess too*
15: **return** OK

id.LEAK() from id′
16: **return** $\bot$

The finalization of $\mathcal{G}\mathfrak{m}$; the shell over $\mathcal{F}_{\mathsf{Rand}}$ is shared

id.FINUNP() from id′ (unpredictability)
17: **require** ¬DONE
18: **require** id.P $= Z$
19: DONE $\leftarrow$ TRUE
20: parse tr as $(e_j)_{j=1}^{m}$
21: $w \leftarrow 1$ if $\exists\, j < k\ \exists\, y:\ e_j = (\text{GUESS}, \cdot, y) \wedge e_k = (\text{RND}, \cdot, y)$
22: else $w \leftarrow 0$
23: **return** w

Two things about the code. The relays refuse the root and GUESS does not, and the asymmetry is exactly Section 17.2's: a relay at the root would address a party $\mathcal{F}_{\mathsf{Rand}}$ does not serve, be answered REJ at line 1, and append (RND, Z, REJ) to the trace — whereupon an environment that guessed REJ would win outright, having predicted nothing. Refused at the relay, nothing is appended, and every RND entry carries a string $\mathcal{F}_{\mathsf{Rand}}$ actually drew. The operation GUESS places no call, so it has nothing to be refused for and is served everywhere, the root included. And the verdict asks honesty of nobody: at a corrupt party the game's own wrapper intercepts before the core runs (Remark 5.4), so

every entry in $\mathtt{tr}$ is an honest core's already, exactly as in Section 17.2.

Proposition 13.1 ($\mathcal{F}_{\mathsf{Rand}}$ is unpredictable). *Let $\mathcal{Z}$ place at most q_G calls on* GUESS *and at most q_R on* RND. *Then for every occupant $\mathcal{X}$ of the adversary slot,*

$$\Pr[\mathrm{WIN}_{\mathrm{FINUNP}}] \leqslant q_G\, q_R\, 2^{-n}.$$

So $\{\mathcal{F}_{\mathsf{Rand}}\}$ has unpredictability within $q_G q_R 2^{-n}$ against $(\mathbb{Z}, \mathbb{X})$ for $\mathbb{Z}$ any class meeting those counts and $\mathbb{X}$ the class of all occupants.

Proof. Fix k with $e_k = (\mathrm{RND}, \cdot, r)$. By tightness no caller but $\mathcal{G}m$ reaches $\mathcal{F}_{\mathsf{Rand}}$, and $\mathcal{G}m$ reaches it only through the shell's RND relay (line 5), so r is the value $\mathcal{F}_{\mathsf{Rand}}$'s own RND drew at line 5: uniform on $\{0,1\}^n$ and independent of everything the execution did before it, $\mathcal{F}_{\mathsf{Rand}}$ placing no call and reading no state to draw it.

Condition on the execution up to the moment before that draw. Every GUESS entry with $j < k$ is determined by that history, and there are at most q_G of them, so the probability that r equals any of their values is at most $q_G 2^{-n}$. There are at most q_R choices of k, and the verdict at line 21 returns 1 only if some pair (j, k) matches, so a union bound over k gives $q_G q_R 2^{-n}$.

Two details the bound rests on. The occupant of the slot does not appear: $\mathcal{F}_{\mathsf{Rand}}$'s cores place no call, so nothing $\mathcal{X}$ does reaches the sampling, and the same bound holds against every occupant without an absorption argument. And a guess made *after* a draw is worthless by construction, the verdict asking $j < k$; an environment that reads r and then guesses it has predicted nothing, which is what makes the trace's order the content of this verdict. □

The bound is the first in this paper that is not 0, and it is worth saying what that changes. Corollary 5.11 reads the same either way: a system π put in place of $\{\mathcal{F}_{\mathsf{Rand}}\}$ has unpredictability within $q_G q_R 2^{-n} + \varepsilon$ under the search reading, the ideal bound and the emulation error simply adding. What it changes is that the ideal side is no longer free — a realization inherits a bound that was already nontrivial, and the two terms are of different kinds: one is the sampling's own, and no protocol does better than it, while the other is what the realization costs. Transfer itself is as cheap as it gets: $\mathcal{F}_{\mathsf{Rand}}$'s cores place no call, responsive or otherwise, so Remark 9.4's caveat never arises and Theorem 4.29 is untouched, exactly as for $\mathcal{G}_{\mathsf{PKI}}$ (Section 16.2).

CHAPTER 14

Storage

14.1 Functionality

$\mathcal{F}_{\mathsf{Store}}$ is $\mathcal{F}_{\mathsf{Rand}}$'s companion: where one produces values and lets them be forgotten, the other takes values and holds them until they are. A party puts a value at an index, gets it back, and erases it; the leak reports what is still held. Together they are what Chapter 1 means by a protocol keeping no state on a machine tape — all randomness through $\mathcal{F}_{\mathsf{Rand}}$, all persistence through $\mathcal{F}_{\mathsf{Store}}$, each exposing what it holds through its own LEAK.

The fields. $\mathcal{F}_{\mathsf{Rand}}$'s exactly, save the parameter: *pid*, **P** and **N** parameters, $\mathbf{U} := \varnothing$, and $par := \bot$, a store reading none. Like $\mathcal{F}_{\mathsf{Rand}}$'s, its state is shared across the parties it serves — one party may get what another put, which is what makes it a store rather than a per-party tape, and what Chapter 1 means by sharing being the point.

Three states, not two. An index is in one of three conditions, and the code keeps them apart on purpose. It is $\square$ if nothing was ever put there; it holds a value if something was put and not erased; and it is $\bot$ if it was put and then erased. The operation PUT accepts an index in either empty condition, so an erased index is reusable; GET refuses only $\square$, so a get on an erased index is answered $\bot$ rather than refused. The distinction earns its place in the difference between those two answers: a caller can tell *nothing was ever here* from *something was here and is gone*, which is what makes erasure observable to the protocol that performed it rather than a silent hole. A store that hid the difference would refuse both alike and could not be asked whether an erasure had happened.

Functionality $\mathcal{F}_{\mathsf{Store}}$
pid, **P**, **N**, **U** := ∅, *par* := ⊥

INITIALIZE():
1: L : ℕ → {0,1}* ∪ {⊥,□}
2: L[∗] ← □

id.PUT(i, x) from id′
3: **require** L[i] ∈ {⊥,□} // an occupied index refuses
4: L[i] ← x
5: **return** OK

id.ERASE(i) from id′
6: L[i] ← ⊥
7: **return** OK

id.GET(i) from id′
8: **require** L[i] ≠ □ // never written refuses
9: **return** L[i] // ⊥ if erased

id.LEAK() from id′
10: **return** L

Two remarks on the code. The refusals are written as **require**s rather than as returns of REJ, which is the same thing said in the framework's own words: Section 3.6 answers a refused **require** with REJ, so lines 3 and 8 give exactly the two refusals a store owes, and no core of this paper returns REJ by writing it. And ERASE refuses nothing: erasing an index that holds nothing is a no-op that answers OK, because a protocol erasing defensively should not have to know whether there was anything there.

14.2 Properties

Two properties, and they are the two halves of what a store is for. FINKEEP, *persistence*: a value put and not erased is the value got back. FINGONE, *erasure*: a value erased is not got back. Both take the search reading, and both are two finalizations of one game — they read the same trace of the same three relays and differ in the verdict alone, which is the test Section 17.2 gives for sharing a shell.

They are played over $\gamma := \{\mathcal{G}\mathfrak{m}\} \cup \{\mathcal{F}_{\mathsf{Store}}\}$ with $\mathcal{F}_{\mathsf{Store}}.\mathbf{N} := \{\mathcal{G}\mathfrak{m}.pid\}$ pinned and $\mathcal{F}_{\mathsf{Store}}.\mathbf{P} \subseteq \mathcal{G}\mathfrak{m}.\mathbf{P}$, tightness as before and for the same reason: a put the game does not see is a value the verdict cannot account for. The relays refuse the root, for Section 13.2's reason: both verdicts compare a got value against a put one, so a REJ of the game's own making in the trace would be read as a value and could win either.

Game shell over $\mathcal{F}_{\text{Store}}$; the finalizations follow

$pid := (\mathcal{G}m.F, 0, 0)$, $\mathbf{P} := \mathcal{F}_{\text{Store}}.\mathbf{P} \cup \{Z\}$, $\mathbf{N} := \mathbf{Z}$, $\mathbf{U} := \{(\mathcal{F}_{\text{Store}}, \text{SERVES}_{\mathcal{G}m})\}$, $par := \bot$

INITIALIZE():
1: DONE $\leftarrow$ FALSE
2: tr $\leftarrow$ ( )

id.PUT(i, x) from id′
3: **require** ¬DONE
4: **require** id.P $\neq Z$
5: $a \leftarrow \text{id}_{\mathcal{F}_{\text{Store}}}$.FULLPUT(i, x) as id
6: tr $\leftarrow$ tr $\cdot$ (PUT, id.P, i, x, a)
7: **return** a

id.ERASE(i) from id′
8: **require** ¬DONE
9: **require** id.P $\neq Z$
10: $\text{id}_{\mathcal{F}_{\text{Store}}}$.FULLERASE(i) as id
11: tr $\leftarrow$ tr $\cdot$ (ERASE, id.P, i)
12: **return** OK

id.GET(i) from id′
13: **require** ¬DONE
14: **require** id.P $\neq Z$
15: $y \leftarrow \text{id}_{\mathcal{F}_{\text{Store}}}$.FULLGET(i) as id
16: tr $\leftarrow$ tr $\cdot$ (GET, id.P, i, y)
17: **return** y

id.LEAK() from id′
18: **return** $\bot$

The finalizations of $\mathcal{G}m$; the shell over $\mathcal{F}_{\text{Store}}$ is shared

id.FINKEEP() from id′ (persistence)
19: **require** ¬DONE
20: **require** id.P $= Z$
21: DONE $\leftarrow$ TRUE; parse tr as $(e_j)_{j=1}^{m}$
22: $w \leftarrow 1$ if $\exists j < k\ \exists i, x, y:\ e_j = (\text{PUT}, \cdot, i, x, \text{OK}) \wedge e_k = (\text{GET}, \cdot, i, y)$
23: $\quad \wedge\ y \neq x \wedge \neg\exists l \in (j, k) : e_l = (\text{ERASE}, \cdot, i)$;
24: else $w \leftarrow 0$
25: **return** w

id.FINGONE() from id′ (erasure)
26: **require** ¬DONE
27: **require** id.P $= Z$
28: DONE $\leftarrow$ TRUE; parse tr as $(e_j)_{j=1}^{m}$
29: $w \leftarrow 1$ if $\exists j < k < l\ \exists i, x:\ e_j = (\text{PUT}, \cdot, i, x, \text{OK})$
30: $\quad \wedge\ e_k = (\text{ERASE}, \cdot, i) \wedge e_l = (\text{GET}, \cdot, i, x)$
31: $\quad \wedge \neg\exists l' \in (k, l) : e_{l'} = (\text{PUT}, \cdot, i, \cdot, \text{OK})$;
32: else $w \leftarrow 0$
33: **return** w

The verdicts, in words. The finalization FINKEEP returns 1 if some get answered other than the value the last successful put at that index left there, no erasure intervening. The finalization FINGONE returns 1 if some get answered exactly the value an erasure was supposed to have removed, no put having restored it in between. Both read the trace's order, as Section 17.2's steadiness does and for the same reason:

a store is a claim about what happens between one call and the next, and between is an ordering.

Proposition 14.1 ($\mathcal{F}_{\mathsf{Store}}$ has both properties)**.** *For* FIN *either of* FINKEEP *and* FINGONE*, for every* $\mathcal{Z} \in \mathbb{Z}_{\gamma,\gamma}$ *and every occupant* $\mathcal{X}$ *of the adversary slot,* $\Pr[\mathrm{WIN}_{\mathrm{FIN}}] = 0$*. So* $\{\mathcal{F}_{\mathsf{Store}}\}$ *has each property within* 0 *against* $(\mathbb{Z}_{\gamma,\gamma}, \mathbb{X})$ *in the sense of Definition 5.8, for* $\mathbb{X}$ *the class of all occupants.*

Proof. One invariant carries both properties: $\mathtt{L}[i]$ changes only inside a call the trace records, and to the value that call names. It comes from tightness (no caller but the game passes the guard), from the game's own wrapper intercepting at a corrupt party before any core runs, and from $\mathcal{F}_{\mathsf{Store}}$'s cores placing no call, so nothing runs between a relay's call and its return and entries append in execution order. Both properties are then read off it by the same move: assume the verdict's conjuncts, and track which lines could have written $\mathtt{L}[i]$ between the entries named. Persistence turns on the put refusing an occupied index, so no later put can intervene unrecorded; erasure turns on $\{0,1\}^*$ containing neither $\perp$ nor $\square$, so the value a get returns after an erase cannot be the value put before it. Everything rests on one invariant: $\mathtt{L}[i]$ changes only inside a call the trace records, and to the value that call names. By tightness no caller but $\mathcal{G}\mathfrak{m}$ passes GUARD's line 2 at $\mathcal{F}_{\mathsf{Store}}$, and $\mathcal{G}\mathfrak{m}$ reaches it only through the shell's three relays (lines 5, 11 and 15); at a corrupt party the game's own wrapper intercepts a step earlier and no core of γ runs at all (Remark 5.4). $\mathcal{F}_{\mathsf{Store}}$'s cores place no call, so no other machine of γ runs between a relay's call and its return, and entries append in execution order. Line numbers below name $\mathcal{F}_{\mathsf{Store}}$'s own cores unless said otherwise.

For persistence: suppose $e_j = (\mathrm{PUT}, \cdot, i, x, \mathrm{OK})$ and $e_k = (\mathrm{GET}, \cdot, i, y)$ with $j < k$ and no ERASE at i between. The put answered OK, so PUT's line 3 passed and its line 4 set $\mathtt{L}[i] \leftarrow x$. Only that line and ERASE's line 6 write $\mathtt{L}[i]$, and by the invariant each is recorded: no ERASE entry at i lies between, and a further PUT at i answering OK cannot lie between either, line 3 refusing an occupied index and $\mathtt{L}[i]$ being occupied from j on. So $\mathtt{L}[i] = x \neq \square$ still at k, GET's line 8 passes, and its line 9 returns x; hence $y = x$ and the verdict at line 22 gives 0.

For erasure: suppose $e_j = (\mathrm{PUT}, \cdot, i, x, \mathrm{OK})$, $e_k = (\mathrm{ERASE}, \cdot, i)$ and $e_l = (\mathrm{GET}, \cdot, i, x)$ with $j < k < l$ and no successful PUT at i between k

and $\mathtt{l}$. ERASE's line 6 set $\mathtt{L}[\mathtt{i}] \leftarrow \bot$ at $\mathtt{k}$, and by the invariant the only line that could restore a value is PUT's line 4 inside a recorded PUT at $\mathtt{i}$ answering OK, which the hypothesis excludes. So $\mathtt{L}[\mathtt{i}] = \bot$ at $\mathtt{l}$, and GET's line 9 returns $\bot$. Now $\mathtt{x} \neq \bot$: $\mathtt{x}$ was written by line 4 from a PUT's argument, and $\{0,1\}^*$ contains neither $\bot$ nor $\square$. Hence the get did not answer $\mathtt{x}$, contrary to $e_{\mathtt{l}}$, and the verdict at line 29 gives 0. $\square$

Both transfer by Corollary 5.11 with nothing to discharge beyond its four standing conditions: $\mathcal{F}_{\mathsf{Store}}$'s cores place no call at all, so the responsive-call hypothesis is met the cheap way and dummy completeness is untouched, exactly as for $\mathcal{F}_{\mathsf{Rand}}$ and $\mathcal{G}_{\mathsf{PKI}}$. The three functionalities that ask the adversary nothing are the three that pay nothing, and the pattern is not a coincidence — what Section 17.2 charges the clock is the notification, and a functionality with no notification has no bill.

CHAPTER 15

Digital Signatures

15.1 Functionality

$\mathcal{F}_{\mathsf{Sig}}$ below idealizes digital signatures, in the tradition of Canetti's own ideal signature functionality [6]. Its process id is a parameter and not a constant. The name component is fixed — that is what the subscript records, by the naming convention of Chapter 1 — but the session and instance are not, so one system may hold a copy per session and several within a session, the situation composition exists to handle, and one Definition 1.2 would forbid outright if the whole process id were pinned. The code below reads neither component, so Chapter 7 lets it be read at any other — for an **N** meeting the condition of Definition 7.2, which any global or purely local one does. Recall from Section 3.4 that $\mathcal{A}(x)$ abbreviates a call to the adversary on behalf of the party being served; in the ideal execution the simulator answers those calls, and every one of them here is sanitized in the sense of Chapter 12 before use.

How we write ideal functionalities. The style is the same throughout. Wherever a value has to be produced and the specification does not pin it down, we let the simulator choose it and sanitize the choice on the fly. The simulator is then meant to have as much freedom as the security goal permits and no more — a design intention, not a theorem — and the goal stays visible in the predicates rather than buried in the code.

In $\mathcal{F}_{\mathsf{Sig}}$ this happens three times, and the first two protect correctness. The operation GEN takes the verification key from the adversary, and $\textsc{Clean}_{vk}$ makes it an actual key, distinct from every key already issued and free of any signature already recorded valid under it, so generation always succeeds and keys neither collide nor arrive preforged. The operation SIGN takes the signature, and $\textsc{Clean}_{\sigma}$ makes it an actual signature not already recorded invalid, so signing succeeds

whenever a key has been generated, and what it produces verifies. Locality asks for that, and asks in addition that the simulator answer without passing the token elsewhere, a condition on the simulator class rather than a property of the code. The operation VERIFY is sanitized for unforgeability instead: a verdict outside $\{0, 1\}$ becomes 0, and so does any verdict of 1 on a fresh triple under a key an honest party generated, which makes signatures *strongly* unforgeable. There the good value is unique, so the sanitization is written inline rather than through SAN. Verification keys are not assumed authenticated, so VERIFY takes a *vk* of its own.

LEAK, which Section 3.2 requires of an ideal functionality, hands the adversary the party's verification key together with the signatures recorded valid under it, and nothing else of the state. The operation INITIALIZE fixes the two tables and their types: `VK` holds one key per served party and `Ver` the verdict on each triple of key, message and signature, both starting everywhere at $\square$, the value that separates "not yet decided" from a recorded 0 or 1. Both $\mathcal{K}$ and Σ are infinite and contain neither $\perp$ nor $\square$: infinite so that the sanitizers never run out of good values, and clean of the two markers so that membership already refuses them.

Where locality is wanted, the simulator class must be restricted to the *responsive* ones: those answering each simulator call without placing any call of their own, so the token returns at once and simulator calls are subroutine rather than coroutine calls — the discipline Chapter 9 enforces in code. For non-interactive, stateless signatures this costs nothing, the usual simulators being responsive already.

One further line in GEN and VERIFY guards against re-entrance. Section 3.6 lets a suspended instance be re-entered, and both operations suspend at an adversary call between testing a table entry and writing it; each therefore re-tests the entry on resumption, and a value recorded meanwhile wins. Without the re-test two nested calls could issue two keys to one party, or return different verdicts on one triple — the very consistency `Ver` exists to record.

Functionality $\mathcal{F}_{\mathsf{Sig}}$
pid, **P**, **N**, **U** $:= \{(\mathcal{A}, \text{SERVES})\}$, *par* $:= \bot$

INITIALIZE():
1: $\mathtt{VK} : \mathcal{F}_{\mathsf{Sig}}.\mathbf{P} \to \mathcal{K} \cup \{\square\}$
2: $\mathtt{VK}[*] \leftarrow \square$
3: $\mathtt{Ver} : \mathcal{K} \times \mathcal{M} \times \Sigma \to \{0,1\} \cup \{\square\}$
4: $\mathtt{Ver}[*,*,*] \leftarrow \square$

id.GEN() from id′
5: **if** $\mathtt{VK}[\mathtt{id.P}] \neq \square$ **then**
6: **return** $\mathtt{VK}[\mathtt{id.P}]$
7: $vk \leftarrow \mathcal{A}(\mathtt{id.GEN})$
8: **if** $\mathtt{VK}[\mathtt{id.P}] \neq \square$ **then**
9: **return** $\mathtt{VK}[\mathtt{id.P}]$ *// recorded while suspended*
10: $vk \leftarrow \text{SAN}[\text{CLEAN}_{vk}](vk; \mathtt{VK}, \mathtt{Ver})$
11: $\mathtt{VK}[\mathtt{id.P}] \leftarrow vk$
12: **return** vk

id.SIGN(msg) from id′
13: $vk \leftarrow \mathtt{VK}[\mathtt{id.P}]$
14: **if** $vk = \square$ **then**
15: **return** $\bot$
16: $\sigma \leftarrow \mathcal{A}(\mathtt{id.SIGN}, msg)$
17: $\sigma \leftarrow \text{SAN}[\text{CLEAN}_{\sigma}](\sigma; msg, vk, \mathtt{Ver})$
18: $\mathtt{Ver}[vk, msg, \sigma] \leftarrow 1$
19: **return** σ

id.VERIFY(vk, msg, σ) from id′
20: **if** $\mathtt{Ver}[vk, msg, \sigma] \neq \square$ **then**
21: **return** $\mathtt{Ver}[vk, msg, \sigma]$
22: $b \leftarrow \mathcal{A}(\mathtt{id.VERIFY}, vk, msg, \sigma)$
23: **if** $\mathtt{Ver}[vk, msg, \sigma] \neq \square$ **then**
24: **return** $\mathtt{Ver}[vk, msg, \sigma]$ *// recorded while suspended*
25: **if** $b \notin \{0,1\}$ **then**
26: $b \leftarrow 0$
27: **if** $\exists\, \mathrm{P}' \notin \mathbf{C} : vk = \mathtt{VK}[\mathrm{P}']$ **then**
28: $b \leftarrow 0$ *// no forgery under an honest key*
29: $\mathtt{Ver}[vk, msg, \sigma] \leftarrow b$
30: **return** b

id.LEAK() from id′
31: **if** $\mathtt{VK}[\mathtt{id.P}] = \square$ **then**
32: **return** $(\bot, \varnothing)$
33: **return** $(\mathtt{VK}[\mathtt{id.P}], \{\,(msg, \sigma) : \mathtt{Ver}[\mathtt{VK}[\mathtt{id.P}], msg, \sigma] = 1\,\})$

$\text{CLEAN}_{vk}(vk; \mathtt{VK}, \mathtt{Ver})$:
34: **return** $vk \in \mathcal{K} \wedge \neg\exists\, \mathrm{P} : \mathtt{VK}[\mathrm{P}] = vk$
35: $\wedge\ \neg\exists\, msg, \sigma : \mathtt{Ver}[vk, msg, \sigma] = 1$

$\text{CLEAN}_{\sigma}(\sigma; msg, vk, \mathtt{Ver})$:
36: **return** $\sigma \in \Sigma \wedge \mathtt{Ver}[vk, msg, \sigma] \neq 0$

15.2 Properties

Definition 5.1 is a shape, and a shape is worth an instance. We give one game $\mathcal{G}\mathfrak{m}$ over $\mathcal{F}_{\mathsf{Sig}}$ carrying two properties: FINCOR, saying that what the signing interface produces verifies, and FINUF, saying that nothing else does. They are two finalizations of one shell rather than two games, which is what the shape now allows and what these two ask for — they read the same trace, of the same oracles, and differ in the verdict alone. The environment chooses which it is judged on by choosing which to call, and calling one closes the game against the other.

Both take the *search* reading of Definition 5.8,

$$\mathrm{ADV}^{\text{FINCOR}}_{\mathcal{G}\mathfrak{m},\mathcal{F}_{\mathsf{Sig}},\mathcal{Z},\mathcal{X}} := \Pr[\,\mathrm{WIN}_{\text{FINCOR}}\,],$$

$$\mathrm{ADV}^{\text{FINUF}}_{\mathcal{G}\mathfrak{m},\mathcal{F}_{\mathsf{Sig}},\mathcal{Z},\mathcal{X}} := \Pr[\,\mathrm{WIN}_{\text{FINUF}}\,],$$

because winning either is meant to be impossible and not merely no better than a coin: a signature scheme whose forgeries succeed half the time is broken, not average.

They are played over the game system $\gamma := \{\mathcal{G}\mathfrak{m}\} \cup \{\mathcal{F}_{\mathsf{Sig}}\}$, with $\mathcal{F}_{\mathsf{Sig}}.\mathbf{N} := \{\mathcal{G}\mathfrak{m}.\mathit{pid}\}$ — the game's process id, pinned, not its name — and $\mathcal{F}_{\mathsf{Sig}}.\mathbf{P} \subseteq \mathcal{G}\mathfrak{m}.\mathbf{P}$, so the relays exist at every party $\mathcal{F}_{\mathsf{Sig}}$ serves and, at the root, address a party it does not serve and are answered REJ at line 1 — the root being where the game finalizes, not where it plays. Inside an interface at id we write $\mathtt{id}_{\mathcal{F}_{\mathsf{Sig}}}$ for $(\mathcal{F}_{\mathsf{Sig}}.\mathit{pid}, \mathtt{id}.\mathrm{P})$, the interface of $\mathcal{F}_{\mathsf{Sig}}$ at the party the game was called at. That is tightness at $\{\mathcal{G}\mathfrak{m}\}$, which Definition 5.1 requires, and here it earns its keep twice over. It makes γ a legal game system; and it makes the trace *complete*. No caller but $\mathcal{G}\mathfrak{m}$ passes line 2 at $\mathcal{F}_{\mathsf{Sig}}$: not the environment, whose Z-identities and fresh names lie outside a pinned $\mathbf{N}$; not the adversary, whose claim is an A-identity; and not a machine the environment hosts at an invented instance of the game's name, which is what pinning the id rather than the name refuses. So every signature $\mathcal{F}_{\mathsf{Sig}}$ ever issues passes through line 8 below and is recorded. A game whose verdict is "this was not obtained from the signing interface" has no other way to know.

Game shell over $\mathcal{F}_{\mathsf{Sig}}$; the finalizations follow

$\mathit{pid} := (\mathcal{G}\mathfrak{m}.\mathrm{F}, 0, 0)$, $\mathbf{P} := \mathcal{F}_{\mathsf{Sig}}.\mathbf{P} \cup \{Z\}$, $\mathbf{N} := \mathbf{Z}$, $\mathbf{U} := \{(\mathcal{F}_{\mathsf{Sig}}, \text{SERVES}_{\mathcal{G}\mathfrak{m}})\}$, $\mathit{par} := \bot$

INITIALIZE():
1: DONE ← FALSE
2: tr ← ( ) // the trace, a sequence

id.GEN() from id′
3: **require** ¬DONE
4: $\mathit{vk} \leftarrow \mathtt{id}_{\mathcal{F}_{\mathsf{Sig}}}$.FULLGEN() as id
5: tr ← tr · (GEN, id.P, vk)
6: **return** vk

id.SIGN(msg) from id′
7: **require** ¬DONE
8: $\sigma \leftarrow \mathtt{id}_{\mathcal{F}_{\mathsf{Sig}}}$.FULLSIGN($\mathit{msg}$) as id
9: tr ← tr · (SIGN, id.P, msg, σ)
10: **return** σ

id.VERIFY($\mathit{vk}, \mathit{msg}, \sigma$) from id′
11: **require** ¬DONE
12: $\mathrm{b} \leftarrow \mathtt{id}_{\mathcal{F}_{\mathsf{Sig}}}$.FULLVERIFY($\mathit{vk}, \mathit{msg}, \sigma$) as id
13: tr ← tr · (VER, id.P, $\mathit{vk}, \mathit{msg}, \sigma$, b)
14: **return** b

id.LEAK() from id′
15: **return** $\bot$

The finalizations of $\mathcal{G}\mathrm{m}$; the shell over $\mathcal{F}_{\mathsf{Sig}}$ is shared

id.FINCOR() from id′ (correctness)
16: **require** ¬DONE
17: **require** id.P = Z
18: DONE ← TRUE
19: **C** ← (C, Z).FULLSTATUS() as id
20: $w \leftarrow 1$ if $\exists$ P, Q, *vk*, *msg*, σ : P ∉ **C** ∧ Q ∉ **C** ∧ σ ∈ Σ
21: ∧ (GEN, P, *vk*) ∈ tr ∧ (SIGN, P, *msg*, σ) ∈ tr ∧ (VER, Q, *vk*, *msg*, σ, 0) ∈ tr;
22: else $w \leftarrow 0$
23: **return** w

id.FINUF() from id′ (unforgeability)
24: **require** ¬DONE
25: **require** id.P = Z
26: DONE ← TRUE
27: **C** ← (C, Z).FULLSTATUS() as id
28: $w \leftarrow 1$ if $\exists$ P, Q, *vk*, *msg*, σ : P ∉ **C** ∧ Q ∉ **C**
29: ∧ (GEN, P, *vk*) ∈ tr ∧ (VER, Q, *vk*, *msg*, σ, 1) ∈ tr ∧ (SIGN, P, *msg*, σ) ∉ tr;
30: else $w \leftarrow 0$
31: **return** w

Four things about the code. The trace is a sequence, ∈ means occurrence in it, and no verdict reads the order. The verdicts read tr and the register and call nothing, which is why the root exemption in $\mathrm{SERVES}_{\mathcal{G}\mathrm{m}}$ suffices and $\mathcal{F}_{\mathsf{Sig}}$ need not serve Z. Both ask P ∉ **C** at the end rather than at the time of signing, and monotonicity (Section 3.5) makes that the stronger reading: a party honest when the game closes was honest throughout. Correctness asks σ ∈ Σ because SIGN answers ⊥ where no key has been generated, and ⊥ is no signature to hold against the functionality. And unforgeability is *strong*: line 28 excludes the triple, not the message, so a second signature on a signed message counts as a forgery — which is the notion Chapter 15's sanitizer was written to give.

Proposition 15.1 ($\mathcal{F}_{\mathsf{Sig}}$ has both properties)**.** *For* FIN *either of* FINCOR *and* FINUF*, for every* $\mathcal{Z} \in \mathbb{Z}_{\gamma,\gamma}$ *and every occupant* $\mathcal{X}$ *of the adversary slot,* $\Pr[\mathrm{WIN}_{\mathrm{FIN}}] = 0$*. So* $\{\mathcal{F}_{\mathsf{Sig}}\}$ *has each property within* 0 *against* $(\mathbb{Z}_{\gamma,\gamma}, \mathbb{X})$ *in the sense of Definition 5.8, for* $\mathbb{X}$ *the class of all occupants.*

Proof. One preliminary and two verdicts. The preliminary is that both parties named by a verdict must be honest, and it is not bookkeeping:

at a corrupt party the core does not run and the answer is the adversary's, so a mediated VERIFY could report a 0 on a good signature or a 1 on a forgery, and either finalization would fire with $\mathcal{F}_{\mathsf{Sig}}$ having done nothing wrong. Correctness then follows from SIGN recording $\mathtt{Ver}[vk, msg, \sigma] \leftarrow 1$ and VERIFY returning a recorded entry at its first line. Unforgeability splits on whether VERIFY read an entry or computed one: it cannot have computed a 1 under an honest party's key, so the entry was recorded, and only SIGN records one. Take parties honest at the close. Monotonicity makes them honest throughout, so no call the game places at them is mediated, line 4 firing only at a corrupt party and the game's own name lying in $\{0,1\}^*$. Both verdicts ask this of the signer P *and* of the verifier Q, and both need it of both: at a corrupt party the core does not run and the answer is the adversary's, so a mediated VERIFY reports whatever it likes — a 0 on a good signature, or a 1 on a forgery — and either finalization would return 1 with $\mathcal{F}_{\mathsf{Sig}}$ having done nothing wrong.

For correctness, suppose $(\text{SIGN}, \mathsf{P}, msg, \sigma) \in \mathtt{tr}$ with $\sigma \in \Sigma$. The core of SIGN ran, so it sanitized σ against CLEAN_σ and then set $\mathtt{Ver}[vk, msg, \sigma] \leftarrow 1$ for $vk = \mathtt{VK}[\mathsf{P}]$, which GEN fixed once and never changes. Any later $\text{VERIFY}(vk, msg, \sigma)$ finds that entry set and returns it at its first line, so it returns 1; thus $(\text{VER}, \mathsf{Q}, vk, msg, \sigma, 0) \notin \mathtt{tr}$ for every Q, and line 20 gives 0.

For unforgeability, suppose $(\text{VER}, \mathsf{Q}, vk, msg, \sigma, 1) \in \mathtt{tr}$ with $vk = \mathtt{VK}[\mathsf{P}]$ for an honest P. Either VERIFY read a recorded entry or it computed one. It cannot have computed a 1: the last test before writing Ver sets $b \leftarrow 0$ whenever $vk = \mathtt{VK}[\mathsf{P}']$ for some $\mathsf{P}' \notin \mathbf{C}$, which P witnesses. So the entry was recorded, and only SIGN records a 1 — at the party whose key is vk, that party being P since CLEAN_{vk} issues no key twice. Hence $(\text{SIGN}, \mathsf{P}, msg, \sigma) \in \mathtt{tr}$ and line 28 gives 0. □

The bound is not the interesting part; that it is 0 merely says the ideal functionality was written to have these properties, as an ideal functionality should be. The content is what Corollary 5.11 then does with it, and the search reading is what makes that useful: a system π put in place of $\{\mathcal{F}_{\mathsf{Sig}}\}$ wins either finalization with probability at most ε, against every adversary and not only the dummy, and under the search reading that probability *is* the advantage, so ε bounds the property itself. Had the decision reading been taken the same conclusion would read 2ε on $2\Pr[\text{WIN}_{\text{FIN}}] - 1$, which for a game meant to be unwinnable

is no constraint at all — a forger succeeding half the time meets it. That is the calibration rule of Chapter 5 applied where it bites, and why Proposition 5.10 carries both displays: each finalization inherits at the scale its own reading fixes, and these two inherit at ε.

Four conditions come with the corollary and none is discharged by emulation alone. The replacement of $\{\mathcal{F}_{\mathsf{Sig}}\}$ by π must be well-formed; $\{\mathcal{G}\mathfrak{m}\} \cup \pi$ must again be a game system, which asks π's members to pin $\mathcal{G}\mathfrak{m}.\mathit{pid}$ where $\mathcal{F}_{\mathsf{Sig}}$ did, since tightness is part of Definition 5.1; the simulator and the adversary must be refusal-oblivious; and the environments must silence no Z-identity. A realization is thus never separately argued to be correct or unforgeable — it inherits both from the one emulation proof, together with those four checks, and the two finalizations above are what the inheritance is *of*.

One caveat travels with it, and this is where it bites. Corollary 5.11 asks that the cores of the game system place no responsive call, and $\mathcal{F}_{\mathsf{Sig}}$ as written reaches the slot through $\mathcal{A}(\cdot)$, so the corollary applies. Should one adopt Chapter 9's advice and write $\mathcal{A}^{!}$ in GEN, SIGN and VERIFY — which buys locality — the transfer must be argued directly instead, for the reason Remark 9.4 gives. Locality and inheritance are, as things stand, priced against each other.

CHAPTER 16

Public-Key Infrastructure

16.1 Functionality

$\mathcal{F}_{\mathsf{Sig}}$ said plainly that it does not authenticate the key it hands out — VERIFY takes a *vk* of its own, unconnected to any caller, because verification keys are not assumed authenticated (Section 15.1). The functionality $\mathcal{G}_{\mathsf{PKI}}$ below is the setup that closes the gap, in the shape of Canetti's own certification functionality [6]: a single, shared directory binding a key to the party that claims it, so that a *vk* handed to VERIFY can be trusted because it came from here rather than because it was simply believed.

What it does is quickly said. Each party may register one key, and only one: REGISTER takes the key as *input* — unlike GEN, which conjures one from the slot, this is a directory and not a factory — and keeps the first key it is given, sanitized against every key already on file so that no two parties ever hold the same one. The operation RETRIEVE answers anyone with whatever is on file for the party they name, or nothing if no one has registered.

The fields. They are $\mathcal{G}_{\mathsf{Clock}}$'s, and for the same reason. The process id is a constant, $(\mathrm{GPKI}, 0, 0)$, so Definition 1.2 caps the system at one copy: a second directory would be a second notion of who owns which key, exactly what a PKI exists to rule out. $\mathbf{N} := \mathbf{Std} \cup \mathbf{A} \cup \mathbf{Z}$ makes $\mathrm{GLOBAL}(\mathcal{G}_{\mathsf{PKI}})$ hold by the same disjunct as the clock's, and $\mathbf{P}$ is again the universe the directory covers rather than who has used it — registration is state, in `VK`, not membership. One field is simpler than the clock's: $\mathbf{U} := \varnothing$. The operations REGISTER and RETRIEVE place no call at all, the key arriving with the request rather than from the slot, so there is nothing $\mathcal{G}_{\mathsf{PKI}}$ depends on beyond the ambient machinery every functionality already has.

Two borrowed guards. REGISTER takes one line from each of the two

examples before it. The first is $\mathcal{G}_{\mathsf{Clock}}$'s: a caller named A or Z acting for an *honest* party is answered that party's own key and changes nothing, exactly as it is answered the time rather than allowed to tick on an honest party's behalf — an outsider may ask what is on file, not decide what goes there. At a corrupt party the check does not fire, so the call proceeds — the adversary registers on a corrupt party's behalf as freely as it ticks one, and whatever key it submits is what the directory will show, mediation letting exactly this call through unmediated as it does for $\mathcal{F}_{\mathsf{AC}}$'s corrupt sends (Section 19.1). The second is $\mathcal{F}_{\mathsf{Sig}}$'s: the first call to succeed wins, later ones returning the key already on file rather than overwriting it, exactly as GEN never re-issues a key once one exists. Sanitizing here differs from both signature predicates in what it repairs: $\textsc{Clean}_{vk}$ and $\textsc{Clean}_{\sigma}$ each fix a value the *slot* proposed; $\textsc{Clean}_{\text{REG}}$ fixes one the *caller* supplied, which is the more exposed position — an honest party's own key is presumably already good, but a corrupt one's is whatever the adversary wrote in, and the one thing a directory cannot let through is a second party claiming a key the first already holds. A good value always exists here for the same reason it does for $\textsc{Clean}_{vk}$: $\mathcal{K}$ is infinite and the keys excluded are exactly those a bounded run can have recorded, finitely many (footnote of Chapter 12).

Functionality $\mathcal{G}_{\mathsf{PKI}}$

pid $:= (\text{GPKI}, 0, 0)$, **P**, **N** $:=$ **Std** $\cup$ **A** $\cup$ **Z**, **U** $:= \varnothing$, *par* $:= \bot$

INITIALIZE():
1: $\mathtt{VK} : \mathcal{G}_{\mathsf{PKI}}.\mathbf{P} \to \mathcal{K} \cup \{\square\}$
2: $\mathtt{VK}[*] \leftarrow \square$

id.REGISTER(vk) from id′
3: **if** id′.F $\in \{\mathsf{A}, \mathsf{Z}\} \wedge$ id.P $\notin$ **C then**
4: **return** VK[id.P] *// a peek, not a write*
5: **if** VK[id.P] $\neq \square$ **then**
6: **return** VK[id.P]
7: $vk \leftarrow$ SAN[$\textsc{Clean}_{\text{REG}}$]($vk$; VK)
8: VK[id.P] $\leftarrow vk$
9: **return** vk

id.RETRIEVE(P) from id′
10: **return** VK[P]

id.LEAK() from id′
11: **return** VK

$\textsc{Clean}_{\text{REG}}(vk; \mathtt{VK})$:
12: **return** $vk \in \mathcal{K} \wedge \neg\exists\, \mathsf{P} : \mathtt{VK}[\mathsf{P}] = vk$

Three remarks on the code. Registration is recorded against the party, not the caller, for the same reason ticks are: GUARD has already tied the call to id.P by line 2, so whichever machine holds the token when P registers, the key lands on P's own entry. There is no DEREGISTER, and none is added: a real PKI can support revocation, but a key once bound is the simplest directory there is, and revoca-

tion is a genuine addition left for elsewhere rather than a gap papered over. And LEAK answers with the whole table, which is no more than RETRIEVE already gives out one entry at a time — the directory is public at every one of its interfaces, not only this one, so LEAK tells the adversary nothing RETRIEVE, asked enough times, would not.

Set against both examples before it, $\mathcal{G}_{\mathsf{PKI}}$ needs less than either. No value is the simulator's to choose out of nothing, as in GEN — the key arrives with the call — so sanitizing here repairs a caller's input rather than a slot's output, the one place this differs from every SAN before it. And no call leaves the core at all, so there is no notification to be responsive about and no price for Chapter 9 to settle: the two properties below transfer as cheaply as a property can.

16.2 Properties

$\mathcal{G}_{\mathsf{Clock}}$ carried two properties for having two kinds of promise — a local one, that the counter moves as coded, and a global one, that it moves only when everyone honest has agreed. The directory $\mathcal{G}_{\mathsf{PKI}}$ splits the same way, and into two finalizations of one game. FINFIX, *fixity*, says a registered key never changes — the local promise, about one party's own entry. FINBIND, *binding*, says no two parties ever hold the same key — the global one, quantified over all of $\mathcal{G}_{\mathsf{PKI}}.\mathbf{P}$ at once. Both take the search reading of Definition 5.8.

Both are played over the game system $\gamma := \{\mathcal{G}\mathfrak{m}\} \cup \{\mathcal{G}_{\mathsf{PKI}}\}$, with $\mathcal{G}_{\mathsf{PKI}}$ taken as in Section 16.1 save one field: $\mathbf{N} := \{\mathcal{G}\mathfrak{m}.\mathit{pid}\}$, pinned — Remark 6.5's forcing, exactly as for the clock (Section 17.2), and for the identical reason: a property of a global $\mathcal{F}$ is a property against an adversary that shares it, and fixity and binding are both claims about *every* key there is, certifiable only where the game's trace is all there is.

Pinned, the directory is no longer global, so line 3 asks for one session and the experiment's own session 0 meets it. No caller but $\mathcal{G}\mathfrak{m}$ passes line 2: not the environment, not a machine it hosts, and not the adversary's $\mathcal{A}$-identities, refused before line 4 could even apply. So every key $\mathcal{G}_{\mathsf{PKI}}$ ever holds enters through the relay below and is recorded — a game whose verdict is "two parties share a key" has no other way to know it. Inside an interface at $\mathtt{id}$ we write $\mathtt{id}_{\mathcal{G}_{\mathsf{PKI}}}$ for $(\mathcal{G}_{\mathsf{PKI}}.\mathit{pid}, \mathtt{id}.\mathrm{P})$.

Game shell over $\mathcal{G}_{\mathsf{PKI}}$; the finalizations follow

$pid := (\mathcal{G}\mathrm{m}.\mathrm{F}, 0, 0)$, $\mathbf{P} := \mathcal{G}_{\mathsf{PKI}}.\mathbf{P} \cup \{\mathrm{Z}\}$, $\mathbf{N} := \mathbf{Z}$, $\mathbf{U} := \{(\mathcal{G}_{\mathsf{PKI}}, \textsc{serves}_{\mathcal{G}\mathrm{m}})\}$, $par := \bot$

INITIALIZE():
1: DONE $\leftarrow$ FALSE
2: tr $\leftarrow$ ( ) // the trace, a sequence

id.REGISTER(vk) from id′
3: **require** ¬DONE
4: **require** id.P $\neq$ Z
5: $vk' \leftarrow \mathrm{id}_{\mathcal{G}_{\mathsf{PKI}}}.\textsc{FullRegister}(vk)$ as id
6: tr $\leftarrow$ tr $\cdot$ (REG, id.P, vk')
7: **return** vk'

id.LEAK() from id′
8: **return** $\bot$

id.RETRIEVE(P) from id′
9: **require** ¬DONE
10: **require** id.P $\neq$ Z
11: $vk' \leftarrow \mathrm{id}_{\mathcal{G}_{\mathsf{PKI}}}.\textsc{FullRetrieve}(\mathrm{P})$ as id
12: **return** vk'

The finalizations of $\mathcal{G}\mathrm{m}$; the shell over $\mathcal{G}_{\mathsf{PKI}}$ is shared

id.FINFIX() from id′ (fixity)
13: **require** ¬DONE
14: **require** id.P = Z
15: DONE $\leftarrow$ TRUE
16: $\mathbf{C} \leftarrow (\mathrm{C}, \mathrm{Z}).\textsc{FullStatus}()$ as id
17: $w \leftarrow 1$ if $\exists\, \mathrm{P}, vk_1, vk_2 :\ \mathrm{P} \notin \mathbf{C} \wedge vk_1 \neq vk_2$
18: $\quad \wedge$ (REG, P, vk_1) $\in$ tr $\wedge$ (REG, P, vk_2) $\in$ tr;
19: else $w \leftarrow 0$
20: **return** w

id.FINBIND() from id′ (binding)
21: **require** ¬DONE
22: **require** id.P = Z
23: DONE $\leftarrow$ TRUE
24: $\mathbf{C} \leftarrow (\mathrm{C}, \mathrm{Z}).\textsc{FullStatus}()$ as id
25: $w \leftarrow 1$ if $\exists\, \mathrm{P}, \mathrm{Q}, vk :\ \mathrm{P} \notin \mathbf{C} \wedge \mathrm{Q} \notin \mathbf{C} \wedge \mathrm{P} \neq \mathrm{Q}$
26: $\quad \wedge$ (REG, P, vk) $\in$ tr $\wedge$ (REG, Q, vk) $\in$ tr;
27: else $w \leftarrow 0$
28: **return** w

Two things about the code. The operation RETRIEVE is relayed and logged though neither verdict reads its trace: a game that could not look anyone's key up would not be exercising $\mathcal{G}_{\mathsf{PKI}}$ at all, and leaving the interface whole costs nothing, exactly as $\mathcal{G}_{\mathsf{Clock}}$'s shell relayed READ for games that read only the mutators. And FINFIX's verdict does not need vk_1 and vk_2 ordered, unlike steadiness's reading of its trace: idempotence means at most one of the two calls that produced them can be the write, the other only a later read of it, and either order loses

the game identically, a changed key being a changed key whichever one came first.

Proposition 16.1 ($\mathcal{G}_{\mathsf{PKI}}$ has fixity)**.** *For every $\mathcal{Z} \in \mathbb{Z}_{\gamma,\gamma}$ and every occupant $\mathcal{X}$ of the adversary slot,* $\Pr[\text{WIN}_{\text{FINFIX}}] = 0$*. So $\{\mathcal{G}_{\mathsf{PKI}}\}$ has fixity within 0 against $(\mathbb{Z}_{\gamma,\gamma}, \mathbb{X})$ in the sense of Definition 5.8, for $\mathbb{X}$ the class of all occupants.*

Proof. Suppose $(\text{REG}, \mathsf{P}, vk_1), (\text{REG}, \mathsf{P}, vk_2) \in \mathtt{tr}$ with $\mathsf{P} \notin \mathbf{C}$. By tightness both were logged at line 5, each running $\mathcal{G}_{\mathsf{PKI}}$.REGISTER's own core at $\mathtt{id}.\mathsf{P} = \mathsf{P}$: the relay's own name is $\mathcal{G}\mathsf{m}$'s, not A or Z, so line 4 never intercepts it. The core's only write is line 8, reached only while $\mathtt{VK}[\mathsf{P}] = \square$, and once reached it fixes $\mathtt{VK}[\mathsf{P}]$ to some sanitized value for good — every later call at P finds $\mathtt{VK}[\mathsf{P}] \neq \square$ and returns line 6's value, the same one, unchanged. So the two calls producing vk_1 and vk_2 agree: whichever ran first wrote the value, and the other only read it back. The verdict at line 17 gives 0. □

Proposition 16.2 ($\mathcal{G}_{\mathsf{PKI}}$ has binding)**.** *For every $\mathcal{Z} \in \mathbb{Z}_{\gamma,\gamma}$ and every occupant $\mathcal{X}$ of the adversary slot,* $\Pr[\text{WIN}_{\text{FINBIND}}] = 0$*. So $\{\mathcal{G}_{\mathsf{PKI}}\}$ has binding within 0 against $(\mathbb{Z}_{\gamma,\gamma}, \mathbb{X})$ in the sense of Definition 5.8, for $\mathbb{X}$ the class of all occupants.*

Proof. Suppose $(\text{REG}, \mathsf{P}, vk), (\text{REG}, \mathsf{Q}, vk) \in \mathtt{tr}$ with $\mathsf{P}, \mathsf{Q} \notin \mathbf{C}$ and $\mathsf{P} \neq \mathsf{Q}$. By Proposition 16.1's argument $vk = \mathtt{VK}[\mathsf{P}]$ and $vk = \mathtt{VK}[\mathsf{Q}]$ from the moment each was first written onward. Say Q's write came second, the case P's symmetric; at that moment REGISTER's line 8 ran $vk \leftarrow \text{SAN}[\text{CLEAN}_{\text{REG}}](\cdot; \mathtt{VK})$ with $\mathtt{VK}[\mathsf{P}]$ already set, and SAN's guarantee (Chapter 12) is that its output satisfies $\text{CLEAN}_{\text{REG}}$ whatever the input — membership in $\mathcal{K}$ and freshness against every key already on file, P's among them. So Q's write cannot equal $\mathtt{VK}[\mathsf{P}]$, contradicting $vk = \mathtt{VK}[\mathsf{P}] = \mathtt{VK}[\mathsf{Q}]$. The verdict at line 25 gives 0. □

Both transfer by Corollary 5.11 with nothing extra to argue. The hypothesis it asks — that the game system's cores place no responsive call — is met the cheap way: $\mathcal{G}_{\mathsf{PKI}}$'s cores place no call of any kind, responsive or not, so Remark 9.4's caveat never arises and dummy completeness (Section 4.4) is untouched. Unlike $\mathcal{G}_{\mathsf{Clock}}$, whose notification bought locality at inheritance's expense, and unlike $\mathcal{F}_{\mathsf{Net}}$ and $\mathcal{F}_{\mathsf{AC}}$,

whose slot calls priced their own transfer, $\mathcal{G}_{\mathsf{PKI}}$ pays nothing: a system π put in place of $\{\mathcal{G}_{\mathsf{PKI}}\}$ inherits fixity and binding directly, the four conditions of Section 15.2's closing paragraph being the only ones left to discharge. A directory that asks the adversary nothing has nothing to spend.

CHAPTER 17

Global Clock

17.1 Functionality

$\mathcal{F}_{\mathsf{Sig}}$ is local and randomized, and most of what was said about it concerned the simulator. The second example is its opposite on both counts. A clock functionality was introduced by Katz, Maurer, Tackmann and Zikas to model synchrony in UC [11]; recast by Badertscher, Maurer, Tschudi and Zikas as a *shared* setup around a monotone counter [3], it has become the canonical global functionality, the one the composable analyses of Bitcoin and its successors keep time by. The clock $\mathcal{G}_{\mathsf{Clock}}$ below is that functionality in the language of this paper.

What it does is quickly said. The counter `now` starts at 0 and only ever steps forward. Anyone the clock serves may read it. Parties enter and leave the round structure through REGISTER and DEREGISTER, and the counter steps when every honest registered party has asked it to: UPDATE records a tick against the calling party, and once the honest registered ticks are all in, the test on line 17 advances the counter and clears them. A round ends exactly when the last honest party in it is done, and the adversary can neither stall the round nor close it early — how the code refuses each is below. Each recorded tick is reported to the adversary before UPDATE returns, on line 20, *responsively* (Chapter 9) and with the answer discarded; what the notification buys and what it costs is settled at the end of Section 17.2. Every operation answers the counter as it stands on return, so the tick that closes a round learns the new time and the ticks before it the old; a party that must know where it stands reads again.

The fields. The process id is a constant, $(\text{GCLOCK}, 0, 0)$, where $\mathcal{F}_{\mathsf{Sig}}$'s was a parameter. Definition 1.2 then puts at most one copy in any system, which is the point: global means one instance, shared, and a second clock would be a second time. $\mathbf{N} := \mathbf{Std} \cup \mathbf{A} \cup \mathbf{Z}$ makes

GLOBAL($\mathcal{G}_{\mathsf{Clock}}$) hold by its first disjunct, so line 3 of GUARD waives the session check and every session reaches the one instance — the very separation the session component exists to make, deliberately not made here. A protocol that keeps time declares ($\mathcal{G}_{\mathsf{Clock}}$, SERVES) in its $\mathbf{U}$, and the global disjunct of Definition 1.3 is what lets members of every session meet the entry at the shared copy. And $\mathbf{U} := \{(\mathcal{A}, \text{SERVES})\}$ for the one call the code places, the notification of line 20. And $\mathbf{P}$ stays a parameter but as the universe rather than the membership: it fixes the parties an interface exists for, while who is *in* the round structure at a given moment is state, entered and left through REGISTER and DEREGISTER as in [3] — dynamic membership is one more table, not a new kind of field.

Reading and ticking. $\mathbf{N}$ is a field of the functionality, not of an operation, so one set governs every core, and the set above is the right one for READ: the adversary and the environment may ask the time — [3] lets both, and the environment reading the shared clock is much of what makes a global setup global. It is too wide for the three mutators, where an adversary free to act for a party could close a round the party had not finished, or drop it from the round structure altogether. The refinement sits as the first line of each core: a caller named $\mathcal{A}$ or $\mathcal{Z}$ acting for an *honest* party is answered the time and changes nothing, while at a corrupt party it passes — the adversary registers, deregisters and ticks for the parties it owns, as in [3], and those ticks are inert where it matters, line 17 never reading a corrupt party's flag. One consequence deserves naming: while no honest party is registered the test of line 17 is vacuous, so the adversary may register a corrupt party and tick it freely — until the first honest registration, time is the adversary's to spend, there too as in [3].

The quantifier of line 17 reads "every honest registered party", and it is worth seeing how little of it $\mathcal{G}_{\mathsf{Clock}}$ enforces itself. At a corrupt party a standard caller's UPDATE is handed to the adversary by line 4 of the full interface before the core runs, and the adversary's own call at the same party is admitted by line 13; either way a corrupt party's ticks are the adversary's to give or withhold, and the $\forall$ — which reads the register by the convention of Section 3.1 — ignores them regardless. Corrupt parties therefore neither hold a round open nor help close one. At an honest party both routes shut: MEDIATE does not fire, and line 13 turns the observers' ticks into reads. Monotonicity adds the last piece: a party corrupted after ticking leaves a stale flag behind, but it is

outside the quantifier from then on, and the next advance clears it.

Functionality $\mathcal{G}_{\mathsf{Clock}}$

pid := (GCLOCK, 0, 0), **P**, **N** := **Std** ∪ **A** ∪ **Z**, **U** := {($\mathcal{A}$, SERVES)}, *par* := ⊥

INITIALIZE():
1: now ← 0
2: reg, tick : $\mathcal{G}_{\mathsf{Clock}}$.**P** → {0, 1}
3: reg[∗] ← 0; tick[∗] ← 0

id.REGISTER() from id′
4: **if** id′.F ∈ {A, Z} ∧ id.P ∉ **C then**
5: **return** now
6: reg[id.P] ← 1; tick[id.P] ← 0 *// a fresh member owes a tick*
7: **return** now

id.DEREGISTER() from id′
8: **if** id′.F ∈ {A, Z} ∧ id.P ∉ **C then**
9: **return** now
10: reg[id.P] ← 0
11: **return** now

id.UPDATE() from id′
12: **if** id′.F ∈ {A, Z} ∧ id.P ∉ **C then**
13: **return** now *// reading, not ticking*
14: **if** reg[id.P] = 0 **then**
15: **return** now *// not in the round structure*
16: tick[id.P] ← 1
17: **if** ∀ P ∈ $\mathcal{G}_{\mathsf{Clock}}$.**P** \ **C** : reg[P] = 1 ⇒ tick[P] = 1 **then**
18: now ← now + 1
19: tick[∗] ← 0 *// the round is over*
20: $\mathcal{A}^{!}$(id.UPDATE, now) *// notify; the answer is discarded*
21: **return** now

id.READ() from id′
22: **return** now

id.LEAK() from id′
23: **return** (now, reg, tick)

Three remarks on the code. Registrations and ticks are recorded against the *party*, not the caller: any machine acting for P may register or tick P, GUARD having already tied the call to the party on line 2. [3] instead has every registered party and functionality tick separately before the round closes; that discipline, if wanted, is tables indexed by identity rather than party, and nothing else changes. The notification is UPDATE's alone; [3] tells the adversary about registrations and departures too, and extending line 20's pattern to the other two mutators is mechanical. And LEAK returns the whole state, a clock keeping no secrets: the counter is anyone's to read anyway, and the tables say only who is in the round and who is done with it.

Set against $\mathcal{F}_{\mathsf{Sig}}$, most of what the signature example needed is still absent, each absence legible in the code. No value is the simulator's to choose — the notification's answer is discarded — so there is nothing to sanitize. And where $\mathcal{F}_{\mathsf{Sig}}$ reached the slot through $\mathcal{A}(\cdot)$ and re-tested its tables against what could happen while it hung, the clock's one call is responsive by construction: every mutation of UPDATE precedes line 20, and RESPOND keeps the token from leaving, so no instance is ever re-entered mid-operation and there is nothing to re-test — enforced against every occupant of the slot, where Chapter 15 could only advise

a simulator class. What the notification costs is recorded where it bites: Section 17.2 plays two properties over the clock and prices it there.

17.2 Properties

Section 15.2 instanced Definition 5.1 once; the clock is worth a second, not least because time is the one thing in this paper whose *order* is the content. We give one game $\mathcal{G}\mathfrak{m}$ over $\mathcal{G}_{\mathsf{Clock}}$ carrying two properties: FINSTEAD, *steadiness*, saying that the counter moves as the code promises — forward, by one, and only when pushed — and FINUNA, *unanimity*, saying that it moves only when every honest registered party has pushed. Both take the search reading of Definition 5.8, winning either being meant to be impossible rather than hard.

They are played over the game system $\gamma := \{\mathcal{G}\mathfrak{m}\} \cup \{\mathcal{G}_{\mathsf{Clock}}\}$, with the clock taken as in Chapter 17 save one field: $\mathbf{N} := \{\mathcal{G}\mathfrak{m}.\mathit{pid}\}$, pinned. The change is forced. Tightness at $\{\mathcal{G}\mathfrak{m}\}$, which Definition 5.1 requires, asks the members under test to admit whole process ids, and a bare **Std** is exactly what Remark 6.5 refuses; Definition 5.7 says why in operational terms — an environment shares a global functionality, touching it as anyone may, and a property of a global $\mathcal{F}$ is a property against an adversary that shares it. For these two properties sharing is not a nuisance but the negation of the claim: steadiness and unanimity are assertions about *all* the ticks there are, and a game can certify them only where its trace is all there is. So the clock is played privately, and the properties certified are of its *code*, which sharing does not change — the situation of Section 15.2 exactly, where the $\mathcal{F}_{\mathsf{Sig}}$ that ships names its protocol and the $\mathcal{F}_{\mathsf{Sig}}$ under test names the game.

Pinned, the clock is no longer global, so line 3 of GUARD asks for one session; the experiment sits in session 0 and the clock's process id carries $s = 0$, so the check passes. And no caller but $\mathcal{G}\mathfrak{m}$ now passes line 2 at the clock: not the environment or a machine it hosts, whose claims lie outside a pinned $\mathbf{N}$, and not the adversary, whose A-identities are refused before line 13 could even answer them the time. So the counter changes only inside line 15 below and the membership only inside the REGISTER and DEREGISTER relays beside it, and every change is recorded — a game whose verdict is "the time moved when no one pushed" has no other way to know. Inside an interface at $\mathtt{id}$ we write $\mathtt{id}_{\mathcal{G}_{\mathsf{Clock}}}$ for $(\mathcal{G}_{\mathsf{Clock}}.\mathit{pid}, \mathtt{id}.\mathrm{P})$, and every trace entry carries the answer τ the clock gave.

One line has no analogue in Section 15.2: the relays refuse the root. There the relays at the root addressed a party $\mathcal{F}_{\mathsf{Sig}}$ does not serve and were answered REJ at line 1, which the verdicts, existential over well-shaped tuples, never read. Steadiness reads *every* entry, so a REJ of the game's own making cannot be allowed into the trace; refused at the relay, nothing is appended, and every entry in `tr` is an answer the clock actually gave. Against a replacement this bites, and rightly: a π answering an honest read anything but a number loses steadiness on the spot, a clock that cannot say the time being broken however it moves.

That line is the first thing a shared shell asks for that a single property would not. Unanimity does not need it — its verdict is existential, like $\mathcal{F}_{\mathsf{Sig}}$'s, and would read past a REJ as those did — so the refusal is steadiness's demand alone, and the shell carries it because both finalizations hang off the shell. A shell is built to the strictest of the verdicts it serves, and the cost of that is paid by the others: unanimity is played on a trace cleaner than it needs. It is a cost worth naming, because it is the one thing sharing a shell can take away. Where a property would be *weakened* by another's demand rather than merely over-served, the answer is a second game and not a second finalization.

Game shell over $\mathcal{G}_{\text{Clock}}$; the finalizations follow

pid := $(\mathcal{G}_{\mathfrak{m}}.\mathrm{F}, 0, 0)$, **P** := $\mathcal{G}_{\text{Clock}}.\mathbf{P} \cup \{\mathrm{Z}\}$, **N** := **Z**, **U** := $\{(\mathcal{G}_{\text{Clock}}, \text{SERVES}_{\mathcal{G}_{\mathfrak{m}}})\}$, *par* := $\bot$

INITIALIZE():
1: DONE ← FALSE
2: tr ← ( ) // the trace, a sequence

id.REGISTER() from id′
3: **require** ¬DONE
4: **require** id.P ≠ Z // the root only finalizes
5: τ ← $\text{id}_{\mathcal{G}_{\text{Clock}}}$.FULLREGISTER() as id
6: tr ← tr · (REG, id.P, τ)
7: **return** τ

id.DEREGISTER() from id′
8: **require** ¬DONE
9: **require** id.P ≠ Z
10: τ ← $\text{id}_{\mathcal{G}_{\text{Clock}}}$.FULLDEREGISTER() as id
11: tr ← tr · (DEREG, id.P, τ)
12: **return** τ

id.TICK() from id′
13: **require** ¬DONE
14: **require** id.P ≠ Z
15: τ ← $\text{id}_{\mathcal{G}_{\text{Clock}}}$.FULLUPDATE() as id
16: tr ← tr · (TICK, id.P, τ)
17: **return** τ

id.READ() from id′
18: **require** ¬DONE
19: **require** id.P ≠ Z
20: τ ← $\text{id}_{\mathcal{G}_{\text{Clock}}}$.FULLREAD() as id
21: tr ← tr · (READ, id.P, τ)
22: **return** τ

id.LEAK() from id′
23: **return** ⊥

The finalizations of $\mathcal{G}\mathfrak{m}$; the shell over $\mathcal{G}_{\mathsf{Clock}}$ is shared

id.FINSTEAD() from id′ (steadiness)
24: **require** ¬DONE
25: **require** id.P = Z
26: DONE ← TRUE
27: parse tr as $((op_j, P_j, \tau_j))_{j=1}^{n}$; $\tau_0 := 0$
28: $w \leftarrow 1$ if $\exists\, j \in \{1, \ldots, n\}$: $\tau_j \notin \{\tau_{j-1}, \tau_{j-1} + 1\}$
29: $\quad \vee\ (\tau_j = \tau_{j-1} + 1 \wedge op_j \neq \text{TICK})$;
30: else $w \leftarrow 0$
31: **return** w

id.FINUNA() from id′ (unanimity)
32: **require** ¬DONE
33: **require** id.P = Z
34: DONE ← TRUE
35: **C** ← (C, Z).FULLSTATUS() as id
36: $w \leftarrow 1$ if $\exists\, t \geqslant 0\ \exists\, P \in \mathcal{G}_{\mathsf{Clock}}.\mathbf{P} \setminus \mathbf{C}$:
37: $\quad (\exists\, \tau_r \leqslant t : (\text{REG}, P, \tau_r) \in \mathtt{tr}) \wedge (\text{DEREG}, P, \cdot) \notin \mathtt{tr}$
38: $\quad \wedge\ (\exists\, \tau > t : (\cdot, \cdot, \tau) \in \mathtt{tr}) \wedge (\forall\, \tau' \in \{t, t+1\} : (\text{TICK}, P, \tau') \notin \mathtt{tr})$;
39: else $w \leftarrow 0$
40: **return** w

The verdicts, in words. The finalization FINSTEAD returns 1 on an answer below its predecessor, an answer more than one above it, or an increase anywhere but at a tick: reads, registrations and departures must find the time where the last answer left it. A tick answered $\tau_{j-1} + 1$ is allowed — it is the tick that closed the round — and $\tau_0 := 0$ is INITIALIZE's word taken as the first observation, so a first read answered 1 is already a counter that moved before anyone pushed. The finalization FINUNA returns 1 if the counter passed some t although some party — honest at the close, registered by round t and never released — has no tick answered t or $t + 1$: no tick placed during round t, that is, the answer $t + 1$ covering the one that closed it. The registration conjuncts read the trace, not the clock's tables, so they too name only what the game itself did.

Two contrasts with Section 15.2, both instructive. There the trace was a sequence whose order no verdict read; here FINSTEAD reads little else, time being an order and not much besides. And there both verdicts had to ask $P \notin \mathbf{C}$ of every party they named; FINSTEAD asks nothing of anyone, because at a corrupt party the game's own wrapper intercepts before the core runs and records (Remark 5.4), so every entry in tr is an honest core's answer already. The one verdict that must

name honesty is FINUNA's, whose claim concerns parties that did *not* call, and it names it at the close — monotonicity (Section 3.5) again making that the stronger reading. That the two differ so — one reading the register, the other not; one reading order, the other membership — is what a shared shell is for: it is the oracles they have in common, not the verdicts, that made them one game.

Remark 17.1 (Agreement is steadiness read across parties)**.** The property a shared clock is usually wanted for is *agreement*: honest parties hold a common view of the time. Here it is not a third game but a corollary of the verdict above, and by design. Line 28 quantifies over the trace *interleaved*, not party by party: between two consecutive TICK entries every answer must equal its predecessor, whoever asked, so two honest parties reading between the same two ticks are answered the same value or the game is won. Stated per party — each party's own subsequence steady — the verdict would be strictly weaker, and agreement a separate property: a replacement could then answer P with 5 and Q with 3 in adjacent activations and fail neither property. What lets one verdict carry both claims is the execution model: a single token totally orders the trace, so "simultaneous reads" do not exist and a common view reduces to the consistency of adjacent entries. Two limits are worth naming. Agreement is among honest parties only, a corrupt party's answers being the adversary's (Remark 5.4); and it says nothing of a party that does not ask, whose view may grow stale while the clock runs ahead — staleness of the party, not disagreement of the clock, and nothing a functionality could promise a caller that never calls. A *drift* variant — views agreeing within some Δ, the loosely synchronized clocks of [11] — would be a genuine weakening of line 28, and under the present shape it is a third finalization rather than a third game: it reads the same trace of the same relays and differs in the verdict alone, which is exactly the test for hanging it off this shell.

Proposition 17.2 ($\mathcal{G}_{\mathsf{Clock}}$ has both properties)**.** *For* FIN *either of* FINSTEAD *and* FINUNA*, for every* $\mathcal{Z} \in \mathbb{Z}_{\gamma,\gamma}$ *and every occupant* $\mathcal{X}$ *of the adversary slot,* $\Pr[\mathrm{WIN}_{\mathrm{FIN}}] = 0$*. So* $\{\mathcal{G}_{\mathsf{Clock}}\}$ *has each property*

within 0 *against* $(\mathbb{Z}_{\gamma,\gamma}, \mathbb{X})$ *in the sense of Definition 5.8, for* $\mathbb{X}$ *the class of all occupants.*

Proof. One invariant carries both properties: the counter changes only inside a tick the trace records and by exactly one, and membership only inside recorded registrations and departures. Establishing it is the first paragraph's work, and it turns on three things — tightness, so no caller but the game reaches the clock's cores; the game reaching them only at honest parties, so no answer is mediated; and no core of γ handing the token to anything that could run one, the single call the clock's cores place being responsive. Entries therefore append in execution order, each carrying the counter as its call returned. Steadiness then follows by comparing consecutive entries, since between them nothing runs. Unanimity is the longer half: it assumes the four conjuncts of the verdict and derives a contradiction from the invariant, using monotonicity of corruption to keep the party honest throughout. Everything rests on one invariant: the counter changes only inside a tick the trace records, and by exactly one; the membership only inside recorded registrations and departures. Both tables are written only in the cores of Chapter 17 — the counter at one line of UPDATE, stepping by one when the test of line 17 passes, `reg` in REGISTER and DEREGISTER — and those cores run only on calls passing GUARD: with $\mathbf{N}$ pinned no caller but $\mathcal{G}\mathfrak{m}$ passes line 2, and $\mathcal{G}\mathfrak{m}$ calls only through the relays, at honest parties — at a corrupt one its own wrapper intercepted a step earlier and no core of γ ran — so no call is mediated and each core's answer is the answer recorded. Moreover no core of γ hands the token to anything that could run one: the one call the clock's cores place is the notification of line 20, which is responsive — RESPOND refuses every call its answerer would place, whatever occupies the slot, and every mutation of UPDATE precedes it in any case — and the relays' one call is on the clock, that chain returning with no other machine of γ having run in between. Entries therefore append in execution order, each carrying the counter as its call returned.

For steadiness: before the first entry the counter is INITIALIZE's $0 = \tau_0$. Between consecutive entries no core of γ runs, so the counter at entry j differs from that at entry $j-1$ by the steps taken inside entry j's own call: none for a read, a registration or a departure, none or one for a tick. Hence $\tau_j \in \{\tau_{j-1}, \tau_{j-1}+1\}$ throughout, with the increment only at a tick, and line 28 gives 0.

For unanimity: suppose the four conjuncts of line 36 hold of some $\mathtt{t}$ and some $\mathtt{P} \in \mathcal{G}_{\mathsf{Clock}}.\mathbf{P} \setminus \mathbf{C}$ at the close — honest throughout, by monotonicity. A recorded registration answered $\tau_r \leqslant \mathtt{t}$ set $\mathtt{reg}[\mathtt{P}] \leftarrow 1$; only the cores of REGISTER and DEREGISTER write $\mathtt{reg}$, the game placed no DEREGISTER at $\mathtt{P}$ — there is no DEREG entry — and the observers cannot reach an honest party, so $\mathtt{reg}[\mathtt{P}] = 1$ from that moment on. Some entry answers $\tau > \mathtt{t}$, and by the invariant the answers are values the counter took, so it stepped from $\mathtt{t}$ to $\mathtt{t}+1$ inside some recorded tick; its line 17 read the flags of every party honest at that moment and registered then — $\mathtt{P}$ among them, the register only ever growing and $\tau_r \leqslant \mathtt{t}$ — so $\mathtt{tick}[\mathtt{P}] = 1$. That flag is written only by the core of UPDATE at $\mathtt{P}$ and cleared at every step of the counter and at every re-registration, so it was set while the counter stood at $\mathtt{t}$, by a tick of the game at $\mathtt{P}$; that tick answered $\mathtt{t}$, or $\mathtt{t}+1$ if it was itself the one that closed the round, and it was recorded. So $(\text{TICK}, \mathtt{P}, \tau') \in \mathtt{tr}$ for some $\tau' \in \{\mathtt{t}, \mathtt{t}+1\}$, contradicting the fourth conjunct, and line 36 gives 0. □

As with $\mathcal{F}_{\mathsf{Sig}}$, the bound being 0 says only that the specification was written to have its properties; the content is the transfer, and here the notification of line 20 collects its price. Corollary 5.11 asks that the cores of the game system place no responsive call, and that line is one, so the transfer to a system π put in place of $\{\mathcal{G}_{\mathsf{Clock}}\}$ must be argued directly, for the reason Remark 9.4 gives — and the same hypothesis guards Theorem 4.29, so a system built over the notifying clock forgoes dummy completeness with it. A clock without the notification — delete line 20 and the **U** entry — has both properties by the same proof and inherits them through the corollary with nothing to discharge.

The choice, note, is only *which* price, not whether. A notification through the ordinary $\mathcal{A}(\cdot)$, the route $\mathcal{F}_{\mathsf{Sig}}$ took, would hand the token to an adversary free to drive nested ticks before answering; answers would then append out of trace order, and steadiness would fail of the ideal clock itself — the two verdicts above are properties of $\mathcal{G}_{\mathsf{Clock}}$ only because line 20 cannot be acted on before UPDATE returns. So where $\mathcal{F}_{\mathsf{Sig}}$ chose inheritance and paid with locality, a notifying clock must choose locality and pay with inheritance; the clock that pays neither is the one that tells the adversary nothing.

CHAPTER 18

Δ-Delayed Network

18.1 Functionality

The clock exists to be read, and the first reader is a network. Communication with a bounded delay is [11]'s; the diffusion form, delays measured on a shared clock, is the network the composable blockchain analyses run over [3]. The network $\mathcal{F}_{\mathsf{Net}}$ below is that functionality in the language of this paper — and the first example to *use* the clock rather than be it. Chapter 1 named a diffusion channel $\mathcal{F}_{\mathsf{Diffuse}}$ and owed no code; $\mathcal{F}_{\mathsf{Net}}$ is what such a channel looks like with time attached.

What it does is quickly said. The operation SEND timestamps the message by the clock — through $\mathtt{id}_{\mathcal{G}_{\mathsf{Clock}}} := (\mathcal{G}_{\mathsf{Clock}}.\mathit{pid}, \mathtt{id}.\mathrm{P})$, the clock's interface at the party being served, the notation of Section 17.2 — stores it, tells the adversary — the network is public — and answers a message id. The operation FETCH asks the adversary what to reveal early, then answers everything released to the fetching party together with everything Δ old. The division of power is the point: the adversary chooses *when* within Δ, and to whom first, and may add traffic of its own; what it cannot do is withhold an honest message past its time, and that bound sits in FETCH's last line, where no answer of the slot reaches it. A message sent at time τ is in every fetch from time $\tau + \Delta$ on, whoever the adversary favoured first.

The fields. They follow $\mathcal{F}_{\mathsf{Sig}}$'s pattern, not the clock's. *pid*, **P** and **N** are parameters: a network is local, one instance per session, its **N** naming the protocol it serves in the local pattern of Chapter 1 — and every instance reads the one shared clock, which is what makes "time τ" comparable across sessions. $\mathbf{U} := \{(\mathcal{G}_{\mathsf{Clock}}, \textsc{serves}), (\mathcal{A}, \textsc{serves})\}$: the first entry requires the clock to serve every party the network does and is met, whatever session the network sits in, at the shared copy

by the global disjunct of Definition 1.3 — a system containing $\mathcal{F}_{\mathsf{Net}}$ is well-formed only with the clock alongside — and the second covers the two slot calls below. *par* := Δ, a parameter a line of code actually reads, as $\mathcal{F}_{\mathsf{Rand}}$'s $\mathfrak{n}$ is (Chapter 13).

Scheduling is answered, not called. [3] lets the adversary deliver a message early by acting on the network of its own accord. Transcribed literally that is an interface the adversary *calls*, so a local **N** containing **A** — reachable from the slot in session 0 and in no other, the one shape Definition 7.2 excludes and relocation cannot carry. So the slot is asked instead: FETCH hands the adversary the time and takes back a pair (S, I) — releases among the stored ids, injections of fresh traffic — sanitized inline, the good repair being canonical, as in Chapter 15. Both slot calls are responsive, the clock's discipline. The operation SEND mutates before it notifies; FETCH cannot, its mutations reading the answer, and responsiveness is what makes the difference immaterial: the token comes straight back, nothing else runs inside either call, and $\mathcal{F}_{\mathsf{Sig}}$'s re-entrance re-tests stay unnecessary here too.

Corruption, without a line of code. A corrupt party's SEND and FETCH are intercepted by line 4 of the full interface before the cores run, so the buffer holds honest sends only, and the cores' clock reads are never mediated — a timestamp is always the clock's own word. What a corrupt party "sends", the adversary supplies from the receiving side instead: the injections of line 16 enter the buffer with sender $\perp$ and timestamp ∞, released to the fetching party alone — no age, and so no ripening into line 20's guarantee, which is for the honest and never conscripts the adversary's own traffic.

Functionality $\mathcal{F}_{\text{Net}}$
pid, **P**, **N**, **U** $:= \{(\mathcal{G}_{\text{Clock}}, \text{SERVES}), (\mathcal{A}, \text{SERVES})\}$, *par* $:= \Delta$

INITIALIZE():
1: `ctr` $\leftarrow 0$
2: `buf` $\leftarrow \varnothing$ *// entries* (m, P, msg, τ)
3: `rel` $: \mathbb{N} \times \mathcal{F}_{\text{Net}}.\mathbf{P} \to \{0,1\}$
4: `rel`$[*,*] \leftarrow 0$ *// what is released to whom*

id.SEND(*msg*) from `id'`
5: $\tau \leftarrow$ `id`$_{\mathcal{G}_{\text{Clock}}}$.FULLREAD() as `id`
6: $m \leftarrow$ `ctr`; `ctr` $\leftarrow$ `ctr` $+ 1$
7: `buf` $\leftarrow$ `buf` $\cup \{(m, \text{id.P}, msg, \tau)\}$
8: $\mathcal{A}^{!}(\text{id.SEND}, m, msg)$ *// the network is public*
9: **return** m

id.LEAK() from `id'`
10: **return** (`buf`, `rel`)

id.FETCH() from `id'`
11: $\tau \leftarrow$ `id`$_{\mathcal{G}_{\text{Clock}}}$.FULLREAD() as `id`
12: $(S, I) \leftarrow \mathcal{A}^{!}(\text{id.FETCH}, \tau)$ *// a non-pair reads as* $(\varnothing, \varnothing)$
13: $S \leftarrow S \cap \{m : (m, \cdot, \cdot, \cdot) \in$ `buf`$\}$;
14: $I \leftarrow I \cap \mathcal{M}$ *// sanitized inline*
15: `rel`$[m, \text{id.P}] \leftarrow 1$ for each $m \in S$
16: for each $msg \in I$:
17: `buf` $\leftarrow$ `buf` $\cup \{($`ctr`$, \bot, msg, \infty)\}$;
18: `rel`$[$`ctr`$, \text{id.P}] \leftarrow 1$; `ctr` $\leftarrow$ `ctr` $+ 1$
19: **return** $\{(m, msg) : (m, P', msg, \tau_s) \in$ `buf`
20: $\wedge$ (`rel`$[m, \text{id.P}] = 1 \vee \tau_s + \Delta \leqslant \tau)\}$

Three remarks on the code. Fetched pairs carry no sender: the network is an unauthenticated diffusion, [3]'s, and the injections are what unauthenticated *means* — a fetch may contain messages no one sent, the wire being the adversary's to write. An authenticated *diffusion* is one field and one line away — return the sender the buffer already keeps, drop the injections. Authentication with addressing is a further step and a different object; Chapter 19 gives it. The table `rel` is sticky: what one fetch reveals no later fetch may hide, so a party's view of the network only grows — and line 20 reads `rel` *or* the deadline, so the adversary's silence past Δ is simply ignored rather than punished. And LEAK returns the whole state, the network keeping no secrets: its contents were told to the adversary at line 8 as they arrived, and the release table is the slot's own doing.

18.2 Properties

Section 18.1 left three properties unformulated: delivery by Δ, monotone views, and no delivery before sending. The last needs no game at all: an id not yet in `buf` is nothing S can name, line 20 intersecting with `buf` before anything else runs, so no occupant of the slot can ever release what has not yet been sent. Monotone views we still leave unformulated, being a property of `rel` alone and orthogonal to timing. What we formalize here is the first, under the name FINLIVE, *liveness*: a message ripened by Δ is missing from no later honest fetch. And

we formalize a property $\mathcal{F}_{\mathsf{Net}}$ was never claimed to have, FINAUTH, *authentication*: a message an honest party fetches was sent by someone. Both take the search reading of Definition 5.8.

They are two finalizations of one game, and this is where that shape earns the most. The network $\mathcal{F}_{\mathsf{Net}}$ has the first outright and, being an unauthenticated diffusion by design — Section 18.1's own remark on the injections of line 16 — provably lacks the second: one shell, one set of oracles, one trace, and two verdicts read off it of which one is unwinnable and the other is won with probability 1. Under two separate games that would be two statements about two objects; here it is one object measured twice. What keeps the measurements apart is Remark 5.3's single flag: an environment that calls FINAUTH has closed the game against FINLIVE, so no run is judged on both, and nothing about the pair is a conjunction.

Both are played over the game system $\gamma := \{\mathcal{G}\mathfrak{m}\} \cup \{\mathcal{F}_{\mathsf{Net}}, \mathcal{G}_{\mathsf{Clock}}\}$, with $\mathcal{F}_{\mathsf{Net}}.\mathbf{N} := \{\mathcal{G}\mathfrak{m}.\mathit{pid}\}$ pinned and $\mathcal{F}_{\mathsf{Net}}.\mathbf{P} \subseteq \mathcal{G}\mathfrak{m}.\mathbf{P}$ — Section 15.2's tightness argument transplanted whole, since $\mathcal{F}_{\mathsf{Net}}$'s own fields already follow $\mathcal{F}_{\mathsf{Sig}}$'s pattern rather than the clock's (Section 18.1). The interface box below states $\mathcal{G}\mathfrak{m}.\mathbf{U}$ in full. No caller but $\mathcal{G}\mathfrak{m}$ passes GUARD at $\mathcal{F}_{\mathsf{Net}}$: not the environment, not the adversary, not a machine hosted at an invented instance of the game's name. So every message $\mathcal{F}_{\mathsf{Net}}$'s core ever stores, and every answer it ever returns to a FETCH, passes through the relays below and is recorded — a game whose verdict names "sent by someone" has no other way to know it. $\mathcal{F}_{\mathsf{Net}}$'s own dependence on the clock and the slot is unaffected by any of this and is met exactly as it was for $\mathcal{F}_{\mathsf{Net}}$ itself, the clock at the shared copy by the global disjunct of Definition 1.3. Inside an interface at $\mathtt{id}$ we write $\mathtt{id}_{\mathcal{F}_{\mathsf{Net}}}$ for $(\mathcal{F}_{\mathsf{Net}}.\mathit{pid}, \mathtt{id}.\mathrm{P})$ and $\mathtt{id}_{\mathcal{G}_{\mathsf{Clock}}}$ for $(\mathcal{G}_{\mathsf{Clock}}.\mathit{pid}, \mathtt{id}.\mathrm{P})$, the second the notation of Section 17.2.

Game shell over $\mathcal{F}_{\text{Net}}$ and $\mathcal{G}_{\text{Clock}}$; the finalizations follow

pid $:=$ $(\mathcal{G}\mathfrak{m}.\text{F}, 0, 0)$, $\mathbf{P} := \mathcal{F}_{\text{Net}}.\mathbf{P} \cup \{\text{Z}\}$, $\mathbf{N} := \text{Z}$, $\mathbf{U} := \{(\mathcal{F}_{\text{Net}}, \text{SERVES}_{\mathcal{G}\mathfrak{m}}), (\mathcal{G}_{\text{Clock}}, \text{SERVES})\}$, $par := \bot$

INITIALIZE():
1: DONE $\leftarrow$ FALSE
2: tr $\leftarrow$ ( ) // the trace, a sequence

id.SEND(*msg*) from id′
3: **require** ¬DONE
4: $\tau \leftarrow \text{id}_{\mathcal{G}_{\text{Clock}}}$.FULLREAD() as id
5: $\mathfrak{m} \leftarrow \text{id}_{\mathcal{F}_{\text{Net}}}$.FULLSEND(*msg*) as id
6: tr $\leftarrow$ tr $\cdot$ (SEND, id.P, $\mathfrak{m}$, τ, *msg*)
7: **return** $\mathfrak{m}$

id.FETCH() from id′
8: **require** ¬DONE
9: $\tau \leftarrow \text{id}_{\mathcal{G}_{\text{Clock}}}$.FULLREAD() as id
10: $\text{M} \leftarrow \text{id}_{\mathcal{F}_{\text{Net}}}$.FULLFETCH() as id
11: tr $\leftarrow$ tr $\cdot$ (FETCH, id.P, τ, M)
12: **return** M

id.LEAK() from id′
13: **return** $\bot$

The finalizations of $\mathcal{G}\mathfrak{m}$; the shell over $\mathcal{F}_{\text{Net}}$ and $\mathcal{G}_{\text{Clock}}$ is shared

id.FINLIVE() from id′ (liveness)
14: **require** ¬DONE
15: **require** id.P $= \text{Z}$
16: DONE $\leftarrow$ TRUE
17: $\mathbf{C} \leftarrow (\text{C}, \text{Z})$.FULLSTATUS() as id
18: $w \leftarrow 1$ if $\exists\, \text{P}, \text{Q}, \mathfrak{m}, \tau_s, msg, \tau, \text{M}:\ \text{P} \notin \mathbf{C} \wedge \text{Q} \notin \mathbf{C}$
19: $\quad \wedge$ (SEND, P, $\mathfrak{m}$, τ_s, *msg*) $\in$ tr $\wedge$ (FETCH, Q, τ, M) $\in$ tr
20: $\quad \wedge\ \tau \geqslant \tau_s + \Delta \wedge (\mathfrak{m}, msg) \notin \text{M}$;
21: else $w \leftarrow 0$
22: **return** w

id.FINAUTH() from id′ (authentication)
23: **require** ¬DONE
24: **require** id.P $= \text{Z}$
25: DONE $\leftarrow$ TRUE
26: $\mathbf{C} \leftarrow (\text{C}, \text{Z})$.FULLSTATUS() as id
27: $w \leftarrow 1$ if $\exists\, \text{Q}, \tau, \text{M}, \mathfrak{m}, msg$:
28: $\quad \text{Q} \notin \mathbf{C} \wedge$ (FETCH, Q, τ, M) $\in$ tr $\wedge$ $(\mathfrak{m}, msg) \in \text{M}$
29: $\quad \wedge\ \neg\exists\, \text{P}, \tau_s$: (SEND, P, $\mathfrak{m}$, τ_s, *msg*) $\in$ tr;
30: else $w \leftarrow 0$
31: **return** w

Three things about the code. Neither relay reads SEND's or FETCH's own return value for a timestamp — there is none to read — so both read the clock themselves, one line above the call to $\mathcal{F}_{\text{Net}}$; the two readings coincide because no core of γ runs between them, not $\mathcal{F}_{\text{Net}}$'s own guard and corruption check, which only reads $\mathbf{C}$ and finds id.P honest, the relay having run at all, and not READ itself, which places no call (Section 17.1). Neither relay excludes id.P $=$ Z, unlike Section 17.2's: $\mathcal{F}_{\text{Net}}$'s fields follow $\mathcal{F}_{\text{Sig}}$'s pattern and not the clock's, and

neither verdict reads any the worse for the root sending or fetching too — so the shell is spared the demand steadiness made of the clock's, and both finalizations get the shell they would have asked for alone. And authentication's verdict asks nothing of a sender, because FETCH returns none to ask about — the fetched pair is (m, msg) alone, and whether some m was ever sent, by whom, is exactly the fact no caller of $\mathcal{F}_{\mathsf{Net}}$ can otherwise recover, which is what the whole game is testing.

Proposition 18.1 ($\mathcal{F}_{\mathsf{Net}}$ has liveness)**.** *For every $\mathcal{Z} \in \mathbb{Z}_{\gamma,\gamma}$ and every occupant $\mathcal{X}$ of the adversary slot,* $\Pr[\mathrm{WIN}_{\mathrm{FINLIVE}}] = 0$. *So $\{\mathcal{F}_{\mathsf{Net}}, \mathcal{G}_{\mathsf{Clock}}\}$ has liveness within 0 against $(\mathbb{Z}_{\gamma,\gamma}, \mathbb{X})$ in the sense of Definition 5.8, for $\mathbb{X}$ the class of all occupants.*

Proof. Suppose $(\text{SEND}, P, m, \tau_s, msg) \in \mathtt{tr}$, written at some activation of the SEND relay. Its own clock read and $\mathcal{F}_{\mathsf{Net}}$.SEND's internal one (Section 18.1) answer the same value, by the argument just given, so the core stores exactly (m, P, msg, τ_s) in $\mathtt{buf}$.

Now suppose also $(\text{FETCH}, Q, \tau, M) \in \mathtt{tr}$ with $\tau \geqslant \tau_s + \Delta$. $\mathcal{F}_{\mathsf{Net}}$.FETCH's core returns every pair (m', msg') with $(m', P', msg', \tau_{s'}) \in \mathtt{buf}$ and $\mathtt{rel}[m', Q] = 1 \vee \tau_{s'} + \Delta \leqslant \tau$ (line 20), τ its own reading of the clock and, again, the relay's. Taking $m' = m$: the buffer entry is (m, P, msg, τ_s) and $\tau_s + \Delta \leqslant \tau$ holds by hypothesis, so the second disjunct is met whatever $\mathtt{rel}[m, Q]$ is and whatever the slot answered at line 12, which settles only S and I and neither disjunct. Hence $(m, msg) \in M$, and line 18 gives 0. □

Proposition 18.2 ($\mathcal{F}_{\mathsf{Net}}$ does not have authentication)**.** *There are $\mathcal{Z}^* \in \mathbb{Z}_{\gamma,\gamma}$ and an occupant $\mathcal{X}^*$ of the adversary slot with* $\Pr[\mathrm{WIN}_{\mathrm{FINAUTH}}] = 1$*: no $\varepsilon < 1$ bounds $\{\mathcal{F}_{\mathsf{Net}}, \mathcal{G}_{\mathsf{Clock}}\}$'s authentication against any class containing them.*

Proof. A counterexample in two machines. The occupant $\mathcal{X}^*$ answers $\mathcal{F}_{\mathsf{Net}}$'s fetch query with one injection and places no call — which is exactly why it is not the dummy, whose only way to answer is to relay to the environment, a call RESPOND refuses. That difference is the whole point of stating these properties over every occupant of the slot rather than the dummy alone: read at the dummy, FINAUTH would come out satisfied, and only vacuously. The environment $\mathcal{Z}^*$ then fetches once and finalizes on FINAUTH and nothing else. The

injection survives sanitizing, reaches the trace, and no SEND relay ever ran, so the verdict fires with advantage 1. Let $\mathcal{X}^*$ have $\mathcal{A}$'s fields but place no call. On the query $\mathtt{id}.\mathrm{A}(\mathtt{id}''.\textsc{Fetch}, \tau)$ with $\mathtt{id}''.\mathrm{F} = \mathcal{F}_{\mathsf{Net}}.pid$ — the shape of line 12 — it returns $(\varnothing, \{msg^*\})$ for one fixed $msg^* \in \mathcal{M}$; on anything else it returns $\bot$. Placing no call, it is never refused by RESPOND at either of $\mathcal{F}_{\mathsf{Net}}$'s slot calls, unlike the dummy, whose only way to answer is to relay to $\mathcal{Z}$ (line 5), a call RESPOND does refuse (Remark 9.4). This is exactly where $\mathcal{X}^*$ differs from $\mathcal{D}$, and exactly why Propositions 15.1, 17.2 and 18.1 are stated over every occupant of the slot and not the dummy alone — the dummy's case of Definition 5.8, with $\mathcal{X}$ dropped from the subscript, would read FINAUTH as satisfied, and only vacuously, $\mathcal{D}$ being unable to inject at all.

Let $\mathcal{Z}^*$ claim $(\mathcal{G}\mathfrak{m}.pid, \mathrm{Q})$ for $\mathrm{Q} \in \mathcal{F}_{\mathsf{Net}}.\mathbf{P}$, call $\textsc{Fetch}()$ once, then claim $(\mathcal{G}\mathfrak{m}.pid, \mathrm{Z})$ and call $\textsc{FinAuth}()$ — and FINAUTH and no other, which is the whole of what it takes to be judged on this property and not the one beside it. It corrupts no one, hosts nothing, and claims only identities of $\mathcal{G}\mathfrak{m}$'s own name, so $\mathcal{Z}^* \in \mathbb{Z}_{\gamma,\gamma}$.

At that fetch, $\mathrm{I} = \{msg^*\}$ survives sanitizing ($\mathrm{I} \leftarrow \mathrm{I} \cap \mathcal{M}$), so the loop of line 16 adds some fresh $(\mathfrak{m}, \bot, msg^*, \infty)$ to $\mathtt{buf}$ and sets $\mathtt{rel}[\mathfrak{m}, \mathrm{Q}] \leftarrow 1$; $(\mathfrak{m}, msg^*)$ is returned, and $(\textsc{Fetch}, \mathrm{Q}, \tau, \mathrm{M}) \in \mathtt{tr}$ with $(\mathfrak{m}, msg^*) \in \mathrm{M}$. No SEND relay ever ran, so $\mathtt{tr}$ carries no entry $(\textsc{Send}, \cdot, \mathfrak{m}, \cdot, msg^*)$ — indeed no SEND entry naming $\mathfrak{m}$ at all, $\mathfrak{m}$ being fresh to the injection. So $\mathrm{Q} \notin \mathbf{C}$ throughout, and line 27 gives 1.

Nothing about that run touches Proposition 18.1. The environment $\mathcal{Z}^*$ closed the game at FINAUTH, so FINLIVE was never answered and $\textsc{Win}_{\textsc{FinLive}}$ did not occur: the two events are disjoint by Remark 5.3, and the same shell can therefore host a property it always keeps beside one it always loses without the two saying anything about each other. □

As with $\mathcal{G}_{\mathsf{Clock}}$, liveness transfers by Corollary 5.11 only if the cores of the game system place no responsive call, and both of $\mathcal{F}_{\mathsf{Net}}$'s do, lines 8 and 12; a system π put in place of $\{\mathcal{F}_{\mathsf{Net}}, \mathcal{G}_{\mathsf{Clock}}\}$ must then have the transfer argued directly, for the reason Remark 9.4 gives, exactly as the clock's own notification cost Section 17.2 the same thing. Tightness, note, did no work in either proof above beyond licensing γ as a game system: unlike correctness and unforgeability, neither verdict here needs to rule out a message entering $\mathtt{buf}$ by some route $\mathcal{G}\mathfrak{m}$ never sees, only to read off what happens to what $\mathcal{G}\mathfrak{m}$ itself placed — so both

propositions would survive a weaker dependence, were one wanted, though nothing above asks for it.

Authentication transfers nothing, having nothing to transfer: Proposition 18.2 is not a bound to inherit but a fact about the specification, exactly the fact Section 18.1 already asserted in prose — proved now rather than merely said, and by an adversary no stranger than one that ignores its mail. No replacement of $\{\mathcal{F}_{\mathsf{Net}}, \mathcal{G}_{\mathsf{Clock}}\}$ is asked to improve on it. What does have the analogous property is Chapter 19's $\mathcal{F}_{\mathsf{AC}}$: a corrupt sender's traffic there enters through SEND itself, under its own identity, rather than through an unaccountable slot answer (Chapter 19, "Authentication is the adversary's own interface"), so the fetched pair carries a sender for a verdict to name, and there the transplanted win condition — fetched, addressed, and never sent — is not met.

CHAPTER 19

Δ-Delayed Authenticated Channel

19.1 Functionality

$\mathcal{F}_{\mathsf{Net}}$ diffuses and nobody signs. The complementary object is the one where every message arrives with its sender attached and none arrives from nobody — the authenticated channel of [7], given the same Δ and the same clock. The channel $\mathcal{F}_{\mathsf{AC}}$ below is that functionality, and the two are worth reading side by side: they differ in a good deal more than the sender field, and the difference is where authentication actually lives.

What it does is quickly said. The operation SEND takes a recipient as well as a message, stamps the pair by the clock, files it under a fresh index, and tells the adversary — authentication is not secrecy, and this channel hides nothing at all. The operation FETCH gathers everything addressed to the fetching party that has come of age, asks the adversary which of the rest to release early, answers the union, and *empties* the entries it answered. What the receiver gets is the whole record: sender, recipient, send time, message. The division of power is $\mathcal{F}_{\mathsf{Net}}$'s — the adversary chooses when within Δ and in what order — and so is the bound: an entry Δ old is in the next fetch of the party it names, whatever the slot answers.

The fields. They are $\mathcal{F}_{\mathsf{Net}}$'s, for $\mathcal{F}_{\mathsf{Net}}$'s reasons. *pid*, **P** and **N** are parameters, one instance per session, with **N** naming the protocol served in the local pattern of Chapter 1, and *par* $:= \Delta$. The **U** field is $\{(\mathcal{G}_{\mathsf{Clock}}, \text{SERVES}), (\mathcal{A}, \text{SERVES})\}$, its clock entry met at the shared copy by the global disjunct of Definition 1.3. Both conjuncts of SERVES are needed and neither is a formality: the first asks the clock to serve every

party the channel does, and without the second the channel's own name would not be admitted at the clock, so GUARD would refuse every read on line 4.

Authentication is the adversary's own interface. $\mathcal{F}_{\mathsf{Net}}$ needs line 16 because a diffusion channel gives adversarial traffic no name to enter under: the wire is anonymous, so forgery is modelled by letting the slot write on it directly. Here there is a name, and the full interface already delivers it. At a corrupt sender the adversary reaches the core — lines 3 and 4 admit $\mathcal{A}$ at a corrupt party and neither mediation hook fires on its calls, both testing $\mathtt{id}'.\mathsf{F} \neq \mathsf{A}$ — so corrupt traffic enters through SEND under the sender's own identity, timestamped by the clock like anyone's. The channel $\mathcal{F}_{\mathsf{AC}}$ therefore has no injections and needs none, and the sender field of an entry is not a report about the wire but a fact about which interface the call came through. That is what an authenticated channel is: forgery is impossible because there is nowhere to forge from.

The price is that adversarial traffic *ripens*. A message the adversary sends from a corrupt party is in the list with a real timestamp, so line 12 will deliver it by Δ and no later answer can retract it. The network $\mathcal{F}_{\mathsf{Net}}$ refuses this deliberately, filing injections at ∞ so the guarantee is for the honest and never conscripts the slot's own traffic. The difference is a consequence of addressing rather than a choice made against it: a message with a named sender and a named recipient is a message the sender is on the record for, and holding the adversary to the deadline it accepted when it spoke as a corrupt party is the reading that keeps the sender field meaning what it says.

Fetching empties. Each entry names one recipient, so answering it discharges it, and line 16 clears what line 12 and the slot between them selected. The network $\mathcal{F}_{\mathsf{Net}}$ cannot do this — a diffused message is owed to everyone — which is why it carries the sticky `rel` table instead and re-answers what it has already answered. The two guarantees are different in kind and neither implies the other: $\mathcal{F}_{\mathsf{Net}}$'s is that a party's view only grows, $\mathcal{F}_{\mathsf{AC}}$'s that a message is delivered exactly once. A protocol over $\mathcal{F}_{\mathsf{Net}}$ must deduplicate and one over $\mathcal{F}_{\mathsf{AC}}$ must not lose what it fetched, and each obligation is the other's absence.

Sanitizing the release. The slot's answer is a set of indices to release early, and the only thing wrong it can be is a set naming entries that are not the fetching party's — or not entries at all. Chapter 12's procedure

repairs exactly this kind of fault, so line 14 runs the answer through it under

$$\textsc{Clean}_{\textsc{addr}}(S;\ \texttt{list}, P)\ :\Longleftrightarrow\ \ S \subseteq \{\, i : \texttt{list}[i] = (\cdot,\ P,\ \cdot,\ \cdot)\,\},$$

with the list and the fetching party as the read-only context. Two things follow that are worth saying, because they are what makes SAN the right instrument here rather than an assertion. A good value always exists, so the footnote of Chapter 12 has nothing to prove in this case: $\varnothing$ satisfies $\textsc{Clean}_{\textsc{addr}}$ whatever the list holds, and being the empty set it is also the lexicographically first such, so a failing answer is repaired to no early release at all. And that repair costs the adversary nothing it could not have had, since an adversary wanting some particular good subset released may submit that subset and be returned it untouched. An assertion in SAN's place would read the same on good answers and hand the slot a denial of service on bad ones — one stray index and the honest party's fetch dies — which is the failure Chapter 12 exists to rule out.

Both slot calls are responsive. The clock's discipline, and here it is load-bearing in a way worth spelling out. The operation FETCH reads the time on line 10 and asks the slot on line 13. Were that call not responsive the adversary would hold the token in between and could drive the clock forward before answering; τ would then be stale by the time line 12 used it, fewer entries would clear $\tau_i + \Delta \leqslant \tau$, and a message past its deadline would be lawfully withheld. Responsiveness is what makes the read and the test one moment. The operation SEND mutates before it notifies and needs no such argument, but is responsive too, for $\mathcal{F}_{\mathsf{Net}}$'s reason: nothing else may run inside either call.

Functionality $\mathcal{F}_{\mathsf{AC}}$
pid, **P**, **N**, $\mathbf{U} := \{(\mathcal{G}_{\mathsf{Clock}}, \textsc{serves}), (\mathcal{A}, \textsc{serves})\}$, *par* $:= \Delta$

Initialize():
1: $\mathtt{ctr} \leftarrow 0$
2: $\mathtt{list} : \mathbb{N} \to (\mathcal{F}_{\mathsf{AC}}.\mathbf{P}^2 \times \mathbb{N} \times \mathcal{M}) \cup \{\bot\}$
3: $\mathtt{list}[*] \leftarrow \bot$ *// entries* (P, Q, τ, msg)

id.Send(Q, *msg*) from id′
4: $\tau \leftarrow \mathtt{id}_{\mathcal{G}_{\mathsf{Clock}}}.\textsc{FullRead}()$ as id
5: $\mathtt{ctr} \leftarrow \mathtt{ctr} + 1$
6: $\mathtt{list}[\mathtt{ctr}] \leftarrow (\mathtt{id}.P, Q, \tau, msg)$
7: $\mathcal{A}^{!}(\mathtt{id}.\textsc{Send}, \mathtt{ctr}, Q, \tau, msg)$
8: **return** OK

id.Leak() from id′
9: **return** list

id.Fetch() from id′
10: $\tau \leftarrow \mathtt{id}_{\mathcal{G}_{\mathsf{Clock}}}.\textsc{FullRead}()$ as id
11: $R \leftarrow \{m : \mathtt{list}[m] = (\cdot, \mathtt{id}.P, \tau_m, \cdot)$
12: $\quad \wedge\ \tau_m + \Delta \leqslant \tau\}$ *// come of age*
13: $S \leftarrow \mathcal{A}^{!}(\mathtt{id}.\textsc{Fetch}, \tau)$ *// released early*
14: $S \leftarrow \textsc{San}[\textsc{Clean}_{\textsc{addr}}](S; \mathtt{list}, \mathtt{id}.P)$
15: $M \leftarrow \{\mathtt{list}[m] : m \in R \cup S\}$
16: $\mathtt{list}[m] \leftarrow \bot$ for each $m \in R \cup S$
17: **return** M

Three remarks on the code. The recipient is not checked against **P** on Send: a message addressed outside the channel's parties is filed and never fetched, which is the same silence a party who declines to fetch would produce, and the guarantee below is a statement about entries someone fetches. The send time travels with the message, so a receiver can date what it receives and check the deadline itself — $\mathcal{F}_{\mathsf{Net}}$'s fetch returns no such thing, and the difference is that $\mathcal{F}_{\mathsf{AC}}$'s Δ is locally auditable where $\mathcal{F}_{\mathsf{Net}}$'s is only a fact about the functionality. And Leak returns the list entire, the channel keeping no secrets: its contents were told to the adversary at line 7 as they arrived, and what has since been drained is what the slot itself watched leave.

19.2 Properties

The paragraph above named two claims: delivery by Δ, and the one $\mathcal{F}_{\mathsf{Net}}$ cannot make — every message fetched was sent, by the party it names. A third, no delivery before sending, again needs no game: an index not yet in list, or already drained, fails $\textsc{Clean}_{\textsc{addr}}$, so San repairs S to $\varnothing$ and nothing not yet real is released early either (Section 18.2 made the same point for $\mathcal{F}_{\mathsf{Net}}$'s buffer). The finalizations FinLive and FinAuth below formalize the other two, transplanting Section 18.2's pair rather than starting over — the shell of relay, log, finalize does not change between a diffusion channel and an addressed one, and neither do the names, these being the same two properties asked of a different channel. Both take the search reading. What changes is which of the two $\mathcal{F}_{\mathsf{AC}}$ can keep: both, this time, and the

second is the point of the whole design. That the transplant is *name for name* is the sharpest way to put the difference between the two channels — one game, asked of $\mathcal{F}_{\mathsf{Net}}$ and of $\mathcal{F}_{\mathsf{AC}}$, answered differently.

Both are played over the game system $\gamma := \{\mathcal{G}\mathfrak{m}\} \cup \{\mathcal{F}_{\mathsf{AC}}, \mathcal{G}_{\mathsf{Clock}}\}$, with $\mathcal{F}_{\mathsf{AC}}.\mathbf{N} := \{\mathcal{G}\mathfrak{m}.\mathit{pid}\}$ pinned and $\mathcal{F}_{\mathsf{AC}}.\mathbf{P} \subseteq \mathcal{G}\mathfrak{m}.\mathbf{P}$ — the same tightness argument as Section 18.2, and for the same reason, $\mathcal{F}_{\mathsf{AC}}$'s fields already following $\mathcal{F}_{\mathsf{Net}}$'s pattern (Section 19.1). No caller but $\mathcal{G}\mathfrak{m}$ passes GUARD at $\mathcal{F}_{\mathsf{AC}}$, so every entry SEND ever places in `list`, and every record FETCH ever returns, passes through the relays below and is recorded. The interface box states $\mathcal{G}\mathfrak{m}.\mathbf{U}$ in full. Inside an interface at `id` we write $\mathtt{id}_{\mathcal{F}_{\mathsf{AC}}}$ for $(\mathcal{F}_{\mathsf{AC}}.\mathit{pid}, \mathtt{id}.\mathrm{P})$ and $\mathtt{id}_{\mathcal{G}_{\mathsf{Clock}}}$ for $(\mathcal{G}_{\mathsf{Clock}}.\mathit{pid}, \mathtt{id}.\mathrm{P})$, the second the notation of Section 17.2.

Game shell over $\mathcal{F}_{\mathsf{AC}}$ and $\mathcal{G}_{\mathsf{Clock}}$; the finalizations follow

$\mathit{pid} := (\mathcal{G}\mathfrak{m}.\mathrm{F}, 0, 0)$, $\mathbf{P} := \mathcal{F}_{\mathsf{AC}}.\mathbf{P} \cup \{\mathrm{Z}\}$, $\mathbf{N} := \mathbf{Z}$, $\mathbf{U} := \{(\mathcal{F}_{\mathsf{AC}}, \mathrm{SERVES}_{\mathcal{G}\mathfrak{m}}), (\mathcal{G}_{\mathsf{Clock}}, \mathrm{SERVES})\}$, $\mathit{par} := \bot$

INITIALIZE():
1: DONE ← FALSE
2: `tr` ← ( ) // *the trace, a sequence*

id.SEND(Q, *msg*) from `id'`
3: **require** ¬DONE
4: τ ← $\mathtt{id}_{\mathcal{G}_{\mathsf{Clock}}}$.FULLREAD() as `id`
5: r ← $\mathtt{id}_{\mathcal{F}_{\mathsf{AC}}}$.FULLSEND(Q, *msg*) as `id`
6: `tr` ← `tr` · (SEND, `id`.P, Q, τ, *msg*)
7: **return** r

id.FETCH() from `id'`
8: **require** ¬DONE
9: τ ← $\mathtt{id}_{\mathcal{G}_{\mathsf{Clock}}}$.FULLREAD() as `id`
10: M ← $\mathtt{id}_{\mathcal{F}_{\mathsf{AC}}}$.FULLFETCH() as `id`
11: `tr` ← `tr` · (FETCH, `id`.P, τ, M)
12: **return** M

id.LEAK() from `id'`
13: **return** ⊥

The finalizations of $\mathcal{G}_{\mathrm{m}}$; the shell over $\mathcal{F}_{\mathsf{AC}}$ and $\mathcal{G}_{\mathsf{Clock}}$ is shared

id.FINLIVE() from id′ (liveness)
14: **require** ¬DONE
15: **require** id.P = Z
16: DONE ← TRUE
17: C ← (C, Z).FULLSTATUS() as id
18: $w \leftarrow 1$ if $\exists\, P, Q, \tau_s, msg, \tau, M:$ $P \notin C \wedge Q \notin C$
19: $\wedge$ (SEND, P, Q, τ_s, *msg*) $\in$ tr $\wedge$ (FETCH, Q, τ, M) $\in$ tr
20: $\wedge$ $\tau \geqslant \tau_s + \Delta \wedge \neg\exists \tau', M':$
21: (FETCH, Q, τ′, M′) $\in$ tr $\wedge$ (P, Q, τ_s, *msg*) $\in$ M′;
22: else $w \leftarrow 0$
23: **return** w

id.FINAUTH() from id′ (authentication)
24: **require** ¬DONE
25: **require** id.P = Z
26: DONE ← TRUE
27: C ← (C, Z).FULLSTATUS() as id
28: $w \leftarrow 1$ if $\exists\, Q, \tau, M, P, \tau_s, msg:$
29: $Q \notin C \wedge$ (FETCH, Q, τ, M) $\in$ tr $\wedge$ (P, Q, τ_s, *msg*) $\in$ M
30: $\wedge$ (SEND, P, Q, τ_s, *msg*) $\notin$ tr;
31: else $w \leftarrow 0$
32: **return** w

Three things about the code, two of them familiar. Both relays read the clock themselves for the same reason as Section 18.2's: the two readings coincide because no core of γ runs between them, READ placing no call of its own. SEND's relay keeps $\mathcal{F}_{\mathsf{AC}}$.SEND's own return, OK, having nothing else to report — unlike $\mathcal{F}_{\mathsf{Net}}$'s, which had a message id to pass back and $\mathcal{F}_{\mathsf{AC}}$ never hands out. And authentication's verdict needs no search over senders or times, unlike its namesake's: a fetched record already names P and τ_s, so the win condition is a direct check that the exact tuple was never sent, not an existential one over what might have produced it — the address $\mathcal{F}_{\mathsf{AC}}$ carries is doing, in the verdict, exactly what it does at the interface.

Proposition 19.1 ($\mathcal{F}_{\mathsf{AC}}$ has liveness). *For every $\mathcal{Z} \in \mathbb{Z}_{\gamma,\gamma}$ and every occupant $\mathcal{X}$ of the adversary slot,* $\Pr[\mathrm{WIN}_{\mathrm{FINLIVE}}] = 0$. *So $\{\mathcal{F}_{\mathsf{AC}}, \mathcal{G}_{\mathsf{Clock}}\}$ has liveness within 0 against $(\mathbb{Z}_{\gamma,\gamma}, \mathbb{X})$ in the sense of Definition 5.8, for $\mathbb{X}$ the class of all occupants.*

Proof. Suppose (SEND, P, Q, τ_s, *msg*) $\in$ tr, written at some activation of the SEND relay (line 5). Its own clock read and $\mathcal{F}_{\mathsf{AC}}$.SEND's internal

one (line 4) agree, by the argument of Section 18.2: no core of γ runs between them, $\mathcal{F}_{\mathsf{AC}}$'s guard and corruption check finding $\mathtt{id}.\mathsf{P}$ honest since the relay ran at all, and READ itself placing no call. So the core wrote some fresh m with $\mathtt{list}[m] = (\mathsf{P}, \mathsf{Q}, \tau_s, \mathit{msg})$ at that moment.

Suppose too $(\text{FETCH}, \mathsf{Q}, \tau, \mathsf{M}) \in \mathtt{tr}$ with $\tau \geqslant \tau_s + \Delta$. If $\mathtt{list}[m]$ still holds $(\mathsf{P}, \mathsf{Q}, \tau_s, \mathit{msg})$ when that fetch's core runs, then $m \in \mathsf{R}$ by line 12 — the entry names Q, and $\tau_s + \Delta \leqslant \tau$ by hypothesis, for τ its own reading of the clock and, by the same argument again, the relay's — so $(\mathsf{P}, \mathsf{Q}, \tau_s, \mathit{msg}) \in \mathsf{M}$ whatever the slot answers at line 13 and whatever SAN does with it, neither touching R. If instead $\mathtt{list}[m]$ had already been cleared, that happened only at line 16 of some earlier fetch by Q — the one line that ever clears an entry — and only on indices already placed in that fetch's own $\mathsf{R} \cup \mathsf{S}$, hence already present in that earlier fetch's own M. Either way $(\mathsf{P}, \mathsf{Q}, \tau_s, \mathit{msg})$ appears in some fetch of Q's, and the verdict at line 18 gives 0. □

Proposition 19.2 ($\mathcal{F}_{\mathsf{AC}}$ has authentication). *For every $\mathcal{Z} \in \mathbb{Z}_{\gamma,\gamma}$ and every occupant $\mathcal{X}$ of the adversary slot, $\Pr[\text{WIN}_{\text{FINAUTH}}] = 0$. So $\{\mathcal{F}_{\mathsf{AC}}, \mathcal{G}_{\mathsf{Clock}}\}$ has authentication within 0 against $(\mathbb{Z}_{\gamma,\gamma}, \mathbb{X})$ in the sense of Definition 5.8, for $\mathbb{X}$ the class of all occupants — the property Section 18.2 proved $\mathcal{F}_{\mathsf{Net}}$ lacks, held here instead.*

Proof. Suppose $(\text{FETCH}, \mathsf{Q}, \tau, \mathsf{M}) \in \mathtt{tr}$ with $\mathsf{Q} \notin \mathbf{C}$ and $(\mathsf{P}, \mathsf{Q}, \tau_s, \mathit{msg}) \in \mathsf{M}$. By $\mathcal{F}_{\mathsf{AC}}$.FETCH's code, $\mathsf{M} = \{\mathtt{list}[m] : m \in \mathsf{R} \cup \mathsf{S}\}$, and both index sets draw only on $\mathtt{list}$ as it already stands: R by its own pattern match (line 12), and S after sanitizing, since $\text{CLEAN}_{\text{ADDR}}$ admits only indices i with $\mathtt{list}[i] = (\cdot, \mathsf{Q}, \cdot, \cdot)$ already and line 3 (Chapter 12) picks among values meeting that predicate — neither manufactures an index $\mathtt{list}$ does not hold. So $(\mathsf{P}, \mathsf{Q}, \tau_s, \mathit{msg}) \in \mathsf{M}$ only if $\mathtt{list}[m] = (\mathsf{P}, \mathsf{Q}, \tau_s, \mathit{msg})$ for some m at some point.

The only line that ever writes $\mathtt{list}[m]$ to a value other than $\perp$ is SEND's own core, and by tightness — $\mathcal{F}_{\mathsf{AC}}.\mathbf{N} = \{\mathcal{G}m.\mathit{pid}\}$ pinned — that core runs only on calls $\mathcal{G}m$'s relay places: not the environment, not the adversary, not a machine hosted at an invented instance of the game's name. So the write producing $(\mathsf{P}, \mathsf{Q}, \tau_s, \mathit{msg})$ happened at some activation of the SEND relay, and by the coinciding-read argument of the previous proof it was logged there as exactly $(\text{SEND}, \mathsf{P}, \mathsf{Q}, \tau_s, \mathit{msg})$. Hence $(\text{SEND}, \mathsf{P}, \mathsf{Q}, \tau_s, \mathit{msg}) \in \mathtt{tr}$, and the verdict at line 28 gives 0. □

As with $\mathcal{G}_{\mathsf{Clock}}$ and $\mathcal{F}_{\mathsf{Net}}$, both properties transfer by Corollary 5.11 only if the game system's cores place no responsive call — and here, as there, neither does: Section 19.1's "Both slot calls are responsive" covers SEND's notification and FETCH's query alike. A system π put in place of $\{\mathcal{F}_{\mathsf{AC}}, \mathcal{G}_{\mathsf{Clock}}\}$ must then have both transfers argued directly, for the reason Remark 9.4 gives, exactly as $\mathcal{F}_{\mathsf{Net}}$'s own notification cost Section 18.2 the same thing.

Unlike the network, nothing here is left standing only as a negative result. Proposition 19.2 is the win condition Section 18.2 transplanted from $\mathcal{F}_{\mathsf{Net}}$ and could not close, closed here instead — and the two proofs, laid side by side, locate the whole difference in one fact: whether tightness can trace a fetched record back to a SEND that produced it. For $\mathcal{F}_{\mathsf{AC}}$ it always can, `list` having no other way to gain an entry; for $\mathcal{F}_{\mathsf{Net}}$ it cannot, line 16 being exactly the other way in. Two chapters, one game transplanted twice, and the address is the entire difference between them.

That the transplant keeps the names is what makes the comparison sayable at all. The finalization FINAUTH is not two properties that resemble each other but one win condition — fetched, addressed, never sent — asked of two channels, and the answer is 1 for $\mathcal{F}_{\mathsf{Net}}$ and 0 for $\mathcal{F}_{\mathsf{AC}}$. Under the shape Chapter 5 replaced, where each property carried its own game, the two would have been $\mathcal{G}\mathfrak{m}_{\mathrm{AUTH}}$ and $\mathcal{G}\mathfrak{m}'_{\mathrm{AUTH}}$ and the sameness would have been a remark; here it is the notation.

CHAPTER 20

Realizing the Authenticated Channel

The channel of Chapter 19 was postulated. This chapter earns it: a protocol ρ over three ideal subroutines — a public-key directory, the unauthenticated network of Chapter 18, and the signatures of Chapter 15 — that UC-realizes an authenticated channel, and does so *exactly*, within 0, every subroutine being ideal and the whole reduction information-theoretic. It is the largest construction in the paper, and the one that exercises the most of it at once: composition, the responsive discipline, sanitization, tightness, and the fundamental lemma of Chapter 11 all appear, each doing a job no other could.

What ρ does. A party sending for the first time draws a signing key from $\mathcal{F}_{\mathsf{Sig}}$ and registers it in the directory $\mathcal{F}_{\mathsf{PKI}}$. To send *msg* to Q it reads the clock for a timestamp τ, increments a per-party sequence number s, signs the tuple $(\P, Q, \tau, s, \mathit{msg})$, and diffuses the tuple with its signature over $\mathcal{F}_{\mathsf{Net}}$. To fetch, it collects what $\mathcal{F}_{\mathsf{Net}}$ delivers, and keeps a diffused item only if it is addressed to the fetcher, carries a signature that verifies under the *claimed* sender's directory key, and has a (sender, s) pair not seen before. What survives it hands up as $(\P, Q, \tau, \mathit{msg})$. Diffusion carries no addresses and no authenticity; the signature supplies the second and the address inside the signed tuple the first, so that a receiver trusts a sender's name exactly because that name is under the sender's own signature.

Why the target is $\mathcal{F}_{\mathsf{AC}}^{\circ}$, not $\mathcal{F}_{\mathsf{AC}}$. A corrupt sender is the obstruction, and it is instructive rather than incidental. Over a diffusion network the adversary writes on the wire whenever it likes, in particular *during* a fetch — $\mathcal{F}_{\mathsf{Net}}$'s injections (line 16) are chosen inside the responsive window of the fetch that carries them. A message so injected, signed

under a key the adversary registered for a corrupt party, is accepted by the receiver and delivered; but no prior SEND ever placed it, so $\mathcal{F}_{\mathsf{AC}}$, which delivers only what its own SEND stored, cannot produce it, and no simulator can call $\mathcal{F}_{\mathsf{AC}}$.SEND inside the fetch's responsive window to repair the gap. Worse, the corrupt sender's timestamp is its own to choose — the signature validates whatever τ the payload carries — where $\mathcal{F}_{\mathsf{AC}}$ stamps the true time of a call it witnessed. Neither gap is a defect in ρ; both are what an authenticated *diffusion* unavoidably gives a corrupt party, and Chapter 20 closes with the argument that no protocol over these subroutines does better. So the realized object is $\mathcal{F}_{\mathsf{AC}}^{\circ}$: $\mathcal{F}_{\mathsf{AC}}$ with a corrupt sender's traffic admitted at the fetch, unstored and freely timestamped, which is exactly the guarantee level of [7]'s authenticated channel and no less. The honest-sender guarantees — delivery by Δ, and authenticity, that an honest party's fetched record was genuinely sent by the honest party it names — survive whole. And against an adversary that corrupts no one the two objects coincide: the fundamental lemma prices the difference at Pr[BAD] for a bad that only a corrupt sender can raise, so ρ realizes $\mathcal{F}_{\mathsf{AC}}$ itself, exactly, whenever nobody is corrupt (Corollary 20.2).

Two subroutines are not quite off the shelf. The signatures are the responsive variant $\mathcal{F}_{\mathsf{Sig}}^{!}$ of Chapter 9 — $\mathcal{A}^{!}$ in GEN, SIGN and VERIFY — and the reason is the token. $\mathcal{F}_{\mathsf{AC}}$'s fetch is atomic: its one slot call is responsive, so no occupant drives the execution between the call and its answer. Were ρ's fetch to reach $\mathcal{F}_{\mathsf{Sig}}$ through the ordinary $\mathcal{A}(\cdot)$, a verification would hand the adversary the token mid-fetch and let it act elsewhere before returning, an interleaving $\mathcal{F}_{\mathsf{AC}}^{\circ}$'s atomic fetch cannot show and no simulator could reproduce. The responsive variant $\mathcal{F}_{\mathsf{Sig}}^{!}$ removes the escape; $\mathcal{F}_{\mathsf{Net}}$'s and $\mathcal{F}_{\mathsf{AC}}^{\circ}$'s calls are already responsive, so every send and every fetch is atomic on both sides. And the directory is *local*, not the global $\mathcal{G}_{\mathsf{PKI}}$ of Chapter 16: a shared directory is written by an honest ρ.SEND in the real world and read straight back by the environment through RETRIEVE, where in the ideal world nothing writes it and the simulator holds only the peek $\mathcal{G}_{\mathsf{PKI}}$ grants at honest parties — an unsimulatable difference, the commitment problem of a global setup in this paper's coordinates. A local $\mathcal{F}_{\mathsf{PKI}}$, its identities private to the session like any hybrid, closes it; Z is kept out of its **N** for the same reason, a direct retrieval by the environment being a call the ideal world has no machine to serve.

The directory forgets bindings on purpose. $\mathcal{F}_{\mathsf{PKI}}$ sanitizes a registered key to be a key and nothing more — *per slot*, dropping $\mathcal{G}_{\mathsf{PKI}}$'s cross-party freshness (Chapter 16, $\textsc{Clean}_{\textsc{reg}}$). The freshness would be a weapon: the adversary chooses an honest party's signing key, it being the value $\mathcal{F}_{\mathsf{Sig}}.\textsc{Gen}$ returns, so it could pre-register that key at a corrupt slot and have the honest party's own registration *repaired* to a different one, leaving directory and signing key disagreeing and the party's every message rejected for ever — a denial of the liveness $\mathcal{F}_{\mathsf{AC}}^{\circ}$ promises unconditionally. Per-slot fixity is enough because attribution never rested on binding: the sender's name travels inside the signed tuple, the receiver looks that name up, and a signature valid under an honest party's key on a tuple naming a *different* sender is a forgery $\mathcal{F}_{\mathsf{Sig}}$ suppresses. Binding buys nothing here and costs liveness, so it goes.

Local directory $\mathcal{F}_{\mathsf{PKI}}$
pid, **P**, **N**, **U** $:= \varnothing$, *par* $:= \bot$

$\textsc{Initialize}()$:
1: $\mathtt{VK} : \mathcal{F}_{\mathsf{PKI}}.\mathbf{P} \to \mathcal{K} \cup \{\square\}$
2: $\mathtt{VK}[*] \leftarrow \square$

$\mathtt{id}.\textsc{Register}(vk)$ from $\mathtt{id}'$
3: **if** $\mathtt{id}'.\mathrm{F} \in \{\mathrm{A}, \mathrm{Z}\} \wedge \mathtt{id}.\mathrm{P} \notin \mathbf{C}$ **then**
4: **return** $\mathtt{VK}[\mathtt{id}.\mathrm{P}]$ *// a peek, not a write*
5: **if** $\mathtt{VK}[\mathtt{id}.\mathrm{P}] \neq \square$ **then**
6: **return** $\mathtt{VK}[\mathtt{id}.\mathrm{P}]$
7: $vk \leftarrow \textsc{San}[\textsc{Clean}_{\textsc{key}}](vk)$
8: $\mathtt{VK}[\mathtt{id}.\mathrm{P}] \leftarrow vk$
9: **return** vk

$\mathtt{id}.\textsc{Retrieve}(\mathrm{P})$ from $\mathtt{id}'$
10: **return** $\mathtt{VK}[\mathrm{P}]$

$\mathtt{id}.\textsc{Leak}()$ from $\mathtt{id}'$
11: **return** $\mathtt{VK}$

$\textsc{Clean}_{\textsc{key}}(vk)$:
12: **return** $vk \in \mathcal{K}$

$\mathcal{F}_{\mathsf{AC}}^{\circ}$ is $\mathcal{F}_{\mathsf{AC}}$ with one operation changed: $\textsc{Fetch}$ takes back a pair (S, I) from the slot, S the early releases among stored ids as before, I a set of *injected* records the slot delivers unstored. Sanitizing pins each injection to a corrupt sender and to the fetching party, so an injection can only ever put a corrupt party's word into the fetch that asked for it; $\mathcal{F}_{\mathsf{AC}}$'s own $\textsc{Send}$, $\textsc{Leak}$ and $\textsc{Initialize}$ are unchanged.

Functionality $\mathcal{F}_{\mathsf{AC}}^{\circ}$: the fetch of $\mathcal{F}_{\mathsf{AC}}$, replaced
fields as $\mathcal{F}_{\mathsf{AC}}$ (Section 19.1)

id.FETCH() from id′
1: $\tau \leftarrow \mathtt{id}_{\mathcal{G}_{\mathsf{Clock}}}.\textsc{FullRead}()$ as id
2: $\mathbf{C} \leftarrow (\mathrm{C}, \mathtt{id}.\mathrm{P}).\textsc{FullStatus}()$ as id
3: $\mathrm{R} \leftarrow \{\mathrm{m} : \mathtt{list}[\mathrm{m}] = (\cdot, \mathtt{id}.\mathrm{P}, \tau_{\mathrm{m}}, \cdot) \wedge \tau_{\mathrm{m}} + \Delta \leqslant \tau\}$
4: $(\mathrm{S}, \mathrm{I}) \leftarrow \mathcal{A}^{!}(\mathtt{id}.\textsc{Fetch}, \tau)$ // releases and injections
5: $\mathrm{S} \leftarrow \textsc{San}[\textsc{Clean}_{\textsc{addr}}](\mathrm{S}; \mathtt{list}, \mathtt{id}.\mathrm{P})$
6: $\mathrm{I} \leftarrow \textsc{San}[\textsc{Clean}_{\textsc{inj}}](\mathrm{I}; \mathbf{C}, \mathtt{id}.\mathrm{P})$
7: $\mathrm{M} \leftarrow \{\mathtt{list}[\mathrm{m}] : \mathrm{m} \in \mathrm{R} \cup \mathrm{S}\} \cup \mathrm{I}$
8: $\mathtt{list}[\mathrm{m}] \leftarrow \bot$ for each $\mathrm{m} \in \mathrm{R} \cup \mathrm{S}$
9: **return** M

$\textsc{Clean}_{\textsc{inj}}(\mathrm{I}; \mathbf{C}, \mathrm{P}_{\mathrm{f}})$:
1: **return** $\mathrm{I} \subseteq \{(\mathrm{P}, \mathrm{Q}, \tau, \mathit{msg}) : \mathrm{P} \in \mathbf{C} \wedge \mathrm{Q} = \mathrm{P}_{\mathrm{f}}\}$

The protocol reads its four subroutines through the interfaces at the party being served, the notation $\mathtt{id}_{\mathcal{F}} := (\mathcal{F}.\mathit{pid}, \mathtt{id}.\mathrm{P})$ of Chapter 18. Its **U** names all four, the clock among them, since ρ stamps its own payloads and does not recover $\mathcal{F}_{\mathsf{Net}}$'s stamp, which $\mathcal{F}_{\mathsf{Net}}$ never returns.

Protocol ρ over $\{\mathcal{F}_{\mathsf{PKI}}, \mathcal{F}_{\mathsf{Net}}, \mathcal{F}_{\mathsf{Sig}}^{!}\}$
pid, **P**, **N**, **U** := $\{(\mathcal{F}_{\mathsf{PKI}}, \textsc{Serves}), (\mathcal{F}_{\mathsf{Net}}, \textsc{Serves}), (\mathcal{F}_{\mathsf{Sig}}^{!}, \textsc{Serves}), (\mathcal{G}_{\mathsf{Clock}}, \textsc{Serves})\}$, *par* := $\bot$

INITIALIZE():
1: $\mathtt{gen} : \mathbf{P} \to \{0, 1\}$; $\mathtt{gen}[*] \leftarrow 0$
2: $\mathtt{seq} : \mathbf{P} \to \mathbb{N}$; $\mathtt{seq}[*] \leftarrow 0$
3: $\mathtt{seen} : \mathbf{P} \to \mathcal{P}(\mathbf{P} \times \mathbb{N})$
4: $\mathtt{seen}[*] \leftarrow \varnothing$

id.SEND(Q, *msg*) from id′
5: **if** $\mathtt{gen}[\mathtt{id}.\mathrm{P}] = 0$ **then**
6: $\quad \mathit{vk} \leftarrow \mathtt{id}_{\mathcal{F}_{\mathsf{Sig}}}.\textsc{FullGen}()$ as id
7: $\quad \mathtt{id}_{\mathcal{F}_{\mathsf{PKI}}}.\textsc{FullRegister}(\mathit{vk})$ as id
8: $\quad \mathtt{gen}[\mathtt{id}.\mathrm{P}] \leftarrow 1$
9: $\tau \leftarrow \mathtt{id}_{\mathcal{G}_{\mathsf{Clock}}}.\textsc{FullRead}()$ as id
10: $\mathtt{seq}[\mathtt{id}.\mathrm{P}] \leftarrow \mathtt{seq}[\mathtt{id}.\mathrm{P}] + 1$
11: $\mathrm{x} \leftarrow (\mathtt{id}.\mathrm{P}, \mathrm{Q}, \tau, \mathtt{seq}[\mathtt{id}.\mathrm{P}], \mathit{msg})$
12: $\sigma \leftarrow \mathtt{id}_{\mathcal{F}_{\mathsf{Sig}}}.\textsc{FullSign}(\mathrm{x})$ as id
13: $\mathtt{id}_{\mathcal{F}_{\mathsf{Net}}}.\textsc{FullSend}((\mathrm{x}, \sigma))$ as id
14: **return** OK

id.FETCH() from id′
15: $\mathrm{N} \leftarrow \mathtt{id}_{\mathcal{F}_{\mathsf{Net}}}.\textsc{FullFetch}()$ as id
16: $\mathrm{M} \leftarrow \varnothing$
17: **for** $(\mathrm{m}, \mathrm{y}) \in \mathrm{N}$ **do**
18: $\quad$ parse y as (x, σ), x = $(\mathrm{P}, \mathrm{Q}, \tau, \mathrm{s}, \mathit{msg})$ // else skip
19: $\quad$ **if** $\mathrm{Q} \neq \mathtt{id}.\mathrm{P}$ **then** skip
20: $\quad \mathit{vk} \leftarrow \mathtt{id}_{\mathcal{F}_{\mathsf{PKI}}}.\textsc{FullRetrieve}(\mathrm{P})$ as id
21: $\quad$ **if** $\mathit{vk} = \square$ **then** skip
22: $\quad \mathrm{b} \leftarrow \mathtt{id}_{\mathcal{F}_{\mathsf{Sig}}}.\textsc{FullVerify}(\mathit{vk}, \mathrm{x}, \sigma)$ as id
23: $\quad$ **if** $\mathrm{b} \neq 1$ **then** skip
24: $\quad$ **if** $(\mathrm{P}, \mathrm{s}) \in \mathtt{seen}[\mathtt{id}.\mathrm{P}]$ **then** skip
25: $\quad \mathtt{seen}[\mathtt{id}.\mathrm{P}] \leftarrow \mathtt{seen}[\mathtt{id}.\mathrm{P}] \cup \{(\mathrm{P}, \mathrm{s})\}$
26: $\quad \mathrm{M} \leftarrow \mathrm{M} \cup \{(\mathrm{P}, \mathrm{Q}, \tau, \mathit{msg})\}$
27: **end for**
28: **return** M

The simulator. Fix an occupant $\mathcal{A}$ of the adversary slot. In the real world $\mathcal{A}$ faces the three hybrids; in the ideal world they are gone, and $\mathcal{S}_{\mathcal{A}}$ must present their faces from inside itself. It is an absorption (Definition 4.26, read on the adversary side): $\mathcal{S}_{\mathcal{A}}$ bundles $\mathcal{A}$ with one internal copy each of $\mathcal{F}_{\mathsf{PKI}}$, $\mathcal{F}_{\mathsf{Net}}$ and $\mathcal{F}_{\mathsf{Sig}}^{!}$, runs $\mathcal{A}$ against them exactly

as the real execution would, and drives them from what $\mathcal{F}_{\mathsf{AC}}^{\circ}$ tells it. On an honest SEND notification $\mathcal{F}_{\mathsf{AC}}^{\circ}$ hands it $(\P, Q, \tau, msg)$; it advances the internal `seq` for ¶, forms $x = (\P, Q, \tau, s, msg)$, and replays ρ's own steps into the internal hybrids — an internal $\mathcal{F}_{\mathsf{Sig}}^{!}$.SIGN, whose $\mathcal{A}^{!}$ query the hosted $\mathcal{A}$ answers, and an internal $\mathcal{F}_{\mathsf{Net}}$.SEND, whose notification it delivers — so the hosted $\mathcal{A}$ sees precisely the queries the real world would place. When $\mathcal{F}_{\mathsf{AC}}^{\circ}$ makes its responsive fetch query, $\mathcal{S}_{\mathcal{A}}$ runs ρ.FETCH's filter over the internal $\mathcal{F}_{\mathsf{Net}}$'s delivery against the hosted $\mathcal{A}$, learns exactly which records ρ would hand up, and answers (S, I): the ripe or early-released honest entries by their $\mathcal{F}_{\mathsf{AC}}^{\circ}$ ids in S, and the corrupt-sender records — those verifying under a key registered at a corrupt slot — in I. Corruption it passes straight through, CORRUPT surviving the bundle as Remark 3.2's does.

Made explicit: two pieces of state the paragraph above passes over silently, both forced by the code just above it rather than chosen. $\mathcal{F}_{\mathsf{AC}}^{\circ}$'s own SEND (inherited from $\mathcal{F}_{\mathsf{AC}}$, Section 19.1) notifies the slot with $(\text{SEND}, m, Q, \tau, msg)$, m its own list index — not $(\P, Q, \tau, msg)$ alone, the sender ¶ read off the identity the query arrives at (Definition 9.2). The simulator $\mathcal{S}_{\mathcal{A}}$ must keep that index, `fid` below, to name the entry later; it cannot recover it at fetch time, having no access to $\mathcal{F}_{\mathsf{AC}}^{\circ}$.`list` itself. And ρ's own tables — `gen` at line 5, `seq`, and the `seen` its freshness test reads at line 24 — are tables $\mathcal{S}_{\mathcal{A}}$ does not host, ρ being no part of the ideal world and only its subroutines internal here, so reproducing its filter needs shadows of its own, kept equal to ρ's by construction rather than by reading them.

Two routing questions the prose leaves open, and both decide whether the simulator works at all. The first is what becomes of a call the hosted $\mathcal{A}$ places on ρ's *own* identity. Line 3 admits $\mathcal{A}$ at a corrupt party and neither mediation hook fires on its calls, so in the real world such a call reaches ρ's core — and every subroutine call that core then places is mediated straight back to $\mathcal{A}$ (line 4, the caller being ρ and the party corrupt), so nothing of it ever reaches $\mathcal{F}_{\mathsf{Net}}$'s buffer. Relayed outward it would instead reach $\mathcal{F}_{\mathsf{AC}}^{\circ}$, whose SEND files a record that *ripens* and is delivered by Δ whatever the slot answers (Section 19.1) — a delivery with no real-world counterpart. So these calls are not relayed: Definition 4.13's explicit routing keeps them inside, where the last handler below runs ρ's core against the hosted $\mathcal{A}$ exactly as MEDIATE does really. The outward set is declared accordingly.

The second is how far a responsive answer reaches. Both handlers

answer a responsive call, and RESPOND (Definition 9.1) refuses every call the responder *places*; calls between two hosted machines are not placed at all (Definition 4.13), so the internal copies are reachable, but nothing outside the bundle is. Two things outside it would otherwise be reached, and $\mathcal{S}_{\mathcal{A}}$'s code avoids each.

A hosted copy's *full* interface opens by reading the corruption register, which Definition 9.2 refuses outright — a responsive answer is computed from what the responder already knows. So the handlers drive the copies through their *cores*, written GEN rather than FULLGEN above, and nothing is lost by it: a party corrupt at the send or the fetch is mediated before $\mathcal{F}^{\circ}_{\mathsf{AC}}$'s own core runs, so no slot query is ever issued on its behalf and no handler ever serves one, and at an honest party the full interface adds only its silencer — the gate passes, and neither hook fires. And $\mathcal{F}_{\mathsf{Net}}$ opens both SEND and FETCH by reading $\mathcal{G}_{\mathsf{Clock}}$, which is shared rather than hosted. That read is answered with the τ the notification or query already carries, which is the right value and not merely a convenient one: on the send it is the stamp the proof shows ρ's and $\mathcal{F}_{\mathsf{Net}}$'s own reads share, and on the fetch it is $\mathcal{F}^{\circ}_{\mathsf{AC}}$'s read, which no responsive call can have advanced past (Proposition 9.3).

Simulator $\mathcal{S}_{\mathcal{A}}$

occupies IDs($\mathcal{A}$); hosts $\mathcal{A}$ and one internal copy each of $\mathcal{F}_{\text{PKI}}$, $\mathcal{F}_{\text{Net}}$, $\mathcal{F}^{!}_{\text{Sig}}$ (Definition 4.26, read on the adversary side); outward set the identities the ideal execution still serves, less ρ's own, which the last handler routes by hand (Definition 4.13)

INITIALIZE():

1: `gen` : $\mathbf{P} \to \{0,1\}$; `gen`$[*] \leftarrow 0$
2: `seq` : $\mathbf{P} \to \mathbb{N}$; `seq`$[*] \leftarrow 0$
3: `seen` : $\mathbf{P} \to \mathcal{P}(\mathbf{P} \times \mathbb{N})$; `seen`$[*] \leftarrow \varnothing$
4: `fid` : $\mathbf{P} \times \mathbb{N} \to \mathbb{N} \cup \{\square\}$; `fid`$[*] \leftarrow \square$ // *own record $\mapsto$ $\mathcal{F}^{\circ}_{\text{AC}}$'s index*

on $\mathcal{F}^{\circ}_{\text{AC}}$'s SEND note (m, Q, τ, msg) at ¶

5: **if** `gen`[¶] $= 0$ **then**
6: $vk \leftarrow$ internal `id`$_{\mathcal{F}_{\text{Sig}}}$.GEN()
7: internal `id`$_{\mathcal{F}_{\text{PKI}}}$.REGISTER($vk$)
8: `gen`[¶] $\leftarrow 1$
9: `seq`[¶] $\leftarrow$ `seq`[¶] $+ 1$; $s \leftarrow$ `seq`[¶]
10: $x \leftarrow (¶, Q, \tau, s, msg)$
11: $\sigma \leftarrow$ internal `id`$_{\mathcal{F}_{\text{Sig}}}$.SIGN($x$) // *hosted $\mathcal{A}$ answers the query*
12: internal `id`$_{\mathcal{F}_{\text{Net}}}$.SEND($(x, \sigma)$) // *its clock read answered τ*
13: `fid`[¶, s] $\leftarrow m$

on $\mathcal{F}^{\circ}_{\text{AC}}$'s responsive FETCH query (τ) at Q

14: $N \leftarrow$ internal `id`$_{\mathcal{F}_{\text{Net}}}$.FETCH() // *clock answered τ; $\mathcal{A}$ answers the release query*
15: $S, I \leftarrow \varnothing, \varnothing$
16: **for** $(m, y) \in N$ **do**
17: parse y as (x, σ), $x = (¶, Q', \tau_x, s, msg)$ // *else skip*
18: **if** $Q' \neq Q$ **then** skip
19: $vk \leftarrow$ internal `id`$_{\mathcal{F}_{\text{PKI}}}$.RETRIEVE(¶)
20: **if** $vk = \square$ **then** skip
21: $b \leftarrow$ internal `id`$_{\mathcal{F}_{\text{Sig}}}$.VERIFY($vk, x, \sigma$)
22: **if** $b \neq 1$ **then** skip
23: **if** $(¶, s) \in$ `seen`[Q] **then** skip
24: `seen`[Q] $\leftarrow$ `seen`[Q] $\cup \{(¶, s)\}$
25: **if** `fid`[¶, s] $\neq \square$
26: **then** $S \leftarrow S \cup \{$`fid`[¶, s]$\}$
27: **else** $I \leftarrow I \cup \{(¶, Q, \tau_x, msg)\}$
28: **end for**
29: **return** (S, I)

on $\mathcal{A}$'s SEND or FETCH at a corrupt ¶

30: run ρ's own core for that operation (the protocol box above) on `gen`, `seq`, `seen`, with every subroutine call it places — to $\mathcal{F}^{!}_{\text{Sig}}$, $\mathcal{F}_{\text{PKI}}$, $\mathcal{F}_{\text{Net}}$ or $\mathcal{G}_{\text{Clock}}$ — handed to the hosted $\mathcal{A}$ rather than placed, as the full interface's line 4 hands it really; return the core's own value
31: never relayed outward: no $\mathcal{F}^{\circ}_{\text{AC}}$.`list` entry, and none drained

Routing, and CORRUPT

32: any other call between the hosted $\mathcal{A}$ and an internal hybrid: served inside, the bundle's ordinary routing (Definition 4.13). A CORRUPT leaves under the hosted $\mathcal{A}$'s own claim, the bundle occupying IDs($\mathcal{A}$), so CORRUPT's own gate `id'`.F $=$ A (its line 3) is met without rewriting

Three things about $\mathcal{S}_{\mathcal{A}}$'s code, all used by the proof but easy to read past. First, `fid`[¶, s] $\neq \square$ at line 26 is exactly "an undrained `list` entry matches the record" in the proof's own words: $\mathcal{S}_{\mathcal{A}}$ recognizes a record precisely when it once relayed the honest SEND that produced it, which is the only way an entry gets into `fid` at all.

Second, `gen` is a shadow of ρ's own flag and not a reading of the internal $\mathcal{F}_{\text{PKI}}$'s `VK`, though at an honest party the two agree — $\mathcal{F}_{\text{PKI}}$'s

REGISTER turns an A- or Z-caller's call there into a peek at its line 4, so only ρ ever writes that entry. They part company at a corrupt one, which is why the shortcut is not taken: there ρ's REGISTER is mediated away before $\mathcal{F}_{\mathsf{PKI}}$'s core runs, leaving $\mathtt{VK}[\P]$ unset while ρ sets $\mathtt{gen}[\P] \leftarrow 1$ regardless, ρ.SEND's line 5 not reading the answer. A simulator testing VK would re-run GEN and REGISTER on the party's every later send and show the hosted $\mathcal{A}$ a pair of queries the real $\mathcal{A}$ never sees.

Third, line 26 puts *every* matching honest record into S, ripe ones included, where $\mathcal{F}^{\circ}_{\mathsf{AC}}$ would have taken the ripe ones through R on its own. This is deliberate and costs nothing: $\textsc{Clean}_{\textsc{addr}}$ admits any undrained entry addressed to Q, and $\mathcal{F}^{\circ}_{\mathsf{AC}}$.FETCH unions R with S at its line 7 and drains their union, so naming a ripe entry twice delivers and drains it once. Naming it is also all $\mathcal{S}_{\mathcal{A}}$ can do, having no clock read of its own inside the query with which to tell ripe from early.

Theorem 20.1 (ρ realizes the authenticated channel)**.** *Let* $\pi := \{\rho, \mathcal{F}_{\mathsf{PKI}}, \mathcal{F}_{\mathsf{Net}}, \mathcal{F}^{!}_{\mathsf{Sig}}\}$ *with parameters matched so that* $\pi \cup \{\mathcal{G}_{\mathsf{Clock}}\}$ *is well-formed,* $\rho.\mathbf{N} = \mathcal{F}^{\circ}_{\mathsf{AC}}.\mathbf{N}$*, and* $\mathcal{F}_{\mathsf{Net}}.par = \mathcal{F}^{\circ}_{\mathsf{AC}}.par = \Delta$*. Then* π *UC-emulates* $\{\mathcal{F}^{\circ}_{\mathsf{AC}}\}$ *within* 0*: for every occupant* $\mathcal{A}$ *of the adversary slot the simulator* $\mathcal{S}_{\mathcal{A}}$ *above gives, for every* $\mathcal{Z} \in \mathbb{Z}_{\pi,\{\mathcal{F}^{\circ}_{\mathsf{AC}}\}}$*,*

$$\textsc{Exec}(\pi, \mathcal{Z}, \mathcal{A}, \mathcal{C}) \equiv \textsc{Exec}(\{\mathcal{F}^{\circ}_{\mathsf{AC}}\}, \mathcal{Z}, \mathcal{S}_{\mathcal{A}}, \mathcal{C}),$$

equal as distributions. The quantifier is over every *adversary, not the dummy alone: every slot call in either world is responsive, so Theorem 4.29 does not reach this and the simulator is exhibited against each* $\mathcal{A}$ *directly.*

Proof. One induction on activations, carrying an invariant in three parts. The coupling pairs machine to machine as in Lemma 4.18, so the routing, claims and refusal arguments are that lemma's; what is new is the state carried across the $\rho/\mathcal{F}^{\circ}_{\mathsf{AC}}$ cut, and the invariant on it is the whole of the argument. *The invariant* states the three parts: the internal hybrids hold what the real ones hold, $\mathcal{S}_{\mathcal{A}}$'s shadows hold what ρ holds, and $\mathcal{F}^{\circ}_{\mathsf{AC}}.\mathtt{list}$ holds exactly the undrained honest sends. Three cases preserve it. *Send* shows the two worlds stamp a send at the same time, and that replaying ρ's own steps into the internal copies keeps them equal. *Fetch* carries the weight: it splits not on the sender's current status but on whether an undrained `list` entry matches the

record, and shows that the unmatched case forces a corrupt sender — which is what unforgeability buys here, and why $\textsc{Clean}_{\text{INJ}}$ never trips. *Corrupt-party traffic* covers the one activation the declared outward set withholds, and shows that withholding it is what the invariant needs. Couple the two executions on equal tapes, paired machine to paired machine as in Lemma 4.18: $\mathcal{Z}$, $\mathcal{C}$ and $\mathcal{G}_{\mathsf{Clock}}$ each to itself, the hosted $\mathcal{A}$ and the internal hybrids of $\mathcal{S}_{\mathcal{A}}$ to the real $\mathcal{A}$ and the real hybrids, and ρ together with what the hybrids hold to $\mathcal{F}^{\circ}_{\mathsf{AC}}$. Everything below is one induction on activations, in the shape of Definition 4.17; the claims and refusal paragraphs are Lemma 4.18's verbatim, and its routing paragraph holds of every call but one — $\mathcal{S}_{\mathcal{A}}$ is a bundle, and the calls that cross its boundary are the ordinary outward and inward relays, save those the hosted $\mathcal{A}$ places on ρ's own identities, which the declared outward set withholds and the last handler serves inside — so only the state carried across the $\rho/\mathcal{F}^{\circ}_{\mathsf{AC}}$ cut is new, and the invariant on it is the whole of the argument.

The invariant. At every corresponding pair of configurations: (i) the internal $\mathcal{F}_{\mathsf{PKI}}$, $\mathcal{F}_{\mathsf{Net}}$ and $\mathcal{F}^{!}_{\mathsf{Sig}}$ of $\mathcal{S}_{\mathcal{A}}$ hold the same tables as the real hybrids, having been initialised alike and driven by the same calls in the same order; (ii) $\mathcal{S}_{\mathcal{A}}$'s shadow `gen`, `seq` and `seen` equal ρ's; and (iii) $\mathcal{F}^{\circ}_{\mathsf{AC}}$.`list` holds exactly the records $(\mathsf{P}, \mathsf{Q}, \tau, \mathit{msg})$ of the sends ρ made through a party's own SEND while that party was honest — one entry per such ρ.SEND, at the timestamp it read — that no fetch has since drained, and no others. Later corruption of the sender does not disturb the entry, corruption being monotone and the record already placed. (i) and (ii) hold because paired copies run the same code on equal tapes under the same driving: at an honest party ρ's core drives the real hybrids and $\mathcal{S}_{\mathcal{A}}$'s handlers drive their copies through the same steps, and at a corrupt one both sides mediate the same subroutine calls away before any callee's core runs, as the third case below checks. (iii) is what the three cases preserve.

Send. $\mathcal{Z}$ calls $\textsc{Send}(\mathsf{Q}, \mathit{msg})$ at an honest ¶. On the ideal side $\mathcal{F}^{\circ}_{\mathsf{AC}}$.SEND reads the clock, appends $(¶, \mathsf{Q}, \tau, \mathit{msg})$ to `list`, and notifies $\mathcal{S}_{\mathcal{A}}$ responsively. On the real side ρ.SEND reads the clock, and its internal $\mathcal{F}_{\mathsf{Net}}$.SEND reads it again; the two clock reads coincide, by the argument of Section 18.2 — no core of either execution runs between them, $\mathcal{G}_{\mathsf{Clock}}$.READ placing no call — so the send is stamped τ on both sides. $\mathcal{S}_{\mathcal{A}}$, notified, replays exactly the GEN/REGISTER/SIGN/SEND steps ρ runs, into copies that (i) keeps equal to ρ's — the internal $\mathcal{F}_{\mathsf{Net}}$'s

own clock read answered with the τ the notification carried, which is that same stamp, since the responder may place no call of its own to make it — and the hosted $\mathcal{A}$ answers each $\mathcal{A}^!$ query as the real $\mathcal{A}$ does over equal tapes. Both return OK; one new `list` entry matches the one send; (iii) holds.

Fetch. $\mathcal{Z}$ calls FETCH() at Q. Everything turns on the two worlds delivering the same set M, and on $\mathcal{S}_\mathcal{A}$ being *able* to name it within $\text{CLEAN}_{\text{INJ}}$. Run ρ.FETCH's filter, which $\mathcal{S}_\mathcal{A}$ mirrors internally: a delivered record (P, Q, τ, msg) is one whose diffused tuple (x, σ), $x = (P, Q, \tau, s, msg)$, is addressed to Q, carries a signature verifying under $\texttt{VK}_{\mathcal{F}_{\text{PKI}}}[P]$, and is fresh in `seen`[Q]. Split not on P's current status but on whether an undrained `list` entry matches the record — the honest case and the injected case being exactly these two.

If one matches: by (iii) it is a send ρ made for P while P was honest, and $\mathcal{F}_{\text{AC}}^\circ$ delivers it from store — through R if ripe, and through S, its id admitted by $\text{CLEAN}_{\text{ADDR}}$ as addressed to Q, if $\mathcal{S}_\mathcal{A}$ releases it early to match $\mathcal{F}_{\text{Net}}$'s early release. Whether or not the sender is corrupt by now, the record is delivered from `list` and never injected, so none is delivered twice.

If none matches: then P is corrupt at the fetch, and this is where unforgeability earns its place. Were P honest at the fetch it would be honest throughout, by monotonicity, so $\texttt{VK}_{\mathcal{F}_{\text{PKI}}}[P]$ would be P's own $\mathcal{F}_{\text{Sig}}$ key — ρ registers exactly what GEN returned — and a signature verifying under an honest party's key is never a forgery: VERIFY's honest-key test (Section 15.1) sets $b \leftarrow 0$ on any fresh triple under such a key, so $b = 1$ forces the triple recorded, and only an honest SIGN records one, which by (iii) would leave a matching `list` entry — against the case. So $P \in \mathbf{C}$; the record is the adversary's to have produced, an $\mathcal{F}_{\text{Net}}$ injection or a signature made as a corrupt party, and $\mathcal{S}_\mathcal{A}$ places (P, Q, τ, msg) in I, which $\text{CLEAN}_{\text{INJ}}$ admits, P being corrupt and Q the fetcher. The channel $\mathcal{F}_{\text{AC}}^\circ$ delivers it unstored, as ρ hands it up. This is what unforgeability buys: no record lacking a genuine send behind it can ever wear an honest name, so the injection channel is only ever asked for corrupt senders, and $\text{CLEAN}_{\text{INJ}}$ never trips.

Freshness matches on both sides by (ii), and draining matches: a stored record delivered is drained from `list` ideally and marked seen really, so $\mathcal{F}_{\text{Net}}$'s re-delivery of the same tuple is filtered by `seen` really and finds no `list` entry ideally, falling into neither case a second time. So the two FETCHes return the same M and step to corresponding

configurations; (iii) is preserved, its stored entries drained in lockstep.

Corrupt-party traffic. $\mathcal{A}$ calls SEND or FETCH at a corrupt ¶, on ρ's own identity — the one activation the outward set withholds, and the case the two above do not reach. The full interface admits it at line 3, ¶ being corrupt, and neither hook fires on an A-caller, so on the real side ρ's core runs. Every subroutine call that core then places carries ρ's name at a corrupt party, so line 4 of that same interface hands each to $\mathcal{A}$ before the callee's core runs: nothing of the activation reaches $\mathcal{F}_{\mathsf{Net}}$'s buffer, $\mathcal{F}_{\mathsf{PKI}}$'s directory or $\mathcal{F}_{\mathsf{Sig}}$'s tables, and its very timestamp is $\mathcal{A}$'s to supply. $\mathcal{S}_{\mathcal{A}}$'s line 30 runs that same core against the hosted $\mathcal{A}$ on the same shadows, so both sides move `gen`, `seq` and `seen` alike and leave the hybrids untouched, preserving (i) and (ii); and neither side touches `list` — ideally because the call never leaves $\mathcal{S}_{\mathcal{A}}$, really because there is no $\mathcal{F}^{\circ}_{\mathsf{AC}}$ to touch — so (iii) is preserved with nothing to check.

Withholding these calls is what (iii) needs, and it is worth saying why rather than leaving it to the routing. Relayed outward the call would reach $\mathcal{F}^{\circ}_{\mathsf{AC}}$.SEND, which files a record at the true time; the ripeness test $\mathcal{F}^{\circ}_{\mathsf{AC}}$ inherits at $\mathcal{F}_{\mathsf{AC}}$.FETCH's line 12 would then deliver it to Q within Δ whatever the slot answered, where the real world delivers nothing at all unless $\mathcal{A}$ injects on its own account. The entry would break (iii) the moment it was filed, and no later answer of $\mathcal{S}_{\mathcal{A}}$'s could retract it — R being $\mathcal{F}^{\circ}_{\mathsf{AC}}$'s to compute. This is the ripening that Section 19.1 charges to a corrupt sender speaking through $\mathcal{F}_{\mathsf{AC}}$.SEND, and a protocol whose corrupt parties speak through $\mathcal{F}_{\mathsf{Net}}$ instead does not pay it.

No other activation touches the cut: a call among $\mathcal{Z}$, $\mathcal{G}_{\mathsf{Clock}}$, $\mathcal{C}$ and $\mathcal{A}$ that targets none of ρ's identities is routed and claimed identically on both sides by Lemma 4.18, and reads no state the invariant governs. Corresponding halting configurations return $\mathcal{Z}$'s one output. The executions are equal as distributions, and the advantage is 0 — not a bound but an identity, every step deterministic once the tapes are fixed and no forgery ever slipping through to force an injection the sanitizer would refuse. □

The bound is 0, and it is worth being clear on why nothing weaker was available and nothing stronger was needed. Every subroutine is ideal, so there is no computational gap to charge for; the reduction is an identity of distributions, as the composition theorem's is. Transfer, note, does not run here: Corollary 5.11 wants cores placing no respon-

sive call, and ρ's reach $\mathcal{F}_{\mathsf{Sig}}^{!}$, $\mathcal{F}_{\mathsf{Net}}$ and the clock all responsively, so a property of $\mathcal{F}_{\mathsf{AC}}^{\circ}$ does not descend to π for free — it must be re-proved over π, exactly as Section 17.2 warned for anything built over the notifying clock. What the theorem gives instead is the stronger thing for a realization: not a bound against the dummy but an equality against every adversary.

Corollary 20.2 (Against non-corrupting classes, ρ realizes $\mathcal{F}_{\mathsf{AC}}$ itself)**.** *Instrument $\mathcal{F}_{\mathsf{AC}}^{\circ}$ to set* bad $\leftarrow$ TRUE *at line 7 when $\mathrm{I} \neq \varnothing$, and let $\mathcal{F}_{\mathsf{AC}}^{\flat}$ be the pair-shaped $\mathcal{F}_{\mathsf{AC}}$ that ignores I — its fetch $\mathcal{F}_{\mathsf{AC}}$'s, taking (S, I) and using S alone. Now $\mathcal{F}_{\mathsf{AC}}^{\circ}$ and $\mathcal{F}_{\mathsf{AC}}^{\flat}$ are identical until bad (Definition 11.1), the injection the one bad-guarded difference. So for occupants and environments that corrupt nobody,* $\textsc{Clean}_{\textsc{inj}}$ *forces $\mathrm{I} = \varnothing$ and $\Pr[\textsc{Bad}] = 0$, whence by Lemma 11.2 the two are indistinguishable and π UC-emulates $\{\mathcal{F}_{\mathsf{AC}}^{\flat}\}$, hence $\{\mathcal{F}_{\mathsf{AC}}\}$, within 0 against the non-corrupting classes. Corrupt a party and the guarantee is $\mathcal{F}_{\mathsf{AC}}^{\circ}$'s, no less: the honest-sender half of* FINAUTH *(Section 19.2) transfers, its* SEND*-tracing argument reading only honest records, while the full* FINAUTH *fails of $\mathcal{F}_{\mathsf{AC}}^{\circ}$ exactly as it does of $\mathcal{F}_{\mathsf{Net}}$, and for the same reason.*

Remark 20.3 (The weakening is forced)**.** No protocol over $\{\mathcal{F}_{\mathsf{PKI}}, \mathcal{F}_{\mathsf{Net}}, \mathcal{F}_{\mathsf{Sig}}^{!}\}$ realizes $\mathcal{F}_{\mathsf{AC}}$ as shipped, so $\mathcal{F}_{\mathsf{AC}}^{\circ}$ is not a convenience but the true target. Suppose one did. A corrupt party diffuses, inside a fetch's responsive window, a well-formed item its receiver accepts — $\mathcal{F}_{\mathsf{Net}}$ grants exactly this, the wire being the adversary's to write (Section 18.1) — carrying a timestamp τ^{*} of the adversary's choosing. On the ideal side the accepting fetch is atomic, so the simulator must have called $\mathcal{F}_{\mathsf{AC}}$.SEND strictly earlier to place a matching entry; but $\mathcal{F}_{\mathsf{AC}}$ stamps that call the true time it was made, not τ^{*}, and the receiver reads τ^{*} off the delivered record. An environment that sends at one time and has its corrupt confederate claim another distinguishes with certainty. The freedom is $\mathcal{F}_{\mathsf{Net}}$'s to give and no receiver-side check removes it, since the signature validates whatever time the payload carries; only weakening the ideal object to admit it — which is $\mathcal{F}_{\mathsf{AC}}^{\circ}$ — closes the gap. This is the diffusion analogue of the impossibility that forces $\mathcal{F}_{\mathsf{Net}}$'s own injections to sit outside its delivery guarantee.

This is the paper's longest single construction, and the point of running it in full is that every layer of the framework shows up load-bearing. Composition supplies the cut between ρ and its hybrids; the responsive discipline of Chapter 9 makes both fetches atomic so the schedules can be coupled at all; sanitization confines a corrupt slot to corrupt senders; tightness would be what a property of $\mathcal{F}_{\mathsf{AC}}^{\circ}$ needs to descend, and its absence under responsive cores is why the realization is proved directly rather than inherited; and the fundamental lemma of Chapter 11 is what turns the exact realization of the weaker object into the exact realization of the stronger one wherever no corruption is in play. The authenticated channel, postulated in Chapter 19, is now a theorem.

CHAPTER 21

$\mathcal{F}_{\mathsf{Sig}}$: A Deeper Dive

Chapter 15 gave a signature functionality and proved two properties of it. This chapter asks the three questions that leaves open. What does a *scheme* have to satisfy for a protocol built on it to inherit those properties — and what does the protocol look like, drawn in this framework's own terms, with its randomness and its state held where Section 3.2 says they must be? Does the converse hold: are the two properties *enough* to characterize $\mathcal{F}_{\mathsf{Sig}}$, so that anything with its interfaces satisfying them realizes it? And is $\mathcal{F}_{\mathsf{Sig}}$ the strongest object of its shape, or does a realization of it satisfy something it does not?

The answers are: yes with a scheme's own security as the bound; no, and instructively so; and no — a realization can be strictly stronger, which is worth knowing about ideal functionalities in general.

21.1 Signature schemes and their games

A *signature scheme* is a triple of algorithms $\mathsf{DS} = (\textsc{Gen}, \textsc{Sign}, \textsc{Ver})$ over a key space $\mathcal{K}$, a message space $\mathcal{M}$, a signature space Σ and a coin space $\{0,1\}^n$, with $\textsc{Gen}$ and $\textsc{Sign}$ taking their coins explicitly:

$$(vk, \mathtt{sk}) \leftarrow \textsc{Gen}(\mathtt{r}), \qquad \sigma \leftarrow \textsc{Sign}(\mathtt{sk}, msg; \mathtt{r}),$$

$$\mathtt{b} \leftarrow \textsc{Ver}(vk, msg, \sigma),$$

$\textsc{Ver}$ deterministic. The coins are arguments rather than an implicit tape because π_{Sig} below draws them through $\mathcal{F}_{\mathsf{Rand}}$, and a scheme whose randomness is hidden inside it could not be wired to a randomness functionality at all.

Its three security notions are games in the ordinary property-based sense, each played by an adversary $\mathcal{B}$ against DS alone — no framework, no execution, the notions a cryptographer would write down anyway, unforgeability under chosen-message attack chief among

them [10]. One shell, one signing oracle, shared by all three in the same idiom the paper's own game systems share a shell across several finalizations (Definition 5.1): $\mathcal{B}$ receives vk, queries SIGN as it likes, then halts by calling exactly one of the three procedures below, which decides whether it wins.

Game for a signature scheme DS: the shell

INITIALIZE():
1: $(vk, \mathtt{sk}) \leftarrow \mathsf{DS.GEN}(\mathtt{r})$ // r uniform
2: **return** vk

SIGN(msg):
3: $\sigma \leftarrow \mathsf{DS.SIGN}(\mathtt{sk}, msg; \mathtt{r})$ // r uniform
4: $\mathtt{tr} \leftarrow \mathtt{tr} \cdot (msg, \sigma)$
5: **return** σ

Finalizations of the game, one per notion; the shell above is shared

COR(msg) (correctness)
6: $\sigma \leftarrow$ SIGN(msg) // a fresh call, on the game's own oracle
7: $w \leftarrow 1$ if $\mathsf{DS.VER}(vk, msg, \sigma) \neq 1$; else $w \leftarrow 0$
8: **return** w

UF(msg, σ) (unforgeability)
9: $w \leftarrow 1$ if $\mathsf{DS.VER}(vk, msg, \sigma) = 1 \wedge (msg, \sigma) \notin \mathtt{tr}$; else $w \leftarrow 0$
10: **return** w

COL(msg, msg', σ) (collision resistance)
11: $w \leftarrow 1$ if $msg \neq msg' \wedge \mathsf{DS.VER}(vk, msg, \sigma) = \mathsf{DS.VER}(vk, msg', \sigma) = 1$; else $w \leftarrow 0$
12: **return** w

Write $\varepsilon_{\mathrm{COR}}(\mathcal{B})$, $\varepsilon_{\mathrm{UF}}(\mathcal{B})$ and $\varepsilon_{\mathrm{COL}}(\mathcal{B})$ for the probabilities that COR, UF and COL return 1. Perfect correctness is $\varepsilon_{\mathrm{COR}} = 0$, which most schemes have and none of what follows needs. Unforgeability is *strong* [1]: line 9 excludes the pair the oracle returned rather than the message alone, matching FINUF's reading in Section 15.2. Collision resistance is not standard and is not implied by the other two; Section 21.4 is where it earns its place.

21.2 The protocol π_{Sig}

π_{Sig} carries $\mathcal{F}_{\mathsf{Sig}}$'s three interfaces and holds nothing of its own: every coin is drawn through $\mathcal{F}_{\mathsf{Rand}}$ and every value that outlives an invo-

cation sits in $\mathcal{F}_{\text{Store}}$, which is Section 3.2's *leakage-well-formed* shape and the first protocol in this paper written to it. That is not decoration. Leakage is gated on the party being corrupt (line 4) but a functionality's LEAK hands over its state *entire*, so a single $\mathcal{F}_{\text{Rand}}$ shared across parties would surrender every party's coins the moment any one party was corrupted, and with them every signing key — no unforgeability could survive it. The protocol π_{Sig} therefore addresses *one instance of each per party*, $\mathcal{F}_{\text{Rand}}^{\text{P}}$ and $\mathcal{F}_{\text{Store}}^{\text{P}}$ with $\mathbf{P} = \{\text{P}\}$, which Remark 5.5 already allows and the guard's blindness to the instance number already reaches. And it erases each coin the moment it has been used, which is what $\mathcal{F}_{\text{Rand}}$'s ERASE was put there for (Section 13.1): what has been erased cannot leak, so a corrupt party surrenders its own signing key and nothing else — not even its own past coins.

Protocol π_{Sig} over $\{\mathcal{F}_{\text{Rand}}^{\text{P}}, \mathcal{F}_{\text{Store}}^{\text{P}}\}_{\text{P}\in\mathbf{P}}$
pid, **P**, **N**, **U** := $\{(\mathcal{F}_{\text{Rand}}, \text{SERVES}), (\mathcal{F}_{\text{Store}}, \text{SERVES})\}$, *par* := DS

id.GEN() from id′
1: $vk \leftarrow \mathtt{id}_{\mathcal{F}_{\text{Store}}}$.FULLGET(VK) as id
2: **if** $vk \neq$ REJ **then**
3: **return** vk // one key per party
4: $(\mathtt{i}, \mathtt{r}) \leftarrow \mathtt{id}_{\mathcal{F}_{\text{Rand}}}$.FULLRND() as id
5: $(vk, \mathtt{sk}) \leftarrow$ DS.GEN($\mathtt{r}$)
6: $\mathtt{id}_{\mathcal{F}_{\text{Rand}}}$.FULLERASE($\mathtt{i}$) as id // the coins cannot leak
7: $\mathtt{id}_{\mathcal{F}_{\text{Store}}}$.FULLPUT(VK, vk) as id
8: $\mathtt{id}_{\mathcal{F}_{\text{Store}}}$.FULLPUT(SK, $\mathtt{sk}$) as id
9: **return** vk

id.VERIFY(vk, msg, σ) from id′
10: **return** DS.VER(vk, msg, σ)

id.SIGN(msg) from id′
11: $\mathtt{sk} \leftarrow \mathtt{id}_{\mathcal{F}_{\text{Store}}}$.FULLGET(SK) as id
12: **if** $\mathtt{sk} =$ REJ **then**
13: **return** $\perp$ // no key yet
14: $(\mathtt{i}, \mathtt{r}) \leftarrow \mathtt{id}_{\mathcal{F}_{\text{Rand}}}$.FULLRND() as id
15: $\sigma \leftarrow$ DS.SIGN($\mathtt{sk}, msg; \mathtt{r}$)
16: $\mathtt{id}_{\mathcal{F}_{\text{Rand}}}$.FULLERASE($\mathtt{i}$) as id // erased as soon as used
17: **return** σ

id.LEAK() from id′
18: **return** $\perp$ // it holds nothing to leak

The draws come back as pairs because $\mathcal{F}_{\text{Rand}}$'s RND answers with the index it wrote beside the string (line 7), which is what lets π_{Sig} name the entry it then erases; FINUNP reads the strings and is indifferent to the index travelling beside them.

Three remarks. π_{Sig}'s own LEAK returns $\perp$ because it has no state to give: the key is $\mathcal{F}_{\text{Store}}^{\text{P}}$'s, and a corrupt party's adversary reads it there, which is precisely what leakage-well-formedness asks — the leak is where the state is, and nowhere else. The operation VERIFY places no call at all, VER being deterministic, so a verification is local and neither draws nor stores; that is why π_{Sig} is the one protocol here whose VERIFY cannot be distinguished from $\mathcal{F}_{\text{Sig}}$'s by the token's movement. And the key is fetched from the store rather than cached

in π_{Sig}, which is what makes "one key per party" a fact about $\mathcal{F}_{\mathsf{Store}}$'s per-slot fixity (Section 14.2, FINKEEP) rather than a promise π_{Sig} makes about itself.

Proposition 21.1 (π_{Sig} inherits the two properties). *Let* $\gamma := \{\mathcal{GM}\} \cup \pi_{\mathsf{Sig}} \cup \{\mathcal{F}^{\mathsf{P}}_{\mathsf{Rand}}, \mathcal{F}^{\mathsf{P}}_{\mathsf{Store}}\}_{\mathsf{P}}$ *be a game system in the sense of Definition 5.1, with* $\mathcal{GM}$ *the shell of Section 15.2. For every* $\mathcal{Z} \in \mathbb{Z}_{\gamma,\gamma}$ *placing at most* q *calls on the relays and every occupant* $\mathcal{X}$ *of the adversary slot there are adversaries* $\mathcal{B}_{\mathrm{COR}}$ *and* $\mathcal{B}_{\mathrm{UF}}$ *against* DS*, each running* $\mathcal{Z}$ *and* $\mathcal{X}$ *once with* $O(q)$ *overhead, such that*

$$\Pr[\mathrm{WIN}_{\mathrm{FINCOR}}] \leqslant q \cdot \varepsilon_{\mathrm{COR}}(\mathcal{B}_{\mathrm{COR}}),$$

$$\Pr[\mathrm{WIN}_{\mathrm{FINUF}}] \leqslant q \cdot \varepsilon_{\mathrm{UF}}(\mathcal{B}_{\mathrm{UF}}) + q^2 2^{-n}.$$

Proof. One preliminary paragraph and two reductions. The preliminary collects what the subroutines already give: tightness makes every trace entry an answer π_{Sig}'s own core gave at an honest party, Proposition 14.1 makes the store fixed and faithful so one key is issued per party for good, and Proposition 13.1 makes the coins uniform and, being erased, unreadable. Each property then reduces to the scheme's own game by the same move: assume the verdict fires, identify the triple it names as a failure of DS outright, and have the reduction guess in advance which of the at most q calls is the failing one — which is where the factor q comes from, in both bounds. By tightness no caller but $\mathcal{GM}$ reaches π_{Sig}, so every GEN, SIGN and VER entry of $\mathtt{tr}$ is an answer π_{Sig}'s own core gave at an honest party — at a corrupt one the game's wrapper intercepts and records nothing (Remark 5.4). By Proposition 14.1 the store is fixed and faithful: the vk and $\mathtt{sk}$ a party puts are what it later gets, and GEN's guard on line 1 therefore issues one key per party for good. By Proposition 13.1 the coins are uniform and, being erased at line 6 and its counterpart in SIGN, are read by nobody: $\mathcal{F}^{\mathsf{P}}_{\mathsf{Rand}}$'s leak is empty of them and $\mathcal{F}^{\mathsf{P}}_{\mathsf{Rand}}$ serves P alone, so no corruption of another party reaches them.

For correctness, suppose line 20 fires: honest P, Q, a $(\mathrm{GEN}, \mathsf{P}, vk)$, a $(\mathrm{SIGN}, \mathsf{P}, msg, \sigma)$ with $\sigma \in \Sigma$, and a $(\mathrm{VER}, \mathsf{Q}, vk, msg, \sigma, 0)$. The verification is $\mathsf{DS}.\mathrm{VER}(vk, msg, \sigma)$ by line 10, and the signature is $\mathsf{DS}.\mathrm{SIGN}(\mathtt{sk}, msg; r)$ under the $\mathtt{sk}$ paired with that very vk by line 5, on coins uniform by the paragraph above. So the triple is a correctness failure of DS outright. The adversary $\mathcal{B}_{\mathrm{COR}}$ runs the execution, guesses

in advance which of the at most q signing calls is the failing one, and outputs its message; it wins whenever the game does and the guess is right, which is the factor q.

For unforgeability, suppose line 28 fires: honest P, Q, (GEN, P, vk), (VER, Q, vk, msg, σ, 1) and no (SIGN, P, msg, σ). Two cases. Either vk is the key GEN issued to P, and then VER(vk, msg, σ) = 1 with (msg, σ) returned by no SIGN at P — a strong forgery, which $\mathcal{B}_{\text{UF}}$ harvests by planting its challenge key at a guessed one of the at most q generating parties and answering that party's signing calls from its oracle, every other party it runs itself; or vk is *another* honest party's key that happens to equal P's, and then two independent GEN(r) draws on uniform n-bit coins collided, which over at most q generations is at most $q^2 2^{-n}$. Note where the erasure is spent: without it, corrupting any party would hand $\mathcal{X}$ the coins of P's own GEN and SIGN, and $\mathcal{B}_{\text{UF}}$ could not answer with an oracle it does not have the coins for. □

This is the first bound in the paper that is neither 0 nor information-theoretic, and it is worth saying what has changed. The functionality $\mathcal{F}_{\text{Sig}}$ has its properties because it was written to (Proposition 15.1); π_{Sig} has them because DS does, and the two q-factors are the ordinary price of a guessing reduction. Corollary 5.11 is what makes the exchange legitimate in the other direction — a property of $\mathcal{F}_{\text{Sig}}$ descends to any realization of it — and this proposition is the same statement approached from the realization's side, proved directly because π_{Sig} is not obtained from $\mathcal{F}_{\text{Sig}}$ by emulation but built from a scheme.

21.3 The converse, and what properties cannot reach

Proposition 21.1 runs from a scheme to the properties. The converse would run the other way and much further: that the two properties *characterize* $\mathcal{F}_{\text{Sig}}$, so that any functionality carrying its interfaces and satisfying them realizes it. It is a natural conjecture — properties are what we ask of an ideal object, so an object satisfying them ought to be as good as the one we asked them of — and it is false.

Proposition 21.2 (The two properties do not characterize $\mathcal{F}_{\text{Sig}}$)**.** *There is a standard functionality $\mathcal{F}'$ with $\mathcal{F}_{\text{Sig}}$'s fields and interfaces, having both* FINCOR *and* FINUF *within* 0 *against every environment and every occupant of the slot, such that $\{\mathcal{F}'\}$ does not UC-emulate $\{\mathcal{F}_{\text{Sig}}\}$*

within any $\varepsilon < \frac{1}{2}$.

Proof. Let $\mathcal{F}'$ be $\mathcal{F}_{\mathsf{Sig}}$ with one change: in VERIFY, when no honest party's key equals the queried vk — that is, when $vk \neq \mathtt{VK}[\mathsf{P}']$ for every $\mathsf{P}' \notin \mathbf{C}$ — the answer is a fresh uniform bit, recorded nowhere, so that a later query on the same triple is answered independently. On every other triple $\mathcal{F}'$ is $\mathcal{F}_{\mathsf{Sig}}$ exactly, memoization included.

$\mathcal{F}'$ has both properties within 0. Each of lines 20 and 28 is existential over a $(\text{GEN}, \mathsf{P}, vk)$ entry with $\mathsf{P} \notin \mathbf{C}$, so a triple whose vk is no honest party's key witnesses neither, and on all other triples $\mathcal{F}'$ answers as $\mathcal{F}_{\mathsf{Sig}}$ does, whose verdicts are 0 by Proposition 15.1.

It does not emulate. Let $\mathcal{Z}$ pick any vk^*, msg, σ with vk^* generated by nobody, corrupt no one, and call $\text{VERIFY}(vk^*, msg, \sigma)$ twice at an honest party, returning 1 if the two answers differ. Against $\mathcal{F}'$ the answers are independent uniform bits, so it returns 1 with probability $\frac{1}{2}$. Against $\mathcal{F}_{\mathsf{Sig}}$ the first call fixes $\mathtt{Ver}[vk^*, msg, \sigma]$ and the second returns it, so the answers agree and it returns 1 with probability 0 — whatever the simulator does, the memoization being $\mathcal{F}_{\mathsf{Sig}}$'s own code and no value of the slot's able to unfix it. The advantage is $\frac{1}{2}$. □

The diagnosis is more useful than the counterexample. Emulation pins an object's whole observable behaviour; the two properties pin two things about it — that honestly produced signatures verify, and that forgeries under honest keys do not. Everything else is out of their reach, and the gap has a shape: both verdicts are existential over $(\text{GEN}, \mathsf{P}, vk)$ with P honest, so *nothing whatever* is said about keys no honest party generated, and nothing is said about consistency, one answer being all either verdict inspects. What is missing is not a stronger notion of unforgeability but two further properties, and they are the ones $\mathcal{F}_{\mathsf{Sig}}$'s code makes obvious once looked for: its VERIFY memoizes, and its GEN issues one key per party.

Two further finalizations of the shell of Section 15.2

id.FINCONS() from id′ (consistency)

require ¬DONE;
require id.P = Z; DONE ← TRUE
$w \leftarrow 1$ if $\exists\, Q, Q', vk, msg, \sigma$: (VER, Q, vk, msg, σ, 0) ∈ tr
  ∧ (VER, Q′, vk, msg, σ, 1) ∈ tr; else $w \leftarrow 0$
return w

id.FINONE() from id′ (one key per party)

require ¬DONE;
require id.P = Z; DONE ← TRUE
C ← (C, Z).FULLSTATUS() as id
$w \leftarrow 1$ if $\exists\, P \notin C$, $vk \neq vk'$: (GEN, P, vk) ∈ tr ∧ (GEN, P, vk') ∈ tr;
  else $w \leftarrow 0$
return w

Both take the search reading, both are attained at 0 by $\mathcal{F}_{\mathsf{Sig}}$ — FINCONS because Ver is written once per triple and read thereafter, FINONE because VK[P] is written once and read thereafter, each by the same first-write-wins argument Proposition 16.1 makes for the directory — and both are attained at 0 by π_{Sig}, by VER's determinism and $\mathcal{F}_{\mathsf{Store}}$'s FINKEEP respectively. Neither asks anything of a corrupt party: FINCONS names no party at all, the game's own wrapper having already kept corrupt answers out of tr.

Theorem 21.3 (The converse, from the four properties)**.** *Let $\mathcal{F}'$ be a standard functionality with $\mathcal{F}_{\mathsf{Sig}}$'s fields and interfaces, whose* SIGN *returns $\perp$ exactly when no key has been generated at the calling party, and let $\mathcal{F}'$ have* FINCOR, FINUF, FINCONS *and* FINONE *within* 0 *against $\mathbb{Z}_{\gamma,\gamma}$ and every occupant of the slot, for γ the game system of Section 15.2 with $\mathcal{F}'$ in place of $\mathcal{F}_{\mathsf{Sig}}$. Then $\{\mathcal{F}'\}$ UC-emulates $\{\mathcal{F}_{\mathsf{Sig}}\}$ within* 0*: for every occupant $\mathcal{A}$ there is a simulator $\mathcal{S}_{\mathcal{A}}$ with*

$$\text{EXEC}\big(\{\mathcal{F}'\}, \mathcal{Z}, \mathcal{A}, \mathcal{C}\big) \;\equiv\; \text{EXEC}\big(\{\mathcal{F}_{\mathsf{Sig}}\}, \mathcal{Z}, \mathcal{S}_{\mathcal{A}}, \mathcal{C}\big)$$

for every $\mathcal{Z} \in \mathbb{Z}_{\{\mathcal{F}'\},\{\mathcal{F}_{\mathsf{Sig}}\}}$.

Proof. The simulator is the obvious one — $\mathcal{A}$ bundled with an internal copy of $\mathcal{F}'$, answering each of $\mathcal{F}_{\mathsf{Sig}}$'s three slot queries by putting the same call to that copy — so the whole of the argument is that the two sides never disagree on a returned value. Since $\mathcal{F}_{\mathsf{Sig}}$'s answer *is* the slot's answer unless its own code overrides, that reduces to showing

no override ever fires, and there are exactly four: GEN's memoization, GEN's sanitizer, SIGN's sanitizer, and VERIFY's honest-key test. Each is taken in turn and excluded by one of the four hypothesised properties at 0, the sanitizer cases splitting further by which repair the sanitizer would make. That is where all four properties are spent, and it is why the theorem needs exactly them. The simulator $\mathcal{S}_{\mathcal{A}}$ bundles $\mathcal{A}$ with an internal copy of $\mathcal{F}'$ and runs it on the calls $\mathcal{F}_{\mathsf{Sig}}$ reports. The functionality $\mathcal{F}_{\mathsf{Sig}}$ asks the slot for exactly the three values it does not pin — the key at GEN, the signature at SIGN, the bit at VERIFY — and $\mathcal{S}_{\mathcal{A}}$ answers each by putting the same call to its internal $\mathcal{F}'$ at the same identity and forwarding what comes back, letting the hosted $\mathcal{A}$ see, and answer, whatever $\mathcal{F}''$s own code would put to a slot. Leakage and corruption instructions it serves from the internal copy likewise; note that this is why $\mathcal{F}''$s LEAK need not resemble $\mathcal{F}_{\mathsf{Sig}}$'s at all, the environment reaching it only through the slot, which $\mathcal{S}_{\mathcal{A}}$ occupies.

Couple the two executions on equal tapes as in Lemma 11.2. Everything reduces to one claim: at every call, $\mathcal{F}_{\mathsf{Sig}}$ returns what the internal $\mathcal{F}'$ returned, so that the two sides agree on the value the environment sees. Since $\mathcal{F}_{\mathsf{Sig}}$'s answer is the slot's answer *unless* its own code overrides, it is enough that no override ever fires, and there are exactly four.

GEN's memoization and GEN's sanitizer. If $\mathtt{VK}[\mathtt{id}.\mathtt{P}]$ is already set $\mathcal{F}_{\mathsf{Sig}}$ returns it; FINONE at 0 says the internal $\mathcal{F}'$ returns the same key at that party too, so the two agree. Otherwise $\textsc{San}[\textsc{Clean}_{vk}]$ replaces the offered vk if it is not a key, or duplicates one already issued, or has a signature already recorded valid under it. Not a key: then $\mathcal{F}'$ answered GEN with a non-key, and a later SIGN at that party would carry $\sigma \in \Sigma$ with $(\text{VER}, \cdot, vk, \cdot, \cdot, 0)$ available to the environment — excluded by FINCOR at 0 once the environment signs and verifies, which it may. Duplicate: two honest parties then share a key, and signing at one and verifying at the other wins FINUF, excluded. Presigned: some $\mathtt{Ver}[vk, msg, \sigma] = 1$ stands before vk is issued, and $\mathcal{F}_{\mathsf{Sig}}$ records a 1 only at SIGN, so an honest party signed under a key not yet generated — which $\mathcal{F}_{\mathsf{Sig}}$'s SIGN refuses, returning $\perp$, and which the hypothesis on $\mathcal{F}''$s SIGN refuses too. So the sanitizer is idle.

SIGN's sanitizer. The sanitizer $\textsc{San}[\textsc{Clean}_{\sigma}]$ replaces σ if it is not a signature or if $\mathtt{Ver}[vk, msg, \sigma] = 0$ already. Not a signature: FINCOR's $\sigma \in \Sigma$ conjunct is exactly the reading that lets this case be excluded, the environment verifying the returned value and FINCOR forbidding a 0.

Already 0: then the same triple verified 0 earlier and verifies 1 now — the internal $\mathcal{F}'$ having produced a signature it must accept — which is FINCONS, excluded at 0. So it too is idle.

VERIFY's memoization and its honest-key suppression. The first returns the recorded verdict where one exists, and FINCONS at 0 says the internal $\mathcal{F}'$ answers the same triple the same way, so they agree; this is the override the counterexample of Proposition 21.2 exploited, and it is exactly what FINCONS closes. The second sets $b \leftarrow 0$ on a fresh triple under an honest party's key, and FINUF at 0 says the internal $\mathcal{F}'$ answers 0 there too — a 1 being its win condition. So both agree.

No override fires, every value the environment sees is the internal $\mathcal{F}'$'s own, and the coupled runs proceed in step to the same halt, exactly as in Lemma 11.2's no-flag case. The distributions are equal and the advantage is 0. □

So the converse holds, and what it took to hold is the point. Two properties were not a characterization; four are, and the two added are not deep facts about signing but the two places $\mathcal{F}_{\mathsf{Sig}}$'s code *commits* — a table written once and read thereafter, twice over. A functionality's properties characterize it exactly when they cover every such commitment, and reading a specification for its commitments is a better way to find the missing property than asking what else one would like to be true.

21.4 A property of π_{Sig} that $\mathcal{F}_{\mathsf{Sig}}$ lacks

The last question is whether $\mathcal{F}_{\mathsf{Sig}}$ is the strongest object of its shape. It is not, and the gap is not an oversight: it is what sanitization costs.

A further finalization: collision resistance

id.FINCOLL() from id′ (collision resistance)
1: **require** ¬DONE;
2: **require** id.P = Z; DONE ← TRUE
3: C ← (C, Z).FULLSTATUS() as id
4: $w \leftarrow 1$ if $\exists\, \mathrm{P} \notin \mathbf{C},\ \mathrm{Q}, \mathrm{Q}' \notin \mathbf{C},\ vk,\ msg \neq msg',\ \sigma$:
5: $(\text{GEN}, \mathrm{P}, vk) \in \mathtt{tr} \wedge (\text{VER}, \mathrm{Q}, vk, msg, \sigma, 1) \in \mathtt{tr}$
6: $\wedge\ (\text{VER}, \mathrm{Q}', vk, msg', \sigma, 1) \in \mathtt{tr}$; else $w \leftarrow 0$
7: **return** w

It takes the search reading, winning being meant to be hard rather than no better than a guess.

Proposition 21.4 (π_{Sig} has it, $\mathcal{F}_{\mathsf{Sig}}$ does not)**.** π_{Sig} *has* FINCOLL *within* $q \cdot \varepsilon_{\mathrm{COL}}(\mathcal{B}_{\mathrm{COL}})$ *for an adversary* $\mathcal{B}_{\mathrm{COL}}$ *against* DS *built as in Proposition 21.1. The functionality* $\mathcal{F}_{\mathsf{Sig}}$ *does not have it within any* $\varepsilon < 1$*: there are* $\mathcal{Z}^*$ *and an occupant* $\mathcal{X}^*$ *with* $\Pr[\mathrm{WIN}_{\mathrm{FINCOLL}}] = 1$.

Proof. For π_{Sig}: a winning trace exhibits one σ verifying under an honest party's vk on two distinct messages, and verification is DS.VER by line 10, so the pair is a collision for DS under a key drawn on uniform coins. The adversary $\mathcal{B}_{\mathrm{COL}}$ plants its challenge key at a guessed one of the at most q generating parties, answers that party's signing from its oracle and runs the rest itself, and outputs the two messages and the signature.

For $\mathcal{F}_{\mathsf{Sig}}$: let $\mathcal{X}^*$ answer every GEN query with a fresh key, every VERIFY query with 0, and both SIGN queries with one fixed $\sigma^* \in \Sigma$. Let $\mathcal{Z}^*$ corrupt nobody, call GEN() then SIGN(msg) and SIGN(msg') for $msg \neq msg'$ at an honest party, then VERIFY on both triples, then finalize. The first signature is accepted, $\textsc{Clean}_\sigma$ asking only that σ^* be a signature and that $\mathtt{Ver}[vk, msg, \sigma^*] \neq 0$, which holds, it being unset; $\mathtt{Ver}[vk, msg, \sigma^*] \leftarrow 1$. The second is accepted for the same reason at the *other* message, $\mathtt{Ver}[vk, msg', \sigma^*]$ being unset as well, and $\mathtt{Ver}[vk, msg', \sigma^*] \leftarrow 1$. Both verifications then return the recorded 1, and the verdict at line 4 gives 1 with certainty. □

The attack is not a defect to be patched — or rather, it can be patched, by adding to $\textsc{Clean}_\sigma$ the conjunct that σ be recorded valid for no other message under vk, and the interest is in what that would cost. Sanitization exists so that a functionality's guarantees do not depend on the slot answering helpfully (Chapter 12); each conjunct of a CLEAN predicate is a guarantee bought, and each is also a value the simulator may no longer choose. The functionality $\mathcal{F}_{\mathsf{Sig}}$ as written buys correctness and unforgeability and declines to buy collision resistance, which is a defensible reading of what a signature functionality is for — and the consequence, made precise by Proposition 21.4, is that a realization may be strictly stronger than the ideal object it realizes.

That is worth sitting with, because it inverts the usual intuition. An ideal functionality is often read as the best possible behaviour, with realizations approximating it from below; here π_{Sig} satisfies a property $\mathcal{F}_{\mathsf{Sig}}$ provably fails, and no amount of emulation transfers it downward — Corollary 5.11 carries properties from the ideal object to

the realization, never the other way. What an ideal functionality fixes is what a protocol may be *relied on* for, not what it happens to achieve; anything a realization achieves beyond that is real, and is invisible to every argument that goes through the ideal object.

Pending Issues

What this paper leaves open, gathered from the places the body raises it, together with what its review does and does not cover. Presentation and typesetting are not listed.

Dummy completeness stops at responsive calls. Theorem 4.29 hypothesises systems whose cores place no responsive call. The dummy's relay is refused at the first such call and answers $\perp$ where a real adversary would answer with substance, so the regrouping fails. Recovering completeness needs a matching notion on the environment's side, and we do not give one.

Games have no responsive theory. All three property-transfer results inherit that condition from Theorem 4.29. A game system whose $\mathcal{F}$ reaches the slot through $\mathcal{A}^!$ sits outside all three and must have its dummy-only hypothesis established directly. Four of this paper's own functionalities decline the hypothesis for exactly this reason and argue their transfers by hand.

The quantifier order has no converse. Definition 4.11 fixes the simulator before the environment, and implies the weaker reading in which the simulator may depend on both. The converse we do not prove. Whether the two notions separate is open.

Blocked pairs are not characterized. Tightness keeps a blocked comparison faithful for the private members, but it constrains no member of I, and an interface member admitting a bare used name is reachable by a hosted fake instance. Pinning those, or admitting only $\mathbf{Z} \cup \mathbf{A}$ where the environment reaches them directly, would close it. We give no general sufficient condition and no characterization.

What spawning would cost. Admissibility withholds the whole name closure, which is more than a comparison strictly needs. Letting a

simulator spawn functionalities at identities φ does not occupy would leave only $\mathrm{IDS}(\varphi) \setminus \mathrm{IDS}(\pi)$ to withhold — empty for the usual pairing. What that relaxation costs in composition is not worked out.

Trace atomicity is not temporal consistency. Every trace entry is assembled atomically, and every append in the paper was checked. Whether an entry's timestamp and the set fetched alongside it agree *in time* is a different property, secured by responsiveness rather than by atomicity. The two are easy to conflate and the paper nowhere separates them in one place.

What the review covers. Six of twenty-one audit units were examined in full, chosen leaves-first by criticality, and in a single pass rather than the two independent ones the protocol asks for. Every defect found clustered on one mechanism — what a bundle places outward, and whether it may use its hosted members while answering a responsive call — and all were fixed. The exposure is therefore in what one pass did not look for; the realization of the authenticated channel is where a further pass should go, being the unit that actually contained defects.

Bibliography

[1] J. H. An, Y. Dodis, and T. Rabin. On the security of joint signature and encryption. In *Advances in Cryptology — EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 83–107. Springer, 2002. https://eprint.iacr.org/2002/046

[2] M. Bellare. Practice-oriented provable-security. In *Proceedings of the First International Workshop on Information Security (ISW '97)*, volume 1396 of *LNCS*, pages 221–231. Springer, 1998. https://cseweb.ucsd.edu/~mihir/papers/isw-pops.pdf

[3] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In *Advances in Cryptology — CRYPTO 2017*, volume 10401 of *LNCS*, pages 324–356. Springer, 2017. https://eprint.iacr.org/2017/149

[4] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015. https://arxiv.org/abs/1203.1539

[5] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology — EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, 2006. https://eprint.iacr.org/2004/331

[6] R. Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop (CSFW 2004)*, pages 219–233. IEEE, 2004. https://eprint.iacr.org/2003/239

[7] R. Canetti. Universally composable security. *Journal of the ACM*, 67(5):28:1–28:94, 2020. https://eprint.iacr.org/2000/067

[8] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *Theory of Cryptography — TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, 2007. https://eprint.iacr.org/2006/432

[9] P. Farshim, M. Karvonen, A. Knispel, M. Kohlweiss, and P. Wadler. UC, categorically: Rigorous diagrammatic proofs. *arXiv preprint arXiv:2608.04521*, 2026. https://arxiv.org/abs/2608.04521

[10] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988. https://people.csail.mit.edu/silvio/Selected%20Scientific%20Papers/Digital%20Signatures/A_Digital_Signature_Scheme_Secure_Against_Adaptive_Chosen-Message_Attack.pdf

[11] J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In *Theory of Cryptography — TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, 2013. https://eprint.iacr.org/2011/310

[12] G. Plotkin and M. Pretnar. Handlers of algebraic effects. In *18th European Symposium on Programming (ESOP 2009)*, volume 5502 of *LNCS*, pages 80–94. Springer, 2009. https://homepages.inf.ed.ac.uk/gdp/publications/Effect_Handlers.pdf

[13] V. Shoup. Sequences of games: A tool for taming complexity in security proofs. *IACR Cryptology ePrint Archive*, Report 2004/332, 2004. https://eprint.iacr.org/2004/332