The Complete, Authorised and Definitive FinTracer Compendium (Volume 1, Part 1, 3rd Revision [a.k.a. "Purgles" edition]): A work in progress (interim, incomplete draft) [Wash before wear]

Michael Brand

November 23, 2023

Contents

Chapter 1

Introduction

1.1 Aims of this document

This document originally only aimed to provide a consistent, unified, up-to-date view on our thinking regarding the FinTracer solution, which was previously introduced piecemeal in multiple algorithm documents.

However, because FinTracer is presently envisaged as quite a large and complex system, with multiple users of quite different roles, assumed backgrounds and visibility into the workings of the system, the document naturally morphed into a system description document, and it is now the project's presently-most-complete representation of the functional—and, increasingly also the non-functional—aspects of what we mean by Purgles Platform and FinTracer system. Thus, the document now covers also all of the additional functionality, which is perhaps non-algorithmic in nature and has nothing to do with privacy preservation, which is nevertheless needed to support the core design goals of the project.

One particularly visible and surprising part of this is the FTIL language. This language was developed as our solution to support the Swiss-army-knife qualities that are required of this system for its successful operational use. We expect that each intel investigation, for example, will follow its own, unique evolution, will create its own, unique typologies, and these will result in their own, unique artefacts, later to be used in unique ways. The only method to support this was to build the FinTracer system around a complete language for distributed, privacy-preserving computation, and as none exists we invented FTIL for this purpose. Much of this document, consequently, is a functional specification of the FTIL language.

Though this document is meant to cater for many audiences, the majority of it as of its previous edition was (in addition to the system's top-level architecture) this specification of the FTIL language. This is a section that was meant for two distinct audience types. The first type is those wishing to learn how to program in FTIL, and how to define typologies using it; the second type is those who want to program the system itself and wish to implement the language according to its specification here. This effort is presently underway, and many of the changes made in this document's latest version are based on feedback from the Alerting project's engineering team, for which I am most grateful.

The document is divided into multiple parts, each part breaking down one aspect of the system. This was done with the aim that different users can easily focus on only those aspects relevant to them.

This present, introductory chapter maps out the scope of the system and introduces the structure of this document.

1.2 What is the FinTracer system?

The **Purgles Platform**, to be created by the **Alerting Project**, is a distributed computation environment composed of computation nodes both at AUSTRAC and at participating Reporting Entities (REs), as well as interfaces to these nodes and other required peripherals, whose overall purpose is to enable privacy-preserving computation over AUSTRAC and RE-held data for the goal of improving AUSTRAC's ability to fight financial crime, as stipulated in the AML/CTF act. Future versions of the platform may expand its capabilities by also including computation nodes at some of AUSTRAC's Partner Agencies (PAs).

It is at this time unclear precisely what type of usage will be made of the system. At its core, the purpose of the Purgles Platform is to aid intel investigations into financial activity where this activity is best detected through observation of domestic transactions.

The **FinTracer system** is a software system for running distributed computation inside the Purgles Platform environment. At its core, it is an engine executing the FinTracer Intermediate Language (FTIL), which is a language specifically developed to facilitate the definition of generic, privacy-preserving computation of the kind required for the Purgles Platform.

At present, the FinTracer system is the only software system planned to be run on the Purgles Platform. The Platform is therefore, at the moment, planned to include only the various tools required to properly interface with, maintain and update the FinTracer system, execute its commands and audit its usage. This may change if at any future time it is decided to include additional subsystems on the Purgles Platform.

Here are a few examples of such potential additional subsystems:

1. Given that the FinTracer system only deals with privacy-preserving querying, it may be necessary for the Purgles Platform to host a second system for general querying, that will serve purely for query automation. Such a system may run general queries, without privacy-preservation constraints, over the same data that the FinTracer system runs on, and by this provide a solution for follow-up querying, after the FinTracer system identifies specific suspects.

2. The Purgles Platform may even house multiple instances of the FinTracer system in order to allow some level of multitasking (presently explicitly excluded from the FinTracer system's design), as this would be, for example, required if the Purgles Platform will need to support periodically-run queries, such as those used in transaction monitoring.

For the purpose of the present document, however, we will use the term "FinTracer system" as largely synonymous with "Purgles Platform", noting that the differences between these, as they are at the current time, are largely ones that are beyond the scope of the present document. For example, we will refer to the computation nodes at the REs as "Peer Nodes" and the computation node at AUSTRAC as "Coordinator Node" even though these names reflect only their usage inside the FinTracer system, and not necessarily their roles or relationships within the wider Purgles Platform.

The **FinTracer algorithm suite** is a set of algorithms/protocols that was designed to deliver the operational goals of the system and will be implemented in FTIL in order to be run in the execution environment of the FinTracer system. For some of these algorithms, FTIL code is provided already in this document. At the present time, the FinTracer algorithm suite includes the basic FinTracer algorithm, some housekeeping algorithms that enhance FinTracer's outputs or allow it to be used as a building-block inside a larger typology definition [1, 2, 3], and multiple algorithms for which it has not yet been decided whether they will be part of the initial release of the FinTracer system, including FinTracer DARK variants [7], tools for working with the system that shield the REs from potentially sensitive information [4, 5, 8], tools for working with PAs [9] and additional peripheral services [6].

The FinTracer algorithm suite represents the basic building blocks that we will want to run on the FinTracer system. The FinTracer system, on the other hand, was designed to support the execution of both these basic building blocks and all their future extensions, while maintaining

the FTIL security and privacy requirements. Historically, FTIL was designed when only the first few of the algorithms in the FinTracer algorithm suite had already been invented, so the fact that the present set of algorithms can be implemented inside the same framework is a testament to its versatility and general-applicability.

1.3 Goals of the FinTracer system

The purpose of the FinTracer system is to allow an AUSTRAC user to query the financial system in ways that cannot currently be done by any single entity, be it an RE or AUSTRAC itself. The use of privacy-preserving technology is integral to this vision, in that it is imperative to preserve the privacy of non-suspicious, non-target accounts in order for this system to enjoy both legality and social license. REs do not receive in FinTracer any information from other REs, and AUSTRAC does not receive any (or substantially any) information beyond what it explicitly asks about.

This statement of the purpose of the FinTracer system portrays the AUSTRAC user as the main user of the system. For this user, the experience of using FinTracer is not far removed from the experience of using a graph database to discover and quantify financial crime. The user may, as a prototypical example, wish to discover instances of a particular type of criminal activity. This activity, typically spanning multiple actors (some criminals, some innocent catalysts and some victims), is then described in terms of the traces that it leaves in the financial system: the actors may be associated with one or more accounts, each account describable through some form of a characterisation, and the relationships between these actor-accounts can similarly be characterised. In some cases these relationships amount to direct fiscal interaction (direct money transfers). Elsewhere, this interaction may be indirect, involving one or more interim accounts. Here, the AUSTRAC user may be able to characterise the indirect interactions by means of the properties of the direct fiscal links composing said interactions, the nature of the intermediate accounts through which these indirect interactions flow, and the degree of indirectness (i.e., the length of the transaction paths connecting said actors).

This entire description, characterising the financial footprint across potentially multiple accounts at potentially multiple REs of a particular crime profile, we refer to as a **crime typology**, or just as a **typology** for short. The purpose of FinTracer is to allow such queries to be formulated and asked, in a way that does not compromise the privacy of any account that does not match the typologies sought.

1.4 Interfacing with the FinTracer system

As discussed, the AUSTRAC user, wishing to match crime typologies (i.e., wishing to find specific instances of financial behaviour consistent with a certain crime typology), may interact with the Purgles Platform in ways quite similar to what would have been the case had FinTracer been a graph database. The classical graph database-style queries are the prototypical queries that match crime typologies, and the user wishing to run these queries will connect to the Purgles Platform in ways comparable with client-server architectures that are typical of (graph-)database interactions.

Under the hood, however, the system supporting these queries is a distributed computation system, with careful management of information visibility and information interaction, rather than a simple, classical client-server system.

What this means, practically, is that the Purgles Platform will need to supply its users, both in AUSTRAC and at the REs, with a wide suite of services, as one would expect from client-server systems. This includes user management (logins and passwords), job and job-state management (e.g., the ability to initiate, monitor and preempt a job, and the persistence of jobs irrespective of individual sessions), support of administration tasks (including the handling of privileges), auditing capabilities, etc..

For the purpose of keeping the FinTracer prototype as a minimalist system, it was decided that the prototype will support only a minimalist differentiation of user privileges (e.g., “administrator”

vs. "non-administrator"[1]), will only support one FinTracer process at a time (so, no background jobs, and no parallelisation of any kind[2]), and will have only very limited support for variable scoping (all FinTracer non-script variables are equally visible to all users of the coordinator node) and for user-session management.

In connecting with a (remote) database, it is generally expected that the system will enable both a fully-functional command-line interface and an API-based interface (comparable with ODBC/JDBC) that will also be fully-functional, allowing for the full range of user actions, including the definition of queries, the running of queries and the reading of results, to be performed using alternative interfaces, or, indeed, automatically using no user interface at all.

In the case of FinTracer specifically, because of the high-latency nature of some queries, it is also important that the system supports notifications (e.g., may be able to e-mail to a user that they now have new action items or that a query result has been received and is ready for further downstream processing). How such notifications are configured and handled, and through which technologies do such alerts get delivered, is not part of the requirements for the FinTracer system. In Section 6.2 we discuss the expected FinTracer system architecture that may support such alerting. Chiefly, the idea is for alerts to generate push APIs to downstream logging and monitoring systems, and for these to be configured (separately, on each node) to alert their users whenever necessary and in whichever way is desired.

In addition to the main, programmatic interface to the system that was discussed so far, there will also be a need, on each node, to occasionally manually upload data. Oftentimes, such data may be in the form of large, structured lists. This being the case, in addition to the standard command-API interface to the system, some form of file transfer will also be required.

All interface discussions in this document are on a conceptual level and should be taken as such. They are independent of any past or future technological choices about specific interface types and specific interface technologies to be used for the system. Having said this, we do believe the system will be run as a "remote" service (possibly in a cloud environment), so all interfaces to the system can safely be assumed to be remote interfaces.

Note that while interfacing with the system is comparable, functionally, to what one would expect in a standard client-server architecture, the fact that this system is actually a distributed computation system, rather than a client-server system, may in many cases complicate the actual implementation of the features mentioned above, and any others necessary for the ongoing operation, maintenance and updating of the Purgles Platform. For example, because of the privacy requirements involved, there cannot be any single party that has full visibility of all parts of the system, even for the purposes of administration, maintenance and troubleshooting.

All services required for such end-to-end management and operation of the system are considered here as part of the Purgles Platform, even though by design they are decoupled from the core functional aspects of the platform. The idea here is that organisations that need to administer a node will have leeway to choose how their node is administered, and these choices will not impact on the core functionality of the FinTracer system, or on corresponding choices made at other nodes. In **Chapter 7: Additional services**, we enumerate which such services are required at each node. The scope of this document is to describe those elements that are required for the FinTracer system as a whole to function. Mostly, these will be elements that need to be provided in a uniform way across all nodes of the system, but the requirements of Chapter 7 are the exception, in that they can be decided for each node individually, and may be provided by technically-distinct means at each node.

[1] This is a slight oversimplification. In **Chapter 3: The FinTracer Intermediate Language**, we discuss the privilege model at greater depth.

[2] This decision is not just because of the system's minimalist nature. It is also part of the system's security design in that it simplifies the analysis of possible execution pathways.

1.5 Structure of the FinTracer system

The FinTracer system is a distributed computation system. Some of its parts will run behind RE firewalls on dedicated computation nodes (which may be, in practice, one or more computers) that are called **peer nodes** and some of its parts will run behind the AUSTRAC firewall on a dedicated computation node (which may be, in practice, one or more computers) that is called a **coordinator node**. These names reflect the conceptual roles of the different types of nodes within the FinTracer system, and specifically in the FinTracer Intermediate Language (FTIL).

Peer nodes and the coordinator node will be referred to jointly as **FinTracer nodes**. FinTracer nodes jointly execute different parts of the same distributed computation protocols. Each serves for both storage and compute.

A minimalist definition of the core FinTracer system includes anything that must run on the peer nodes, anything that must run on the coordinator node, and any interfaces and services to these nodes that the system must provide.

Our vision for the long-term-future FinTracer system extends well beyond this minimalist definition, and includes many elements that can be provided for additional functionality and convenience. Intentionally, the core FinTracer system has been designed so that it can be decoupled from any such extending services, so that these can be added at later dates, without this requiring any tweaking of the software that is run on the FinTracer nodes, and most particularly on the peer nodes.

We do not anticipate that within the presently-planned lifespan of the Alerting Project any such extra services will be delivered beyond what is required for a Minimum Viable Product (MVP). Assuming that the project is a success and spawns follow-up projects, such extra services may be within scope for these follow-up projects.

As examples of such extra services not within the scope of the present Alerting Project, consider the following.

1. If we return to the comparison of the FinTracer system to a (graph) database, databases typically provide their users with a declarative query language, which users use to describe what they are seeking to match. Queries in this query language are then compiled by the query compiler in order to make them into procedural descriptions of how to search for matches, and these are optimised by a query optimiser. (The competition among database vendors is often on the performance of the query optimiser, which is why a substantial chunk of a database vendor's development budget is on the query optimiser.) In FinTracer, the writing of a declarative query, describing the typology to be matched (in what we will presumptively call **FTQL**, the **FinTracer Query Language**), is something that can be done offline in a standalone computer. This query can presumably be compiled offline using a **FinTracer Query Compiler** and to some degree also optimised offline using a **FinTracer Query Optimiser**, all of which are not absolute musts for a first version of FinTracer, and all of which (with the exception of some advanced optimiser functionality) can be implemented outside of the FinTracer nodes and be entirely decoupled from the core FinTracer functionality. We assume that these are all out-of-scope for the Alerting Project. The FinTracer system that will be delivered by the Alerting Project will only work with procedural descriptions of how to search for particular typologies that will be provided in a **FinTracer Intermediate Language** (**FTIL**), which this document will describe.

2. While FTIL includes relatively generic coordinator-node-side flow control, so that, e.g., parameter scanning, hyper-parameter tuning, and even such advanced features as cross-validation can all be encapsulated in FTIL code, sophisticated algorithms requiring sophisticated flow control (such as, for example, machine learning) would be more naturally implemented as processes external to the FinTracer system, which would interface with the FinTracer system to automatically run queries and analyse results. The FinTracer system will include the necessary hooks in order to allow such external software to interface with it and perform such actions, but the external software itself is presently out-of-scope.

Notably, one type of external software that may or may not be required for a Minimum Viable Product, and which may, accordingly, be required on an early release of the system, is support for transaction monitoring. To support daily transaction monitoring, some queries will need to be instantiated automatically on a periodic basis. FTIL does not provide tools to perform this. An outside service, automatically pushing queries to the FTIL command-line and automatically reading the results, will be required, and it will need to use the hooks described above.

The general idea is that any such external code, such as code for machine learning or for automatic, periodic querying, can be written in any language of the developer's choice. Any such external tool will be able to use FTIL in the form of API calls, and in this way interface with the FinTracer system. By design, the system will continue to function efficiently even in scenarios where such API calls incur substantial overheads.

1.6 Arrangement of this document

Following this introductory chapter, the rest of this document is arranged as follows.

We begin with **Part I: FinTracer functionality**, where we lay out the functionality provided by FinTracer, which delivers the purpose of the system. This part is arranged as follows.

We begin, in **Chapter 2: The FinTracer executor**, by discussing the FinTracer execution environment.

In **Chapter 3: The FinTracer Intermediate Language**, we describe FinTracer's capabilities and its interface to the AUSTRAC user by laying out the FinTracer Intermediate Language (FTIL), which is the tool by which the AUSTRAC user interacts with the system, and through which they can deploy FinTracer's functionality.

In **Chapter 4: Example algorithms**, we provide examples of the usage of FTIL in order to describe the various known FinTracer algorithms, starting with the basic FinTracer, through FinTracer DARK and oblivious querying, and into advanced uses, relevant for matching various types of typologies of real-world interest. These examples demonstrate FTIL's versatility in action, and implement the building blocks that we will want the finished FinTracer system to support on day one.

In **Chapter 5: Cryptography**, we describe the underlying cryptographic primitives that underpin the functionality of the executor.

Following this, we then complete our overview of the FinTracer system's requirements with **Part II: System requirements**, where we describe additional requirements that serve to supply and support the capabilities described in Part I. This part is arranged as follows.

In **Chapter 6: Auxiliary interfaces**, we describe additional, non-FTIL interfaces to the FinTracer system, such as those provided to the RE-side operator and those required in order to administer the system.

In **Chapter 7: Additional services**, we describe additional services that will be required to support the functioning of the system, such as authentication and logging services.

In **Chapter 8: Nonfunctional requirements**, we round off the system requirements by turning to the system's nonfunctional requirements.

Finally, in **Part III: Additional discussion**, we present supporting discussion, such as regarding projected system costs and requirements regarding its proper use.

In **Chapter 9: Infrastructure specifications**, we enumerate the costs of using the FinTracer system, including computational costs, communication costs, storage costs and entropy requirements.

In **Chapter 10: Cryptanalysis**, we perform cryptanalysis of the FinTracer system, both in terms of the guarantees provided by the system and where the limitations of these guarantees are.

In **Chapter 11: FTIL practicalities**, we discuss FTIL's design principles, how it meets its goals, and what users can and should do to reap the most of it.

The document concludes with **Appendix A: Changelog**, where a change log from previous versions is included, for the convenience of long-time readers.

1.7 Priorities

As this document can be understood as a requirements document, and as these requirements relate to the Minimum Viable Product (MVP) deliverable of the Alerting Project in its first release, we have at (rare) points in this document written about some of what is documented that it is "Priority 2" or "Priority 3". Such statements can be interpreted as: we do not see these requirements as a must for the MVP, but they are certainly desirable for the FTIL language at large, and should probably be supported at some future point.

Part I

FinTracer functionality

Chapter 2

The FinTracer executor

2.1 Introduction

Unlike other languages, FTIL knows exactly what its execution environment is, and there is a symbiotic relationship between language and environment. For this reason, any discussion of FTIL must begin by describing its operating environment.

Notably, the FTIL user's view of the system is a logical view. The variables and their manipulation methods are all viewed at a semantic level, corresponding to "what the user is trying to achieve". As a result, our description here of the FTIL environment will be a highly functional, logical description of it. In actuality, multiple layers of lower-level actions may be required to support these semantic-level activities, but these are beyond the scope of this document.

2.2 Execution nodes

FinTracer runs on several computation nodes referred to as *FinTracer nodes*. Of these, one is defined as a *coordinator node* and is the main access point to the system, and the others are *peer nodes*. In our implementation, the coordinator node will sit at AUSTRAC, behind the AUSTRAC firewall, whereas the peer nodes will sit at the participating REs, each behind its RE's respective firewall.

Each node is named and the name and role of each node (i.e., whether it is a coordinator node or a peer node) is known to all participants.

This is a logical view of the system. In practice, each node may be implemented as one or more computers.

2.3 Quick intro to FTIL

For the purpose of describing the executor, the only important properties of FTIL are the following.

FTIL is an interpreted language. It is interpreted and executed one command at a time, and each command manipulates the underlying state of the system, which is stored as a set of value-holding *entities*. Each entity is referred to by one or more *variables*. Entities employ reference counting, so if no variables remain that refer to an entity, the entity deallocates itself. There is also a garbage collection mechanism, to remove entities that only remain due to circular references.

The FTIL language describes all the ways in which it is possible to manipulate entities and variables. This will be detailed in Chapter 3. In general, the language is built to avoid unnecessary data duplication. Some operations do generate new entities, but many change an entity in-place. In some cases, FTIL uses copy-on-write, meaning that if only a single variable refers to an entity, it can be modified in-place, but if multiple variables refer to the same entity, it is first copied out, and only then modified, so that the modification only affects the modified variable.

Each variable in FTIL is associated with a *scope* (most—explicitly so, but some—implicitly). This scope signifies, essentially, which nodes[1] are aware of the variable's existence and hold at least some of the underlying entity's data.

Code in FTIL may be typed in at the command line one command at a time. However, FTIL users may also define *scripts*, which are FTIL's version of functions. They allow users to specify entire canned sequences of parameterised instructions to be executed. Some scripts, known as *module scripts* or *signed scripts* are loaded into the system as part of a *module*. Modules, like FTIL variables, can be defined on some nodes but not on others, so have an effective scope. Thus, some code (command-line code and unsigned/non-module scripts) is not intrinsically associated with a scope definition, and other code (module scripts) is.

Scripts in FTIL can call other scripts, and an *execution stack* is maintained for this purpose.

When an FTIL command is executed, whether it is a script or a built-in command, it is executed on a specific subset of the FinTracer nodes. This subset may or may not include the coordinator node. We refer to the set of nodes that run a command as the command's *execution scope*, or just its "scope", for short.

2.4 Compute Managers and Segment Managers

The two basic functionalities supported by the executor are

1. to determine what command needs to be executed next and by whom. We refer to this function as *coordination*. The software module in charge of this part of the execution is the *Compute Manager*. The system has one Compute Manager, and it resides in the coordinator node; and

2. to execute the desired command. We refer to this function as *execution*. The software module in charge of this part of the execution is the *Segment Manager*. There is a Segment Manager in each FinTracer node (i.e., both in peer nodes and at the coordinator node).

Because each FinTracer node has exactly one Segment Manager, we will often refer to the nodes and to the Segment Managers interchangeably. Similarly, a node's name will be treated as synonymous with its Segment Manager's ID.

The Segment Manager in the coordinator node is a fully-functional Segment Manager. As we shall see, when the Compute Manager communicates its commands to Segment Managers, it treats the Segment Manager in the coordinator node itself exactly as it does any other Segment Manager.

All FinTracer nodes have a dual role. The coordinator node has a dual role in that it contains one Segment Manager and one Compute Manager. The peer nodes have a dual role in that each contains one Segment Manager and one *Shadow Compute Manager*. The role of the Shadow Compute Manager is explained in Section 2.11, but in general it is a safety mechanism wherein each peer node is able to ascertain that the Compute Manager at the coordinator node is following the agreed protocols.

2.4.1 Segment Manager subsystems

On a logical level, each Segment Manager, whether on a peer node or on the coordinator node, houses two distinct subsystems. There is the execution subsystem and the storage subsystem. The execution subsystem itself has no (non-ephemeral) memory of its own. It relies entirely on the storage subsystem to store any value. Any value stored is stored persistently. Stored values may be accessed, manipulated and even deleted by the execution subsystem.

The execution subsystem does use temporary storage, however, as described in Section 2.5.1.

More specifically, the execution subsystem manipulates FTIL entities and FTIL variables, both of which reside in a part of the storage subsystem we call the *Variable Store*. However, there are

[1] In terminology to be introduced shortly: specifically which Segment Managers.

other parts of the storage subsystem, and these are not manipulated by FTIL. Some are read-only to FTIL and to the execution subsystem, and some are not directly visible from within FTIL at all.

In total, storage includes the following.

Variable Store: A storage location that manages all information about the FTIL variables and the values they hold. Entities (value objects) in FTIL retain, beyond their core magnitudes (which may be stored entirely or partially on any given Segment Manager) also their metadata. They may also include auxiliary information relevant to system maintenance such as the identity of their creator, date of creation, and last modification date, but the details of such auxiliary data are beyond the scope of this document. FTIL variables (which is FTIL's name for references to entities) retain which entity they refer to, as well as additional metadata properties. (See Section 3.8.3.) Multiple variables may refer to a single entity. The Variable Store also stores *identifiers*, which are names for variables. Each identifier is associated with a particular execution stack position: the identifier is "local" to the executed script at that stack position. Information in the Variable Store only gets updated by the FTIL executor, working through the execution subsystem of the Segment Manager (except, presumably, when it is manipulated by an administrator user). It reflects the Segment Manager's view of the state of the system. There is no functional requirement that this Variable Store be implemented as a database. However, it does need to support large, complex updates as atomic operations, and in some designs implementing the FinTracer system (See Section 2.5.1) may rely also on other transactionability properties. For this reason, a database implementation for the Variable Store is recommended.

Transaction Store: A queryable location, read-only to FTIL, that includes all domestic transaction information that is visible to both participants of a transaction. Each RE stores in their Transaction Store all such information visible to them, including for domestic transactions beyond those where both participating parties are owners of FinTracer nodes. The Transaction Store is SQL-queryable or equivalent, for which reason here, too, a database implementation is recommended. In the initial version of the Alerting project, the Transaction Store is expected to be populated by a "push" mechanism. However, FTIL was designed to also support a "pull" mechanism, if such was available. In general, a "pull' mechanism would indicate that, despite the Transaction Store being read-only to FTIL, it is the FTIL query that populates the Transaction Store. Note that while we do not require any specific storage format for the Transaction Store, it is a functional requirement that no SQL query looking at transactions that are all between party A and party B will be able to tell whether it is executing on the Transaction Store of A or of B. We recommend for this a to/from save format, in order to avoid any reference to "party" and "counterparty".

Auxiliary DB: Any other (SQL) queryable data. This is likely to include account and account owner information, as well as information regarding any international transfer and cash transaction to or from the account, and ideally also any other type of fund movement to or from the account, as well as what Suspicious Matter Reports (SMRs) were filed regarding any account. The Auxiliary DB must also include the information available in the Transaction Store, so that transaction information can be queried together with any other available information. The Auxiliary DB also includes *delegate certificate* and *script certificate* information (See Section 6.3), and on the coordinator node may include module files as well. These will have special tables that house them. Additionally, the Auxiliary DB may include other bits of information required for the smooth running of the system, such as tables with system status flags, indicating when the system's various data stores are ready to be queried, and are not in incoherent or volatile states. Like the Transaction Store, the Auxiliary DB is read only from FTIL. Like the Transaction Store, the fact that it should be SQL queryable makes a database implementation preferable. Unlike the Transaction Store, the Auxiliary DB is a heterogeneous mix of many information types, for which reason we expect it to be implemented as an entire database schema. By contrast, the Transaction Store only requires

a single table, and we expect Auxiliary DB access and Transaction Store access to utilise two views on the same physical table, avoiding unnecessary data duplication. Having said this, the functional requirement is not that data duplication be avoided, but rather that the two views be identical at all times.

User Input: A landing place for manual user input (which may or may not be bulk input). When the system actually reads the input, this data is erased from the landing place, transformed as necessary to system-internal storage types, assigned to an FTIL variable and copied to the Variable Store. This data will likely, but not necessarily, be in a self-describing data format, like JSON or XML. User input to the system is named, and it should be possible to house multiple inputs with different names simultaneously in the User Input area. An example implementation that meets these requirements is a file system or a file directory.

System Data: A storage area for any additional data needed for the overall functioning of the system, beyond what is needed for FTIL. This may include, for example, data about users. It is inaccessible to FTIL directly.

Additionally, some information generated by the system is never stored within the system at all, and is immediately forwarded on to downstream processing systems. An example of this is logging information.

2.4.2 Compute Manager subsystems

Like the Segment Manager subsystems, the Compute Manager also includes a stateless execution subsystem and a persistent memory store.

Two differences require pointing out, however.

1. The Compute Manager can also access the Segment Manager storage subsystem on its node, as it will need to access information stored there. For example, it will need access to the Variable Store in order to read scripts that are stored there. (A script is a type of variable and is stored in the Variable Store.) Also, it may need access to the system data in order to perform permission validation. We do not expect, however, that the Compute Manager will require access to any other Segment Manager storage subsystems.

2. The Compute Manager's own persistent memory store is composed largely of its Variable Registry. This is similar to the Segment Manager Variable Store in that it contains for each entity its metadata, and for each variable and identifier its properties. However, the Variable Registry is different in that the variables are stored without their "value". (See Section 3.8 for the exact delineation between value and metadata.) The Compute Manager's Variable Registry is the system's only complete record of all variables in the system (whereas Segment Manager Variable Stores only store variables that are in their scope), but it may contain many variables whose values are not accessible to any given node, including to the coordinator node.

The Variable Registry has a dual role in FTIL. First, when the Compute Manager determines whether a command passes its prerequisite checks, it uses the Variable Registry data to determine this. The Compute Manager and every Shadow Compute Manager has enough information to check the prerequisites of any command that they are asked to perform. Also, the Compute Manager has in the Variable Registry enough information in order to continue updating the Variable Registry from command to command (e.g., in determining which variables become "protected" or "immutable" as a result of running any command).

Second, some commands in FTIL are conceptually performed by the Compute Manager, in the sense that Variable Registry data is used in them. This includes decisions on flow control, scoping, and some cases of indirect addressing (See Section 3.7).

Note 2.4.1. When we say "Conceptually performed by the Compute Manager", this does not in any way reflect an implied or suggested software implementation architecture. Indeed, because in

peer nodes there is little distinction between the actions of the Shadow Compute Manager and those of the Segment Manager (There is no per-node Variable Registry, so in peer nodes both types of actions are performed on the nodes' Variable Store data, blurring their distinction), it is quite conceivable that in an actual software implementation there will be little distinction between the software pathways used on the peer nodes for Segment Manager actions and for Shadow Compute Manager actions, and, consequently, potentially these operations may also be handled through quite similar software pathways also on the coordinator node.

Usage of the phrase "Performed by the Compute Manager" in this document should therefore be understood to solely mean "Utilising Variable Registry data".

2.5 Advanced architecture issues

2.5.1 Caching and the microservices architecture

As described above, the Variable Store, storing FTIL's variables, is architecturally segregated from the subsystems that interpret the FTIL commands and that perform the actual variable manipulation. These subsystems are considered "stateless", in that they do not require any permanent storage. In terms of the architecture involved, we expect the FinTracer system to be at least in part deployed in a cloud environment. We expect the executor to run as a microservice, which is to say on a separate cloud node to the Variable Store, and we expect the Variable Store to be implemented as a database.

FTIL should not, however, be implemented in the naive way, in which the handling of every FTIL command must begin by specifying which objects it manipulates, followed by the loading of these elements from the Variable Store, their manipulation, and ultimately saving them back after every command.

On the contrary, much of the design of FTIL is informed by a need for efficient execution, and the language is geared towards in-place manipulation wherever possible.

As a result, we expect the handling of FTIL variables to be predominantly done within the compute subsystem, and preferably in memory. This is a form of caching. The system should write back to the Variable store, as frequently as practicable, any value that has changed. Such writes must, however, always reflect a coherent state of the system, as it was at the end of a specific FTIL instruction.

As such, it is better to think of the Variable Store as the system's backup than as the system's "single source of truth".

Some comments regarding this:

1. Because this is merely a caching system, there is no necessity for every FTIL variable to be simultaneously held in the compute system's memory. They can be loaded when they are needed.

2. We expect reads from and writes to the database to be atomic, implemented as database transactions. This means that if a variable is read, it is necessarily read correctly, and with a coherent value, and if it is written, it is necessarily written correctly, in full. A failure in the middle of writing, for example, cannot result in a half-written, non-coherent final value. Instead, such a write is rolled back, and the failure is reported to the executor. This indicates that the state of the Variable Store is always legal. Moreover, coherency must be maintained not just at the level of an individual variable. All variables that have been modified up to a save point must be saved together, in a single atomic operation, to ensure that the entire state of the system is coherent.

3. Even beyond the coherency level described above, while this section does not attempt to give exact answers regarding when to write back and when not, it is important that write-backs must be synchronised between all Variable Stores on all nodes. If at any point an error occurs that requires discarding the local caches and re-loading values from the local Variable Stores, the loaded values must reflect a globally-coherent state of the system.

4. Our expectation for the initial release of the FinTracer system is that it will not support complicated write-back strategies. Instead, write-backs will happen after every command. For future versions, however, we note that write-backs must occur at the very least in the following conditions: (1) Every time a top-level script terminates, (2) Before entering into an FTIL (outermost) "`modifying()`" block, (3) before and after any "`broadcast`" or "`transmit`".

The FTIL "`modifying()`" command is an exception to the rule that write-backs to the Variable Store should happen as frequently as is practicable.

The "`modifying()`" command defines an execution block within which no write-back to the Variable Store is allowed. The compute subsystem must work solely off of its own cache, if necessary loading more variables from the Variable Store into the cache. Any command failure/exit within a `modifying` block will result in the entire cache being invalidated and the state of the system being returned to its previous state.

Such "`modifying()`" blocks can be nested, but from the perspective of the FTIL executor only the outermost block has impact.

Note 2.5.1. The above description implies a specific implementation, but the idea of this section is not to constrain the solution design to this specific implementation. It is the functionality that is important: if a command is aborted while inside a "`modifying()`" block, the system must revert to its state prior to entering the block. This can be implemented in many ways. The description here merely emphasises the additional functional requirement that such execution be handled efficiently, with minimal amounts of unnecessary data copying. Whether this is done by introducing "save points", by database roll-backs, or by programming atomic update capabilities directly in the Compute Manager (to name a few examples) is not a requirement. We will, throughout this document, use all these various terms interchangeably, when discussing FTIL's data update strategies.

2.5.2 Requirements of the SQL environment

As described in Section 2.4.1, there are two SQL-queryable locations in the FinTracer system's architecture: the Transaction Store, being a table, and the Auxiliary DB, being a schema. Both locations are described as read-only.

This view is reinforced also in Section 3.11.2 and Section 3.11.3, where we describe specific FTIL instructions that execute particular SQL queries on this data and return results (in the form of FTIL variables).

In fact, FTIL is by design a quite limited environment, and, as a result, much of the power required to define complex, real-world typologies has been relegated in the system's requirements to the SQL layer. While most of this document does not deal with the SQL layer, and therefore does not mention these requirements, in this section we enumerate some of what makes for a powerful SQL environment, which are features that will be needed for the system to be practically useful.

Scratch pad: While the Auxiliary DB is read-only, and should remain so in order not to taint any issues of data ownership, SQL code actually querying the system should run on a separate schema that has read access to the Auxiliary DB but also the ability to create, read and write new tables, as well as to ultimately delete them when they are no longer necessary. Having such a "scratch pad" area will allow the system to store intermediate calculation results and not need to always compute everything directly from the raw Auxiliary DB data. We expect much data wrangling SQL code to need to be applied to the raw Auxiliary DB data before it becomes usable, and this code will be a common prerequisite to the vast majority of actual user queries.[2]

[2] How separated this scratch pad must be from the Auxiliary DB is a question of nonfunctional requirements, to be addressed in Chapter 8. This is an important question, because in a naive implementation the scratch pad may cause the system to run out of space, or may introduce unexpected loads that interfere with the system's other operations, such as in updating the Auxiliary DB.

User Defined Functions: Many data manipulations cannot be done at all, or cannot be done efficiently in FTIL, and we rely on the SQL layer to perform them.[3] Unfortunately, in many cases the functions needed are not easy to implement using plain SQL, and one requires for their efficient implementation either User Defined Functions (UDFs) or User Defined Aggregates (UDAs), which databases often allow one to write in a lower-level language like C.[4] We understand that introducing a UDF or a UDA into a database is the installation of new software, akin to a software update (but without the advantages of FTIL's permission management system), so the introduction of any such functionality will most likely require a lengthy manual approval process, meaning that new UDFs will have to be introduced only rarely. This is a known limitation, and, we believe, a manageable one, as opposed to not having UDF/UDA support at all, which, we believe, would make the system extremely limited in practice.

Transaction Store parity: In Section 2.4.1 we require that "no SQL query looking at transactions that are all between party A and party B will be able to tell whether it is executing on the Transaction Store of A or of B." We expect the SQL layer to be the one ensuring this invariance: instantiations of the FTIL function `rm` should not run the function's SQL conditional directly on the complete Transaction Store data, but rather, at minimum, should split the information by the identity of the (unordered) RE-pair involved in each transaction and run the SQL conditional provided in an environment that only sees the transactions belonging to one such $\{A, B\}$ pair, with the idea that this will make the code unable to tell whether it is running on the Transaction Store of A or of B.

Note 2.5.2. We do require SQL conditionals used in `rm` to support UDFs and UDAs, same as for Auxiliary DB querying, but, due to the technical difficulties involved, will at this time not require a two-sided viewable (i.e., Transaction Store) version of the "scratch pad" area, as well. We do believe that not having the ability to store intermediate computation results in two-sided-viewable querying imposes limits on the usability of the system, and suggest that both the architecture requirements in order to support such a read-write solution to the Transaction Store and the user requirements for having such a solution will be revisited post-MVP, at a time when the usage profile of the system is better understood.

2.6 FinTracer users

The FinTracer environment includes four core types of users: the AUSTRAC-side user, the RE-side user, the AUSTRAC-side administrator and the RE-side administrator. Administrator users deal with general system maintenance. As it pertains to FTIL, they have only two roles:

1. They manage users and passwords. This is a role that is done independently for each node. Each node has its own list of users and administrators and manages these locally. This role is identical for both peer nodes and the coordinator node.

2. They configure certain values that are not generated by the system itself. For example, they may upload the master certification key for FTIL scripts, as discussed in Section 6.3.1. As another example, they may perform the necessary reconfiguration if the list of participating REs needs to be adjusted.

All other operations are performed by regular users. FinTracer has no form of permission management other than this. All regular users in each node have exactly the same permissions.

[3] For example, FTIL does not allow computation on lists of strings, so any such manipulation will have to be done in SQL.

[4] Examples of such functions that may be useful for defining typologies are various locality sensitive hashes and `rm` variations.

The only distinction is that AUSTRAC-side users, being the regular users of the coordinator node, have a very distinct role to RE-side users, who are the regular users of peer nodes.[5]

Note 2.6.1. All certifications discussed in Section 6.3 other than the master certification are stored in special tables in the Auxiliary DB. The RE-side administrator can delegate permissions to populate these tables as they see fit. In practice, populating these tables may be done by the RE-side administrator, by the RE-side user, by yet a fifth type of user in the system, or even by an automatic process. It is a distinct role.

We expect the FinTracer system, once operational, to be used in the service of multiple investigations at once, each investigation potentially carried out by multiple users. Presently, because all these users share exactly the same permissions, it is technically possible for users of one investigation to tamper with the artefacts relating to other investigations. In Section 11.1 we discuss how to structure such collaborative investigations so as to prevent accidental stepping of toes between investigations. Such conventions cannot stop *malicious* tampering with investigation artefacts, but they do make any such activity easily detectable in system logs.

In future, we expect FTIL's permission system to be expanded so as to support "group" permissions, with each group (representing an investigation) only able to see and manipulate its own artefacts. This would be a natural extension to the language. However, no such permission system is expected for the system's initial release.

2.7 FTIL sessions

A coordinator-node-side user's interaction with FinTracer begins with them logging into a node, being authenticated, and arriving at the FTIL command line. This log-in process allows the user's actions to be identified in log files with the person who performed them. The FTIL environment is also aware of the user's identity.

If a user logs out while the system is idle, this terminates the session.

However, FTIL sessions are not tied to interactive user sessions. A user can initiate an FTIL command and then log out, letting the command run without there being anyone logged into the coordinator node. This is necessary, because FTIL scripts may run for weeks, especially if they require RE-side manual input. The rules regarding when sessions start and when they end are therefore a little different when the system is not idle.

If the system is running code when a user logs out, this does not immediately end the previous session. The code will continue running as usual, without a connected user, in the old session. The session will only terminate when the running execution finishes. It does not matter how the code terminates, whether it is through normal termination, as a result of an abort, or through a user command.

If the system is running code when a user logs in, this does not immediately start a new session, nor does it terminate the previous session, and the logged-in user does not arrive at the FTIL command line, but rather at a "system busy" indicator.[6] The new user should be able to determine which user started the currently-running session, and will also be able to preempt it, if necessary.

In short, the general rule is that session start and end happen not at log in and log out, but rather on the first occasion after them in which the system is idle. If a user logs in and then logs out, and the system was never idle at any time between these two events, this does not result in any new session.

There is no concurrency in FTIL sessions: a new session cannot start until the previous session has ended. In particular, a logged-in user can be confident that no background action and no

[5] AUSTRAC-side user local scripts present a minor exception to this rule, as they are not visible to all AUSTRAC-side users equally. See Section 2.7 for details. The main point, however, stands, that all AUSTRAC-side regular users have the same role in the system.

[6] This is a case where the system performs actions in an asynchronous manner. There are other examples, such as decisions regarding cache persistence, or the "`rm`" example discussed in Section 3.13.6. It is not the case that the FinTracer system does not have parallelisation or asynchronous execution. Rather, FTIL does not provide any, to its users.

action by any other user is changing the state of the FTIL environment during or between their commands.

While at the FTIL prompt, the user has the ability to define, manipulate, interact with, and ultimately delete variables (and by extension also entities and identifiers). The state of these is the only state maintained by the system. Values that are local to any given node are persisted in the node's Variable Store, and general "metadata" (loosely meaning "data about data and data about the current execution state") is stored also in the Variable Registry.

In both cases, variables are mostly stored in a single system repository, where they are equally accessible by all FTIL users. The single exception to this is non-module scripts (i.e., scripts defined on the FTIL command line). These are stored, on the coordinator node, on a per-user basis.

As a result, a user who has logged out and logged back in may discover that the variables they have left there may have changed in the interim because of another user that has logged in meanwhile. However, all non-module scripts will remain exactly as the user had left them.

This allows multiple users to work on the same investigations, seamlessly operating on the same set of investigation objects, while not having to worry about volatility in their personal script environment. (We discuss in Section 11.1 how to safely share code between users working on common investigations.)

Outside of the state reflected in the Variable Store, FTIL follows a stateless "command / response" sequence. It has no notion of a job or context, only an execution stack.

Note 2.7.1. It would be reasonable for users to want notification, for example an e-mail push notification, when important events happen in the system, such as the end of a script's execution, a command failure, an end of session, etc.. See Section 6.2 for how the system can provide such services.

2.8 External resources

For the most part, execution of FTIL is constrained to the environment specified so far. Two FTIL commands, however, refer to resources beyond this environment. These are "`exec()`" and "`import`". In both cases,

1. The external resource is accessed specifically by the coordinator node, and not any other node,
2. The external resource is specified by a URI; it may reside in the Auxiliary DB, in the manual input area, or elsewhere entirely,
3. The external resource is a text file, containing FTIL code.

The difference is that "`exec()`" treats the code in the URI as FTIL commands to be executed immediately, essentially as though they were typed by the user at the command line, whereas "`import`" creates a `module` entity that stores the code (which is a collection of scripts) for later use.

2.9 Exit status

Commands in FTIL have an *exit status* of "success", "fail" or "exit".

Success indicates that the command has completed successfully and execution can continue to the next command.

Fail indicates that the command has failed. This can happen for any of the standard reasons such as syntax errors, use of undefined variables, incorrect types for the operation, etc.. However, in FTIL there are also additional checks, as described in Section 2.11, that are meant to verify that all relevant nodes agree that the command is valid, that they are willing to execute it,

and that it is the same for all participating nodes. Together, we call this list of conditions that need to be verified prior to FTIL command execution the command's *prerequisites*. We refer to their validation as *prerequisite checking*.

Exit indicates that by a designed choice, whether directly by a user or by the implementation of an automatic policy, execution has been terminated. There are only a few commands that can return a status of "exit". They are the commands "`confirm()`" and "`limit()`". The former is a command that is executed locally at the coordinator node; the latter is executed in the standard way, i.e. on whatever nodes are in the command's execution scope. Both commands may fail if there is, for example, a parsing error in interpreting their parameter, but otherwise return either a "success" or "exit".

We refer to both failed and exited commands as *aborted*.

Aborted commands report an exit message to the user, but this message cannot be intercepted by FTIL in any way, nor can the exit status. It is merely displayed to the user.

If an abort occurs from inside a script, this stops not only the aborted command but also the calling script and all other scripts on the execution stack. Execution returns to the command line, and there is no way for the FTIL user to stop this. In such a case, FTIL also displays the execution stack position at the time of abort, at the level of individual lines of individual scripts.

Clean-up following an abort depends on the abort type, as described in Section 3.13.6.

If the "back-ups" (save points) described in Section 2.5.1 are not done after every FTIL command, after a command abort the user may find themselves in a state older than one in which the system aborted. There should be clear prompting for the user regarding when, i.e. at what execution stack state, the restored back-up was taken, in addition to the prompting regarding where the abort occurred.

2.10 Command scoping

With the exception of some locally-executed commands (such as the "`dir()`" command, allowing a user to list the presently-defined FTIL variable identifiers) every FTIL command is divided to two parts: an optional *scoping* part and a mandatory *action* part. We denote the scoping by an "`on`" prefix:

```
on(scope) command
```

Scoping determines the participants in a computation. From the perspective of FTIL semantics, all nodes that are in scope execute the command and all nodes that are not in scope are idle for the duration of the command's execution.

From the perspective of the FTIL executor, however, scoping is a little more complicated. The FTIL executor recognises for each command two scopes, an *outer scope*, which is the scope in which the full "`on(scope) command`" instruction is executed, and an *inner scope*, which is the scope in which "`command`" is executed. The command runs on the inner scope.

If a command is run without an "`on(scope)`" clause, the default for the inner scope is the outer scope. Otherwise, scoping may narrow the outer scope. Scoping may not add into the inner scope nodes that are not in the outer scope. Attempting to do so is an error and will result in the command failing.

The scope is defined as an FTIL value of type `nodeset` or `nodeid`. This value can be described in any way, such as via a system constant, a variable, or by means of a computation. The scope definition is resolved at the coordinator node (Technically: by the Compute Manager) and any computation leading to it must be local to that node.

Note 2.10.1. Within scripts, FTIL allows scoping as its own, separate command, creating a scoping block. Instead of

```
on(scope) command
```

the FTIL programmer can write

```
on(scope):
  command
```

to the same semantic effect. From the perspective of the FTIL executor, this splits scoping from command execution entirely.

Thus, for most purposes, one does not need to worry about inner scopes and outer scopes. Rather, one can consider scoping as a separate command to the underlying one, and each of the two commands works under its own scope.

Throughout most of this document, when we use the term "scope" we mean it in this sense. So, when discussing the scope in which scoping is done, we implicitly mean the outer scope, but when discussing the scope in which commands are run, we implicitly mean the inner scope. The terms "outer" and "inner" will only rarely be used explicitly, for disambiguation.

2.11 FTIL execution modes

FTIL execution occurs in one of four modes. Which mode is used for a specific instruction depends not on the instruction itself, but rather on where it is executed from. Command-line execution, for example, is different to execution of commands from a script. If the same command appears in the command-line and in a script, it may be interpreted differently in each case. Importantly, however, if a script is invoked at the command line, the execution of the command invoking the script follows the rules of command-line execution; it is only the commands *inside* the script that use script execution mode.

The idea of FTIL is that it provides a safe execution environment for multi-party protocols, where the parties do not necessarily trust each other. The four execution modes differ in the level of autonomy they provide for the FTIL user. They are on a spectrum that begins with a mode in which the user is allowed maximum autonomy in running local commands, but because of this is provided with the least amount of power to instigate remote operations, and that ends with a mode in which the user can run the most sensitive of remote commands, but because of this works under the tightest controls.

We describe the environments here in the order implied by this spectrum. We refer to this order as the execution *level*. A higher level means less permissiveness for user agility, more power for controlled operations.

2.11.1 Command-line mode

The first mode deals with how commands are executed at the command line.

In command-line execution the outer scope is always the set of all nodes.

This first level of execution, the most permissive of the four, is unique in that it allows the FTIL user to refer to external resources, which is done by use of the commands "`exec()`" and "`import`".

Prerequisite checking when a command is read from the FTIL command-line is handled as follows.

1. The command is read and parsed by the Compute Manager, which may check for potential errors. If errors are found, the command fails and is aborted.

2. If no errors are found, the Compute Manager sends the command to be executed to the Segment Managers that are in the command's (inner) scope. The scope definition is also sent, as it participates in the command's prerequisite validation at each node. Note that the scope definition is passed to the Segment Managers *by value*. In other words, if the scope is defined, for example, implicitly, by a variable or by a computation, it is not the name of the variable or the code of the computation that is shared with the Segment Managers, but rather the resolved scope, i.e. the explicit list of which nodes are to be in scope.

3. The participating Segment Managers all send the command information received from the Compute Manager to each other, in order to validate that all received the same command. This is the first check on the command.

4. The participating Segment Managers continue with any other error checking, individually, and ultimately send to all other participating Compute Managers whether their prerequisite checks, in total, passed or failed. At this stage, for example, it is checked that all variables required for the operation are defined in each Segment Manager's Variable Store and are of the correct types. This is a local check that repeats a check done by the coordinator. Conceptually, it is performed by the Shadow Compute Managers.

5. Once a node has ascertained that the local prerequisite check has passed successfully not just for itself but for all other participating nodes as well, it reports this to the Compute Manager. If any checks failed, the Segment Manager reports this to the Compute Manager with a message to be propagated to the AUSTRAC user. Any error detected at any point aborts execution immediately. A node does not continue processing of a failed command.

6. Typically, command-line execution will not involve inter-node communication, but if any is needed (e.g., in order to handle an "`import`" command) this is performed now.

7. The Compute Manager collates the statuses from all participating Segment Managers. If any failed, it propagates their message to the AUSTRAC user and communicates to all Segment Managers that execution has been aborted.

8. Otherwise, it signals to the participating Segment Managers to execute the command.

9. The command is now executed at the nodes.

10. If an unrecoverable error occurs at this point, the Segment Managers signal a failure to the Compute Manager and to each other.

11. If any Segment Manager signalled failure, the Compute Manager will inform both the user and the participating Segment Managers and will abort execution as described in Section 3.13.6.

12. Otherwise, the Compute Manager waits until all Segment Managers have reported success (which they report also to each other) and returns the user to the command prompt.

2.11.2 (Unsigned) script mode

The second mode deals with how commands are executed within unsigned scripts.

All levels from this point on deal with in-script execution, and the principles of in-script execution are the same in all cases.

A script is essentially a canned sequence of FTIL instructions, which may accept variables as parameters. Scripts support flow control (such as "`while`" and "`for`" loops and "`if`" statements). However, the conditionals of these flow control instructions are evaluated solely by the Compute Manager and not by the Segment Managers.[7] Commands inside FTIL scripts can themselves be script invocations.

[7] Here and elsewhere, statements such as this should be understood as a requirement that such evaluation be done on the coordinator node, not at the peer nodes. In fact, it may not be possible for peer nodes to evaluate these conditionals independently. The statement that it is "not done by the Segment Managers" merely reflects the idea that the Segment Manager on the coordinator node does not have a special role. Any functionality special to the coordinator node is therefore described as a Compute Manager functionality, which is to say separate from the Segment Manager functionality. In actual software implementation, there is no problem with the Compute Manager delegating the actual evaluation of the conditionals to its local Segment Manager so as to avoid code duplication.

Additionally, here and elsewhere, when we say that a computation is handled by the Compute Manager it also means that it utilises data stored in the Variable Registry. For a more detailed discussion of where and how Variable Registry data is used in the system, see Section 3.7.

Commands inside scripts may also include their own scoping directive. In addition to the "`on(scope) command`" format of scoping, inside scripts the "`on(scope)`" directive can itself be used to define an entire block of commands wherein the scope is set. For the commands within such a scoping block, the new scope definition becomes their outer scope.

Similarly, when a script is called, the inner scope of the command running the script becomes the outer scope for the commands run within the script. If one of these commands is itself a script call, this leads to scope nesting. The old scope is restored at the end of the inner script's execution.

If no scope is given explicitly, it is the largest possible scope in the context, i.e. the outer scope.

The value of `scope` in a scoping directive, the conditional in an `if` or `while` clause, and the range of the iterator variable in a `for` loop—these are all evaluated by the coordinator node before the commands in their block are evaluated (or, in the case of `while`, before each time the block is evaluated). They are, however, not communicated to any other node. Peer nodes remain oblivious to all these locally-run commands.

What *is* communicated to peer nodes is only individual commands, when they are executed. These are communicated together with their inner scopes to all nodes within these inner scopes. Any node not involved in the running of any given command is not made aware of it at all.

The term "individual command" covers in this case also the start and end of the execution of any script. In this way, although peer nodes are not informed of the commands they are not participants in, they are nevertheless still able to follow the execution stack while they are in scope.

Otherwise, the process of executing a command inside a script is the same as the process of executing it from the command line.

2.11.3 Signed script mode

The next mode of execution deals with commands that are run inside module scripts.

Modules are composed of a collection of scripts. When an `import` command is called, this creates a new entity in the system that is of type `module`, which stores the scripts. The scope in which this entity is defined is the scope in which the `import` command is executed. These nodes then become the only nodes that can run the imported scripts.

As a result, module scripts enable far more rigorous verification than non-module scripts. As the Compute Manager runs on the coordinator node, each peer node's Shadow Compute Manager follows along, performing the same run-management in parallel in order to ensure that the Compute Manager is doing its job properly and is not straying from the protocol. The Shadow Compute Managers assert at every instruction, among other things, that the correct next line is being executed.

The Shadow Compute Managers can do this regarding all aspects of the execution, except that they cannot resolve on their own `if` and `while` conditionals, `for`-loop iterator values, and the scopes used in scoping commands. These all continue to be evaluated by the coordinator, but their values do get communicated to all peer nodes that are within the commands' *outer* scope, as does the iterator variable in a `for`-loop.[8]

When executing a module script, the FTIL executor performs the following.

1. First, the Compute Manager transmits to each Segment Manager within the command's *outer* scope the command to be run, as well as its (inner) scope, the identity of its module,

[8] As will be explained in detail in Chapter 3, technically, the variable holding the iterator value to a `rm`-loop is treated by the Segment Managers as a shared, constant value, whose scope is the scope of the loop command, but this variable's value is also stored in the Variable Registry, making it visible to the Compute Manager as well. The fact that the variable is "constant" means, in this context, that the variable's value cannot be changed by any command, nor can the variable be deallocated or reassigned. It is deallocated automatically at the end of the `rm` block.

The full details of the FTIL mechanics of flow-control blocks are given in Section 3.13.4. Also, communicating the scoping/flow-control parameter values to nodes beyond the coordinator node may or may not be a privileged operation, depending on details discussed in Section 3.7 and summarised in Section 2.11.4.

and where in the module file the next line to be executed is. Contrary to unsigned script execution, in module scripts this is done for every executed line in the module, including locally-executed portions like `for` and `while` loops.

2. All nodes within the outer scope go over the prerequisite validation process. This is essentially the same process as was described in lower levels of execution, but differs in that this time all nodes in the outer scope perform validation, not just those in the inner scope, which actually execute the underlying command. It is performed through the following steps.

   (a) The outer scope nodes exchange between each other all material received from the coordinator node, to verify that it is the same in all nodes,

   (b) All outer scope nodes verify that this is the correct command to be executed.

   (c) All inner scope nodes additionally verify that they are able to execute the command correctly (e.g., that all necessary variables are defined and are of the appropriate types),

   (d) The inner scope nodes inform all outer scope nodes that they agree to execute the command,

   (e) The outer scope nodes inform the Compute Manager that they are ready,

   (f) The Compute Manager greenlights the execution,

   (g) The inner scope Segment Managers execute the command (which may include multiple recursive executions of the same process described here),

   (h) The inner scope nodes inform all outer scope nodes that the command has finished executing successfully, once it has, and

   (i) The outer scope nodes inform the Compute Manager.

Because the Shadow Compute Managers must be able to follow along the execution path, they are notified not only when a command needs to be executed, but also when the end of a script is reached.

Note that this means that during execution of a signed script Segment Managers that are at any point scoped out can see that the command scoping them out is being executed, but do not see details into its execution. For example:

- If a Segment Manager is being scoped out by a scoping block, the Segment Manager will not know (other than what it can infer independently), regarding commands within said block that have their own scope definitions, which Segment Managers are to execute which commands.

- If a Segment Manager is being scoped out of an "`if`" block, the Segment Manager will not know whether the condition of the "`if`" was true or false.

- If a Segment Manager is being scoped out of a "`while`" block, it will not know how many times the "`while`" block has been repeated.

- If a Segment Manager is being scoped out by a call to a sub-script, it will not know the execution path taken inside the sub-script.

The purpose of this mechanism is for all participating parties to be able to verify that the script is properly executed, without divulging unnecessary extra information.

Other than these differences regarding the execution model of signed scripts vs unsigned scripts, there is also a difference between these in their FTIL semantics: signed scripts allow one command type that unsigned scripts do not, namely the assignment of a signed script as the method of an `object`. (See Section 3.13.5.)

2.11.4 Privileged mode

The fourth and final execution mode is privileged execution. This is a mode in which some module scripts (and all commands therein) are executed. The exact rules regarding when privileged mode is entered and when it is exited are discussed in Section 3.15. For the purpose of describing the executor, only two details are important.

1. The decision of whether to run a module script in privileged mode or not is made independently at each node. This makes privileged mode and signed script mode special, in that they may co-exist concurrently.

2. The node-specific information based on which it is decided which mode to run in is the node-specific choice of which scripts to *certify* and which not. Script certification is described in Section 6.3, but in principle each node can decide regarding each module script whether or not it is certified. (This is not one-to-one related, however, to which script runs in privileged mode.)

In terms of the executor, privileged execution is identical to non-privileged execution, except for its behaviour with respect to a small set of FTIL commands known as the *privileged* commands.

The name "privileged commands" is somewhat misleading, in that it implies incorrectly that these are commands that can only run in privileged mode. In fact, each of these commands has its own, unique, complex prerequisite check sequence, and while their "privileged" nature does entail that part of this check involves the execution modes, even that part is different from command to command.

A full description of each command and its prerequisites is given in Chapter 3, because these prerequisites are part of the semantics of the FTIL language and interact closely with other language features. In rough outline, however, the privileged commands and their individual restrictions—or at least those that impact execution modes—are as follows. (We recommend that readers revisit this section after familiarising themselves with FTIL syntax and semantics in Chapter 3.)

The "`rm`" command: If `transmit` is called in a given (inner) scope, all nodes within that scope must be in privileged mode,

The "`rm`" command: If `broadcast` is called, the coordinator node must be in privileged mode,

Flow control and scoping: The commands "`if`", "`for`" and "`while`" are FTIL's flow control commands, and "`on`" is FTIL's scoping command. Each takes a parameter. In "`if`" and "`while`" this parameter is a conditional, in "`for`" it is the loop's range, and in "`on`" it is the scope definition. When these commands are executed in a scope that is not just the coordinator node, and when their parameter is not a shared variable (See Section 3.2.2) within the scope in which they are executing, the language implicitly `broadcast`s the parameter to all executing nodes, and this follows the same restrictions as a normal `broadcast` would. Otherwise, these commands do not require any special privilege.

The "`rm`" directive: If "`modifying(obj)`" is called on a variable, `obj` and the (inner) scope of the command is the same as the scope of `obj`, all nodes in this scope must be in privileged mode. Additionally, if "`modifying(obj)`" is called on a variable, `obj` and the (inner) scope of the command is narrower than the scope of `obj`, each node in the scope of `obj` must have certified the specific script within which the "`modifying()`" block is directly located, regardless of its present execution mode.

The "`rm`" and "`rm`" commands: The prerequisites for these commands, explained in full in Section 3.10, include the verification of particular conditions about their operands. However, the exact conditions to be verified depend on whether the command is executed in privileged mode or not. Note that successful execution, in turn, impacts the variable these commands are applied on, which changes how they interact with later FTIL code, and specifically with the prerequisite checking for such code.

If any precondition fails, whether it is related to privilege or not, the command as a whole fails and the entire execution stack aborts.

Each node needs to keep track independently of which scripts it is running in privileged mode and which not.

The privileged commands "`transmit`" and "`broadcast`" are the only FTIL commands where the FTIL user can explicitly request data to be transferred between nodes. Where such communication is needed, it happens at the stage in the execution process after the command is greenlit for execution. In addition to any data that requires communicating that is sent at this point, the nodes also send to each other the type of the variable whose information is being sent. This is checked at the receiving end against the expected data type; if there are any discrepancies, this generates an error. The communication protocol chosen will be resilient, in the sense that if a node goes down for a short period[9], communication can resume without loss of information once it goes up again. When all communication has been done and any other part of executing the command has been performed successfully, the Segment Managers signal success to the Compute Manager.

Notably, because FTIL carries runtime type information for every variable, each Segment Manager can determine exactly the type of the arguments in a "`transmit`", and can therefore interpret correctly any stream of data arriving at it from another Segment Manager and deserialise it appropriately. The purpose of also sending the type information from the sending Segment Manager is an extra safety mechanism, designed to ensure that the two Segment Managers agree in their views of the state of the system.

This is not the case, however, with the FTIL command "`broadcast()`". In this command, the receiving peer node may not be aware of the existence of the broadcast variable prior to the "`broadcast()`", and will need to receive from the Compute Manager a type hint. In the case of "`broadcast()`", the type of the destination variable is always equal to the type of the broadcast source variable.

2.12 The FinTracer execution context

Although we have so far implicitly assumed that there is only one "FinTracer system", practical considerations will probably cause us to want to be able to support either multiple systems, or multiple possible configurations within the same system.

A user may wish to specify which Compute Manager they connect to, which Segment Managers, and within the Segment Managers which resources to connect to and with which parameters, in order to find the appropriate data stores. (These are, of course, merely examples.)

Though none of this may be relevant for an MVP, longer term we envision access to the FinTracer system to be done from inside a larger environment (such as a Python command-line or a Jupyter Notebook), and all these parameters to be defined by means of a context object that will need to be initialised and set prior to logging in, and that user log in will, in fact, be performed via one of its methods.

Such a solution would be analogous to the use of a Spark context within a PySpark environment.

Discussions of any such execution context solution are beyond the scope of the present document.

[9] Exact length TBD.

Chapter 3

The FinTracer Intermediate Language

3.1 Introduction

3.1.1 Chapter overview

The present chapter details the FTIL language. It does so by going bottom up, introducing concepts one-by-one, in a logical fashion, attempting to only introduce new concepts once all prerequisites for them have been previously introduced, and once they can be discussed in full, exact, detail, with minimum reliance on "forward references" to later concepts. Forward references cannot be eliminated entirely, as no concept is so completely isolated that it can be introduced first. As a result of this, particularly in the initial sections, one still finds cases of such mentions of later concepts, and especially when it comes to exemplifying material, but by and large the ordering of the chapter was meant to minimise the reliance on these.

Following this introductory section, introduction of FTIL begins, in Section 3.2, by describing the fundamentals of referencing, how anything can be referred to within FTIL. The section introduces some basic terminology that will be used throughout the chapter later on. Section 3.3 and Section 3.4 then continue with further items of basic housekeeping.

In Section 3.5, we introduce the FTIL types, which is to say "What kinds of entities *can* the FTIL programmer reference?" In Section 3.6, we introduce the few such entities that are already pre-defined. In Section 3.7, we show how these fundamentals can be joined together to create a command, and how that command is interpreted. Commands in FTIL manipulate the data that is stored in the system. This data is listed and catalogued in Section 3.8 and Section 3.9.

We now move from discussing the data to discussing the instructions that manipulate this data. We begin in Section 3.10 by introducing labels, which are the FTIL equivalent of interfaces and form the connective tissue between data and operations. Then, in Section 3.11, we list all basic commands in the language. In Section 3.12 we show how these commands can be strung together to form more powerful commands, and in Section 3.13 we introduce scripts, which, in turn, give the programmer the ability to execute commands one after the other with full flow-control capabilities.

Finally, Section 3.14 introduces modules, which are FTIL's means of packaging multiple scripts together in order to distribute them, and Section 3.15 concludes the chapter by explaining FTIL's privileged execution mechanism, which determines what features are available to the FTIL program at any given time.

This bottom up approach continues also through Chapter 4, where we use FTIL itself in order to program additional functionality. We begin by programming simple utilities such as a key manager, and slowly build up to more complex functionality, reaching in Section 4.11 to a complete implementation of the basic FinTracer algorithm.

Such a systematic bottom-up presentation is helpful to a person wanting to implement the

FTIL language (which we do), and is also helpful as a reference manual for later users of FTIL (which will also be very useful) because every concept is introduced in a single place, where all caveats and all gotchas are listed explicitly. Unfortunately, such a presentation is not conducive at all as a person's initial introduction to the FTIL language. Because initially this is going to be this document's main use, in this section, we provide a short, high level overview that can provide a gentle introduction to the FTIL language.[1]

Unlike the rest of this chapter, this section should therefore not to be thought of as presenting requirements. Its statements should be taken as conceptual, describing the intent of the language, and may present it in a simplified light that skips over many of the fine details. Such details will become important later on, but not for the purpose of initial introduction.

3.1.2 Why FTIL?

The purpose of the Alerting project is to allow collaboration between stakeholders in the fight against financial crime, and the FinTracer system is the technical solution that delivers this, by allowing secure, privacy-preserving distributed computation involving all relevant parties.

There are two factors that make these computations secure and privacy-preserving. The first is the computing protocols involved, that meet the conditions of privacy preservation. The second is the fact that each party can transparently see what is running on its node and how that is coordinated among all nodes, so as to verify that the protocols are adhered to.

To make matters more complicated, each intel investigation defines unique typologies, so it's not the case that the system can merely support a number of canned algorithms. The system needs to be flexible.

Moreover, privacy preservation is a holistic property. It is not enough to test whether each step in a process is privacy preserving, the end-to-end process must be privacy preserving. The upshot, in terms of the system's requirements, is the following:

1. Any information that is stored at any node of the system must be stored in a transparent way: it exists in the form of individual variables, each individually interpretable by the node operator.

2. For each such variable, it should be clear what is allowed to be done with it, and what safety guarantees it provides.

3. The system must define rules that specify these safety guarantees and that uphold them by intrinsically prohibiting operators from executing privacy-non-preserving operations.

4. Participants must have the ability to inspect and ratify these definitions before they impact computations on their nodes.

5. The system of guarantees thus defined must be powerful enough to ensure that such ratification does not need to happen frequently. Even though each investigation will run multiple unique queries, these should typically not require new ratification. Typically, ratification is only needed when a new algorithm is introduced.

In this chapter we describe the basic structure and features of the FinTracer Intermediate Language (FTIL), which was designed as a solution for these needs.

FTIL is available to the AUSTRAC user of FinTracer, once logged into the FinTracer coordinator node.

Despite its name, FTIL does not necessarily need to be thought of as a programming language. The individual FTIL commands can be thought of as the parameters of the API calls that manipulate stored objects, and the FTIL execution model can be thought of as the system of rules that manage safety guarantees and determine which API is legal at any point in time.

[1] Readers hoping for an even gentler introduction to FTIL will do best to start with Chapter 4, where examples of FTIL in action are given.

Importantly, managing such safety guarantees requires an understanding of context. For example, while it is normally unsafe to transmit a variable's value between computation nodes, there are specific contexts, specific steps of specific algorithms, where a given, specific variable's value can be safely transmitted between a given pair of nodes. These contexts are encapsulated as FTIL scripts, chaining several FTIL commands.

Thus, while FTIL looks like a fully-functioning language, its purpose is merely to embody the requirements listed above, and to define the rules of privacy-guarantee manipulation. It is certainly not the specific syntax of FTIL that matters here, and it is possible to "implement" FTIL, in the extreme case, without even a parser: FinTracer code can be represented in any standard language as programs calling the relevant APIs with the relevant parameters. The FTIL executor only needs to manage the appropriate guarantees.

As a case in point, FTIL must, as described above, contain mechanisms that allow AUSTRAC to define a new algorithm (which will not be an end-to-end end-user query, but rather a single building-block in the process), and a participating RE must

1. Be able to inspect it, offline, in order to ratify that said RE allows its use, having deemed it safe, and

2. Be able to inspect it, online, as it is run, in order to verify in real time that the computation AUSTRAC is initiating matches what was previously ratified.

In this document, we describe this as being done by storing a file of FTIL scripts, known as a *module*, for the RE to certify. It is straightforward for FTIL code running in the system to be matched against the contents of such a file (by means of a digital signature) in order to verify the certification. If FTIL is implemented not as a language, but rather as a set of API calls, one would still need such files to be transmittable to the REs, certifiable by them and matchable against real-time API requests, but the process to do so may be a little more roundabout than simply having the file list the API requests in the form of FTIL code.

Thus, consider FTIL's formatting as a language merely as a matter of convenience in describing the requirements of the system, and FTIL's specific syntax as nothing more than one of many ways to represent these requirements.

Having said this, we do believe that a language-like implementation of FTIL is the most straightforward and end-user-accessible way to implement it so as to attain its design goals, which is why it is presented in this light here.

Out of necessity, in order to keep the discussion concrete, a specific syntax for FTIL is defined here and used in examples, but *caveat lector* that this syntax is not necessarily going to be the final syntax of the system. Detailed considerations regarding ease of implementability, for example, may require it to change. However, any alternative syntax will need to carry the same semantic capabilities and meet the system's overall design goals as discussed in this introductory section.

3.1.3 FTIL in a nutshell

Though FTIL was designed to be a simple language to implement, it does include, as minimalist stubs at least, much of what makes a powerful modern language. FTIL is a procedural, managed, strongly-typed, interpreted language. It supports introspection and templates.

The language itself was initially fashioned after Python (and still maintains Python-like block-definition syntax). Its top-level variables are references-to-values that use reference counting and dynamic typing (i.e., variable types are determined by assignment). However, it diverges from Python in being a copy-on-write language, and also having significant amounts of static typing.

Some points where dramatic restrictions were added to FTIL compared to what one would expect from an all-purpose language/environment are:

1. FTIL has very little in terms of user permission management. All non-script variables are equally visible to any user of the system.

2. FTIL has no notion of a job and no parallelisation of any kind. The closest to a "job" the language supports are scripts.

3. The language has no exception handling. Errors lead to immediate aborted execution of any command and any script that command is part of.

4. The only type of non-synchronous behaviour allowed is that the FTIL user may at any time cause the present execution to abort (e.g., via a "`^C`").

5. FTIL does not have an extendable type system. Some useful types are pre-defined as part of a "standard library", but for any user requiring anything more than what is either a pre-defined type or a pre-defined template, the only tool in the language is the variable type "`object`", which is a generic type that the user can populate with any desired members and methods.

FTIL is interpreted. It supports scripts, but individual commands can run on the command-line. When they are run, they use the format

```
on(scope) command
```

where the "`on(scope)`" part, known as the *scoping* part, is optional. Scoping tells FTIL on which nodes to run the command. If no scope is given, FTIL will run on all nodes it can. While on the command-line, this means commands will run on the set of all nodes in the system.

FTIL is better described as a *parallelised* language than as merely a distributed language, because it executes in a fully-synchronised manner across all computation nodes. It can also be said to be a *SIMD* language, because at any given time only one command can run, though the same command can run simultaneously on multiple nodes, each node using its own local data.

3.1.4 Design considerations in FTIL

The design of FTIL was motivated by the following criteria.

Functionality: FTIL is the interface language to the FinTracer system, it is the interface available to the AUSTRAC user of FinTracer once logged into the FinTracer coordinator node, and must therefore support the functionality that the system was built to deliver. This includes the basic FinTracer algorithm as well as its variants, its advanced uses, its extensions FinTracer DARK [7] and oblivious querying [4, 5, 8], etc..

Extendability: The functionality of FinTracer is not a closed set of functions. The system's functionality must be able to extend to the search for crime typologies that are not known and not anticipated at the time of the system's design. Moreover, we have seen with the FinTracer system an incredibly rapid progression of algorithms and algorithmic solutions, and we would not want to now freeze this progression, which seems at the time of writing to be very much ongoing, with both better methods to attain existing algorithmic goals and new methods to attain novel algorithmic goals being documented all the time.

Privacy by design: The system must support not only the privacy-preserving execution of existing algorithms, but also safeguards to ensure that it maintains its privacy guarantees also in future use, even when extending its functionality as discussed above. To clarify: guaranteeing privacy is a range of requirements aimed at a range of potential situations. For example, one needs to consider data leaks due to (i) human operator error, (ii) implementation bugs, (iii) unexpected algorithm weaknesses, (iv) unintentional misuse, (v) intentionally malicious use of the system, (vi) hacking into the system that entirely circumvents its designed interfaces. This is not an exhaustive list, and even within this list there are distinct subdivisions. For example, the system has a range of distinct operators, and the character, risk and severity of potential human operator errors all differ between them.

Understandability: FTIL cannot be a "write once, read never" language. Its code must be understandable at a semantic level of "what we want to execute". The reason for this is that much of the ongoing governance of the system's use will be at a policy level, done by non-programmers. It is therefore neither possible nor practicable to try to guarantee the safe execution of FinTracer code purely by means of extensive processes of code and algorithm review.

Efficiency: FinTracer will perform Big Data processing (requiring efficient code implementation) on large data (requiring efficient storage and communication).

Implementability: The FinTracer system is built by a small team working under time and budget constraints. The FTIL language (and the rest of the Purgles Platform) must be realistically implementable within these constraints.

The problem with these design goals is that they include many apparent contradictions. Effectively making the system extendable, for example, is essentially impossible without making its potential functionality complete enough to circumvent privacy by design. Understandability requires the system to communicate with the user at a high, semantic level that hides much of the technical complexity of execution, but hidden technical complexities might impact potential risks that need to be addressed, hence risking privacy by design, and, furthermore, might impede efficient execution.

Perhaps most importantly, each of the design criteria listed before "implementability" introduces additional complications and additional functionality that will need to be supported. As we will see, in total, in order to meet all requirements FTIL will end up being quite a full-featured, modern language. That, of course, runs directly against our need to make it minimalist, for it to be at all implementable within the given time and budget constraints.

FTIL, as presented in this chapter, was designed with all these conflicting requirements in mind, and we believe it meets them all. However, it does not always meet them in obvious ways.[2]

3.1.5 So... what's the solution?

FTIL is to be the main interface language for the AUSTRAC user to the Purgles Platform in running Intel investigations. We expect such investigations to be long-running, and to accumulate over time data collateral for later use within the context of an investigation (and maybe also even beyond). For this reason, we did not want to opt for a language that is based on the execution of individual programs but rather to have an interactive environment hosting one, continuous, long session. Within this session, the user can create new data artefacts that are stored and are accessible through variables, and later actions can manipulate these artefacts, including by using them in conjunction with other artefacts. Users may eventually choose to discard given artefacts when they are no longer needed, but the data artefacts themselves should not have intrinsically limited lifespans.

In this kind of a long session environment, it would have been unnatural to rely on some static definition for variables, forever limiting a variable's capabilities based on earlier decisions that were made at some past point in time. Instead, we opted for a dynamic environment where items can be assigned their types and capabilities dynamically, and decided to provide the user with the ability to inspect variables at any given time in order to understand what they are and what they can do.

One successful language that has this kind of an operating model is the Python language. We modelled much of the look-and-feel of FTIL after Python, gaining by this not just the good support for long, interactive sessions but also one of its other well-known advantages, namely in being a highly *understandable* language, looking at times very much like pseudocode.

[2] Though the general gist of how FTIL tackles this problem is explained below, readers who are already familiar with FTIL may wish to consult Section 11.4 to see how successful FTIL's approach is, in meeting its design goals.

Importantly, FTIL, like Python, is procedural. In procedural languages the programmer has full control on what the computer executes, which is a main feature we will require for later establishing the language's security.

It is also Turing-complete, and therefore tick-marks the design criterion of *functionality*.

This, unfortunately, is where we need to depart from analogies to Python, however, because the main challenge for which we developed FTIL is one that is not currently faced or addressed by any standard programming language. Namely, FTIL was designed to work in a *distributed execution environment* in which *participants do not fully trust each other*, but do wish to *jointly compute a result* in a way that provides desired new information to pre-specified computation targets, while *preventing all parties from gaining any other meaningful information collateral*.

In academia, this topic of a lack of trust between computation participants has generally been addressed by developing appropriate *communication protocols* that tell each party what it needs to do at any given point in time, and to whom it needs to send which information. These communication protocols are designed to ensure that each party (if it is faithfully following the protocol) is able to verify that all other parties are faithfully following the protocol as well.

The specific issue of wanting to prevent parties from gaining meaningful information other than by design is handled by a special sub-category within communication protocols called *privacy-preserving* communication protocols.

In the context of the Purgles Platform, there is one party, namely AUSTRAC, that decides what to compute at each point in time, and generally speaking AUSTRAC itself is the main information-beneficiary of such computation.

We have therefore designed the FinTracer system to be a distributed computation system where each participating party is a *node* and one node, where AUSTRAC sits, is deemed the *coordinator* node, and is where FTIL is initiated from.

FTIL was designed specifically for implementing privacy-preserving communication protocols, but its main features can be viewed as simply enabling the running of communication protocols of any kind, in a distributed environment where parties do not trust each other.

The reason for this decision was to keep the language as simple as possible, in the interest of *implementability*. Constructs in the language had to be few, simple and generic, with the intent that all higher functionality, including all communication protocols themselves, will be constructed by programming in the language (as is done in Chapter 4) rather than need to be delivered as part of the language.

Specifically, each node in the system (i.e., each participant) keeps all its computation data at any point in time in *entities* in its Variable Store. These entities can be referenced via *variables*. Importantly, all variables must always remain interpretable (because each party must understand exactly what is going on in the protocol, and in particular the portion of the protocol that it, itself, is meant to execute).

In FTIL, data items come with much auxiliary information that describes them: they have a name, a type, etc.., and individual commands manipulate the variables and their data across the various computation nodes in a way that is visible and clear to all participants. This provides *interpretability*.

Commands in FTIL are not simply executed one by one, but are, rather, first communicated full *scripts* at a time.[3] Each script is a software implementation of a communication protocol. Thus, each party has *visibility at the protocol level* into what is currently running, across all nodes.

In fact, FTIL is not merely based on distributed execution but specifically on *parallel* execution. Execution is always synchronised between the nodes, so each protocol is executed in lock-step: every participant not only knows how far along they are in executing their protocol, but also what each other party would have been doing up until the same time. FTIL does not progress to the next command until all nodes have successfully completed processing the previous one. This is a very high level of visibility that allows the *coordinated execution* of arbitrary, complex communication protocols.

[3] Actually, even more than that: full modules of scripts

Beyond simply being able to execute communication protocols, one can think of FTIL as a system that constantly computes the calculus of "What is privacy preserving?". The language does not just manipulate its variables, as described above, but also keeps at all times track of what should be allowable to do with these variables, from a privacy-preservation standpoint.

Some activities are automatically prohibited, in ways embedded in the semantics of the language. Some are automatically allowed. (For example, FTIL recognises that computations that happen locally at each node separately do not impact privacy, so typically these are automatically allowed.)

This, however, leaves a large gap of activities that cannot be automatically determined to be either safe or unsafe. Privacy-preserving protocols are typically complex protocols involving careful use of advanced cryptographic methods, and it takes very careful scrutiny in order to determine whether a protocol is safe or not.

To address this, beyond simple visibility into the protocols being executed, individual nodes have in FTIL a measure of *control*. We have already established that AUSTRAC has complete control over which communication protocols get initiated, but in FTIL each node has *approval* power over what it is willing to execute. Thus, only if all parties agree to run a particular protocol (e.g., after having manually inspected the protocol and having agreed to its terms, being satisfied with its privacy-preservation properties) can the protocol be executed. Otherwise, attempts to execute it will fail, and will be aborted.

This is done by a process called *script certification*. When a script is certified by a node, the node is providing approval to take part in executing the script. In FTIL scripts are packaged into *modules* that are identifiable by their digital signatures. Certificates are given to a specific script (identified by name) within a specific module (identified by its signature). Each node is free to certify or not to certify any script, independently of what any other node does.

To address the three remaining design criteria we haven't yet mentioned, FTIL brings in three new ideas: **privileged execution** is the main remaining ingredient in FTIL's *privacy by design*, **labelled collections** handle much of FTIL's *extendability* and **elementary entities** enable FTIL to attain *efficiency* despite all this baggage. We discuss each in turn.

Privileged execution

As discussed, FTIL is really a calculator for "What is allowable with my variables now?". Attaining privacy by design is really all about understanding what makes something safe and what not. A miscalculation that erroneously concludes something is safe can lead to a breach of data security and privacy. A miscalculation in the opposite direction leads to too much reliance on point-by-point manual certification. In such a scenario there is no extendability: every time a user wants to run anything, it needs to be individually certified. (Considering that allowing new code to be run on a node may require approvals that take months to obtain, every such certification request is incredibly costly, and can render entire investigations non-viable for the platform.) The need to create FTIL in the first place is first and foremost in order to avoid this one problem.

To truly understand FTIL's workings as a "safety calculator", it is best not to think of its scripts as programs, or even as protocols. Instead, think of an FTIL script as a description of a *context*.

As mentioned, the first step in the safety calculation is to separate out those commands that are intrinsically safe (e.g., do not involve any inter-node communication) with those that are not. We call the commands that are not intrinsically safe *privileged commands*, because they require some special permission that will grant their execution.

Instead of requiring scripts to be certified in order to run at all, FTIL, using the distinction between privileged and non-privileged commands, only requires certification in order to run the privileged commands. With this provision in place, users can already run any non-privileged commands of their choice without this requiring certification. Scripts that are composed entirely of non-privileged commands can run this way, too. Certification is still required for scripts that utilise privileged commands, but these can run in any order, including with any non-privileged

commands between them.[4]

To explain the role of scripts as "context providers", consider now the following. By their very definition, individual privileged commands should never be allowed to run as-is, on their own. So, anything that only runs one such command and nothing else should never be certified. A script, however, is an entire design of how to use privileged commands. It details an entire communication protocol. The certifier can now study the entire protocol and reach the conclusion that within this particular context, the particular privileged commands used *are* safe to use.

For example, there is no reason to allow the sending of arbitrary data from Node A to Node B on its own. But within the context of a script, the certifier can see what that data is and where it came from, and based on this decide that the data *is* safe to be shared. For example, in the FinTracer System we purposefully separate transaction data to a dedicated "Transaction Store", whose design is such that any certifier on Node A can be sure that any information included in the Transaction Store on Node A relating to transactions between Node A and Node B is also mirrored in the Transaction Store of Node B. Sending such data over to Node B, as we do in the FinTracer algorithm, is therefore by design safe.[5]

Once one begins to think of scripts as context, it makes sense to have nested contexts, too: if Script A calls Script B, and Script B performs privileged operations, it is enough to either certify Script A (the wider context) or Script B (the narrower context). Indeed, FTIL allows scripts to call other scripts, and it is even allowed for one party to certify the narrower context while another party decides to certify the wider context, instead. As long as the privileged operations are covered in both cases, that's fine.

Importantly, scripts in FTIL can only call other scripts within the same module. The reason for this, again, can be traced to questions of providing adequate context for a script certifier: recall that certification of a script requires the digital signature of its module. Any potential script certifier can now be sure that any certificate they give to a script will immediately become invalid if one were to alter the module. Thus, if Script A calls Script B, the script certifier of A can be sure of exactly what code will be executed. The wider context of A also provides the narrower context of B. No separate certification of B is required.

In keeping with this world-view of wanting to be confident of the execution context, FTIL also has no form of exception handling. Execution of a protocol is done only through its "happy path". Any exception immediately leads to an aborted execution. This makes flow control much simpler, allowing the certifiers to ascertain easily that the script never performs anything other than what it was intended to do.

Labelled collections

The problem with solving extendability just using certified scripts is that it is still quite a limiting solution. The only real possibility for getting novel functionality without requiring new certification is by chaining multiple certified scripts one after the other, each acting essentially independently.

Consider a situation where you have two scripts, Script A and Script B, and the idea is that you compute something with Script A and then pass that result on as input to Script B for further processing. For example, in the FinTracer algorithm we have processes that generate tags corresponding to sets of accounts of interest, and we have other processes that take these sets and manipulate them (for example, in order to deliver them to the AUSTRAC user). How can one be sure that the input to the second script really was generated properly?

FTIL's answer to this is *collection labelling*. A *collection* is a type of FTIL entity, and a *label* is a string that we can pin to a collection. It signifies what the collection's data is fit for. This

[4] FTIL is interpreted. It doesn't actually examine a script to see whether it performs anything privileged in order to determine whether running it requires certification. The script is run, and simply aborts, if it isn't certified, the first time a privileged command is reached.

[5] Besides the information in the Transaction Store, FTIL has an entire category of variables known as the "shared" variables, which are pieces of information that are 'known to be known' to all nodes within a given scope. Actions that reveal the values of these variables within their own definition scopes do not leak by this information. This is another example of an "intrinsically safe" command that FTIL handles automatically and requires no certification for.

can be compared to 'interfaces' in standard programming languages, except interfaces only verify details about the entity that can be verified automatically. By contrast, FTIL's collection labelling allows one to fully specify what the collection is expected to be, at a deep semantic level. For example, if a script generates a FinTracer tag, we will label its output as being a FinTracer tag. If a script expects its input to be a FinTracer tag, for example for the purpose of outputting it to the AUSTRAC user, it will check the label of the input to verify that what it was sent truly is a FinTracer tag.

This collection labelling is a powerful and versatile tool for extendability. Here are some examples:

- It allows one to mix and match one's processes, without this compromising security. For example, there are plenty of different ways in which we know how to generate FinTracer tags and there are plenty of ways in which we know how to manipulate them further. By using the FinTracer tag label throughout, we can use any tag generator with any tag manipulator, while maintaining the invariant that scripts always know that their input is what they expect it to be.

- Scripts often take more than one input, and it is important not only that the inputs are individually "good", but also that together they form a coherent whole. Labelling allows us to do this because collections can contain many data fields (as member attributes). We can package all inputs that we want to be coherent with each other into a single, large collection, and then label the collection to signify that it guarantees this coherence.

- Oftentimes what we want to guarantee is not a particular set of data but rather a particular functionality. For example, in the FinTracer algorithm, we wish to propagate tags. This functionality can be encapsulated inside a "tag propagator". By using labels we can send to scripts that need it tag propagator `objects` (where an `object` is one type of an FTIL collection), without having to worry about how these `objects` actually propagate their tags. For example, maybe we're sending standard FinTracer propagation `objects`, or maybe we're sending FinTracer DARK propagation `objects`. We can do this in FTIL because `objects` carry not just member attributes but also methods, so can be fully encapsulated and provide polymorphism. A script can receive an `object` as a parameter and then call one of its methods to get its functionality. (And it may call several different methods, and be sure they are all coherent.) Collection labels allow the script to know exactly what each method does on a semantic level, even when how that functionality is implemented is completely up to the `object`.

- Method-calling also allows extendability beyond the borders of a single module: scripts can run the methods of any `object` they have, regardless of what module the methods' underlying scripts came from. This is as opposed to scripts calling other scripts directly, which is only allowed within the confines of a single module in order to maintain privacy guarantees.

To complete the picture of end-to-end security, note that the labelling of collections itself is handled using the same security measures as everything else in FTIL: scripts, executed by the AUSTRAC user, can take collections and label them, but these scripts need to be certified in order to be executed because labelling itself is a privileged operation. (To add a level of detail here, the labelling itself is not what impacts security and privacy. What impacts security and privacy is relying on the label in order to transmit data. For this reason, transmitting data is a privileged operation of the kind that fails if the relevant security guarantees are not in place, but labelling does not fail if it is run outside privileged mode. Instead, it leads to what we call a *weak label*. This is FTIL's way of saying that the collection is labelled but on that particular node the label is not trusted. This is useful, because the same label may be trusted by some nodes but not trusted by others.)

Once a collection is labelled, it cannot be modified further in uncontrolled ways. Still, labelling is not like 'const': quite often, collections need to change over time in order to provide the functionality they were designed to deliver. (For example, an `object` in charge of tag refreshing

must be able to update its stockpile of pre-encrypted zeroes.) FTIL allows labelled collections to be modified, but only by scripts that are certified to respect the label. Thus, label scrutiny for collections takes in FTIL a whole-of-lifecycle approach: certification is required to create the label, to modify the collection, and ultimately to trigger privileged functionality.

In total, the combination of certified scripts and labelled collections allows FTIL to understand for every piece of data in the system what it can be used for. This allows both unlimited extendability and rigorous constraints that maintain privacy and security: the system knows what is safe and what not, and allows everything that is secure and nothing that isn't.

Elementary entities

Privacy-preserving protocols tend to be quite intensive in all dimensions: compute, storage and communication. This requires FTIL to be efficient.

In general, languages are optimised either for efficiency or for control. If we were to program in C, for example, we would have been able to maximise efficiency because the C compiler is focused only on optimising the execution itself. On the other hand, if we were to program in Python, the language would have needed to not just perform the desired operations but also to understand what it is doing: Python offers 'introspection', meaning that every entity knows exactly what type it is as well as many other details about itself. Maintaining these takes space and updating them takes time.

In FTIL, as described above, entities do need to know what they are and where they came from in order to maintain all security guarantees. In order to reconcile this with efficiency, FTIL takes the approach of partitioning entity types in the language into two categories: those that maintain security guarantees, and those that can manipulate data efficiently. Specifically, the collection types are those types that manage security guarantees, which they do by use of labels, whereas the entities that manipulate data efficiently are known as *elementary entities*.[6]

One can think of the elementary entities as containing the core data in the system, with each participant in a computation seeing exactly the part of this data that they are allowed to see. Everything else in the system, including all information maintained by collections, is 'data about data'; it is "bookkeeping information" that allows the language to know what is allowed and what not at every point in time. The core data is distributed among the Variable Stores of nodes, each node seeing only what it is allowed to see, whereas the 'data about data' is stored in its entirety in the Variable Registry at the coordinator node, with the relevant portions of it copied out to the individual nodes, for each node to be able to ascertain individually that each action it is asked to do is valid and legal.

Elementary types include both basic, fixed-sized types (like numerical types and structs) and *containers*, which are types like lists and dictionaries, that may store vast amounts of *elements*, each of which is itself of a basic type. FTIL stores overheads per elementary type, but not for each element within a container. Elements that are packaged inside a large container are stored efficiently, can be manipulated efficiently, and can be serialised and deserialised efficiently as part of inter-node communication. This is aided, in part, by the fact that FTIL supports only very simple container types. Lists of lists, for example, are not allowed, and the element type for a container must always be the same across the entire container.

Because FTIL specialises in privacy preserving protocols, and because it is intended for financial data, FTIL comes built-in with many types that are specific to these domains, and has much built-in functionality that is implemented efficiently for these types. For example, there is built-in support for cryptographic types, including for encryption, decryption, and other cryptographic functions.

For containers, FTIL also supports what we call the *mass operations syntax*. This is like a mini-language within the language that can only manipulate containers and can only do so in ways that are data parallel. The idea is that such operations are good candidates for efficient implementation, e.g. on GPUs.

[6] The idea of requiring efficiency only from some types is not unique to FTIL. It exists, for example, also in Python's mathematical extensions, such as `rm`.

In total, the mass operations syntax is what allows FTIL maximal flexibility in extending its functionality when it comes to simple, non-privileged operations that only need to manipulate data in highly predictable always, and which have no direct security implications. By having such syntax within the language (rather than integrating any desired capabilities by means of language extensions) certifiers of scripts can examine all such operations, understand exactly what they do, and be able to determine whether they provide the appropriate context in each script.

The mass operations syntax looks like a 'list comprehension', but in terms of actual functionality it was modelled after database querying primitives which are known to be universal (and are, in fact, the same primitives one can find in any Big Data processing environment, such is in Hadoop, Spark, etc.)

The final efficiency feature for elementary entities is that their variables use multiple tricks in order to eliminate the need for data copying whenever it is not absolutely necessary. Variables in FTIL are *references to entities*. As a result, assignment from one variable to another doesn't require data copying. The new variable merely becomes another reference to the same entity. However, unlike in Python, in FTIL the separation between variables and elementary entities is purely for efficiency. It can never impact the final result of any computation. If multiple variables refer to the same entity and one of the variables attempts to modify its entity in-place this is allowed, but it triggers a copy-on-write, so all other variables referring to the same entity are not modified. Instead, the entity is copied away and that copy, now pointed-to by only the one variable, is modified.

FTIL uses a similar trick (this time using what we call *identifiers*, which are references to variables) in order to make sure that no data copying is needed also in passing parameters to scripts as well as in returning values from scripts.

(In fact, it is worth mentioning that FTIL tries not to restrict its programmers other than for direct security reasons. An example of this is that scripts require no form of declaration for their parameters. A user can call a script with whatever types of variables they choose as parameters, and scripts, on the other hand, are free to choose for each parameter in real time whether they will use it as an input parameter, an output parameter or as both input and output. This fits with FTIL's charter of being an "intermediate" language. Its purpose is to be powerful, not to provide guardrails and training wheels. We envision that environments outside of the Purgles Platform that will want to send instructions automatically to FTIL may want to implement their own sets of guardrails.)

Thus, the combination of elementary entities, labelled collections and privileged execution is what allows FTIL to meet all of its remaining design goals, and to overcome the apparent contradictions between them.

3.2 FTIL fundamentals

In this section, we provide a whirlwind tour of some of the most basic, low-level concepts underpinning how FTIL treats variables and data.[7] The purpose of this section is merely to introduce some basic terminology and some fundamental notions that will be used throughout the rest of the chapter, but in the process we inevitably mention in passing many other, deeper and more complicated ideas. All these more advanced ideas will be returned to in later sections, where they will be introduced more systematically. Readers are encouraged to re-read this section a second time after they already have a better grasp of FTIL's mechanics, at which time it is guaranteed to make more sense.

3.2.1 FTIL variable basics

FTIL is a strongly typed language. Each entity has a *type* that is set at its creation time and cannot be changed later. Entities are referenced by *variables*. A variable's *scope* is the set of nodes that have this variable in their Variable Store. It may or may not include the coordinator node.

[7] Don't despair! It gets easier after this.

An entity's type and a variable's scope are examples of *metadata* properties. They are properties of the variable (or, as is more often the case, of the entity it refers to) that are not modifiable by the FTIL user directly but provide important information. For most entity types, FTIL provides read-only access to some such metadata properties via the variables' `metadata` attribute.

For example, for most entity types, if `x` is the name of an FTIL variable that refers to a particular entity, `x.metadata.type` provides the entity's type, while `x.metadata.scope` provides the variable's scope.

FTIL has many different types. These types broadly divide into five categories:

1. *Fixed-sized* types (Discussed in Section 3.5.1),
2. *Container* types (Discussed in Section 3.5.2),
3. *Collection* types (Discussed in Section 3.5.3),
4. *Special* types (Discussed in Section 3.5.4), and
5. *Pseudotypes* (Discussed in Section 3.5.5).

The fixed-sized and container types are collectively known as *elementary* types. This is because the data contents of entities of these types are data elements. This is as opposed to *referential* types. Entities of referential types have attributes that are themselves variables, i.e. references to other entities. Excluding pseudotypes, FTIL's referential types are the *collections* and instances of the `transmitter` template special type. FTIL does not have any explicit pointer syntax.

The scope of variables is usually determined by the scope of the command that creates them. For example:

```
a = 7
del a
on(CoordinatorID) a = 3.5
```

Without going into the mechanics of execution, and considering only the state at the end of processing each command.

- The first line creates a variable "`a`" of widest possible scope and an entity of type `int` of the same scope, whose value, shared in every node in its scope, is 7. It assigns the variable to refer to the entity.
- The second line deletes `a` and thereby deallocates the entity. This is a command that has to be run at the same scope as the scope of `a`.
- The third line re-creates "`a`", but this time only at the scope of the coordinator node. It also creates a new entity of type `float` whose value is 3.5, shared at the same scope, which `a` now refers to.

It would not have been legal to omit the second line, because variables cannot be deleted or reassigned at any scope other than their own.

3.2.2 Type flavours

Limiting our discussion now to only elementary types, FTIL supports two *flavours* of such types. FTIL entities of any elementary type can be in either flavour.

Shared entities are ones whose values are entirely known to all Segment Managers within their scope.

Distributed entities are ones whose values are distributed among the Segment Managers in their scope, such that each node sees only its portion of the value.

Note 3.2.1. Previous versions of this document mention other flavours, including "local". Presently, the term *local* is only used here informally, to describe entities whose scope is only the coordinator node. There are no special properties for local variables, but some computations always occur in the "local" (i.e., the coordinator node) scope, and only use the "local" values of the variables used.

The flavour of an entity is set at its creation and cannot be changed later. It can be accessed, for a variable named `x`, through the `x.metadata.shared` attribute, which is of Boolean type.

FTIL does not allow changing the value of shared entities, which keeps this value equal between nodes.

Entities that are not of elementary type do not have such flavours. However, we will often use terms like "distributed" and "shared" informally for such types, in order to describe whether or not their information is fully known to all nodes in the scope where they are defined.

For example, the `metadata` attribute is not itself of an elementary type, but we still say that it is "shared", because all nodes in a variable's scope see it, and it is identical in all.

However, unlike in the case of elementary entities, which are truly shared, such "effectively shared" entities are not necessarily required to remain unchanged. It is enough that the language maintains their equality across all nodes.

In the case of the `metadata` attribute, its information does get changed by some operations.[8] However, the language is constructed in such a way that such operations must always run in the same scope as the underlying entity and affect its `metadata` equally on all nodes.

Typically, manipulation of shared entities results in other shared entities. With some exceptions (like random functions), when invoking a built-in FTIL function with only shared parameters, the result will be a shared entity.

Consider, for example, the following FTIL code.

```
a = 7
b = a + 1
```

As discussed before, the first line defines a new variable called "`a`" that is of type `int`, whose value is 7, and is a shared, elementary variable defined at the maximal possible scope.

The second line then defines a variable `b`, of the same properties, also shared, but whose value is 8 within the same scope. The reason `b` is shared is that the calculation "`a + 1`" uses only shared inputs. Hence, its output is shared, and that output remains shared also when it is assigned to the new variable `b`.

FTIL provides a built-in function, "`distribute()`" that takes a shared elementary entity and returns a distributed entity with the same value and type, at the scope in which the command was run. The distributed entity is a new entity, so executing "`distribute()`" always involves data copying.

Here is an example FTIL session to illustrate the above.

```
a = 7
b = distribute(a)
```

Here, on executing the second line a new variable, "`b`", is defined. Like before, it is of type `int` and maximal scope, but this time it is distributed and contains the value 7 in each node where it is defined. The value of variable `a` is left unchanged.

Because `b` is distributed, it is now possible to modify its value, and specifically to modify it separately in each node, causing the value to diverge between nodes.

Remark for the advanced reader 3.2.1. The following example is a more complicated version of the first example given in this section. It demonstrates how these concepts interact in real-world usage.

First-time readers should skip this example, and return to it only after they have gained more familiarity with FTIL. The example uses many constructs that will only be explained in later sections.

[8] For example, by collection labelling.

```
a = 7
if a.metadata.type == int:
  _peer_ids = setminus(scope(), nodeset(CoordinatorID))
```

The first command, as already discussed, makes `a` a shared `int`, defined at the maximal scope, whose value is 7.

The `if` statement compares `a`'s type with `int`. Both sides of the comparison are shared values that are defined in (at least) the present scope, and therefore the resulting Boolean value, which is "True", is shared, too.

The "`scope()`" function receives no parameters. Its result value is therefore shared, and "`CoordinatorID`" is a system constant that is of shared flavour. The result of the `setminus` is for this reason also shared: all inputs to it are shared. This shared result is then assigned to a new variable, "`_peer_ids`", so it, too, becomes a shared variable.

3.2.3 Protection

An FTIL variable is called *protected* if it cannot be deleted or reassigned from the entity that it currently references.

As detailed in Section 3.8.3, the system maintains for each variable a *protection counter*. If the counter is nonzero, the variable is protected by FTIL from deletion or reassignment.

Whenever an FTIL assignment defines a new variable, that variable is initially non-protected (has a protection counter of zero). Whenever an existing variable is re-assigned to a new entity—something that can only occur if the variable was non-protected to begin with—it retains its non-protected status.

Remark for the advanced reader 3.2.2. System constants are never "assigned" as such, so are not an exception to this rule. They are protected from their creation.

There are no FTIL commands that specifically allow the FTIL user to make variables protected, but some FTIL operations modify this property as part of their actions.

3.2.4 Immutability

An FTIL variable is called *immutable* if the value of the entity it references cannot be modified in place. A variable is called *const* if it is both protected and immutable.

When a referential entity (a collection or a `transmitter`) is immutable, none of its attributes can be changed, nor can new attributes be created or existing ones be deallocated.

How the immutability property is handled depends on the type of entity.

Below, we discuss elementary distributed entities, `transmitter`s and collections. All other FTIL variables are immutable by their nature, not supporting any operation that modifies them.

For elementary entities and transmitters

In the context of distributed elementary entities and of `transmitter`s, the system explicitly manages a flag (See Section 3.8.3) that determines whether or not they are immutable. If the flag is on for a particular variable (i.e., if its value is "True") the language prevents that variable from modifying the entity it refers to.

There are no FTIL commands that specifically allow the FTIL user to make variables immutable, but some FTIL operations modify this property as part of their actions.

Whenever an FTIL assignment defines a new variable of a type that has an immutability flag, that flag is initially set to False. Whenever an existing variable is is re-assigned to a new entity of a type that has an immutability flag, it becomes non-immutable even if it previously was immutable.

Remark for the advanced reader 3.2.3. System constants are never "assigned" as such, so are not an exception to this rule. If they are of a type that has an immutability flag, they are immutable from their creation.

Multiple variables may refer to the same underlying distributed elementary entity, with some being immutable and others not. This does not pose a problem because FTIL uses a *copy-on-write* system: when a variable (necessarily, a non-immutable one) attempts to modify its underlying entity, if said entity is referenced by only that single variable, its value is modified in place; however, if it is referenced by more than one variable, it is first copied out; the variable (even if it is protected) is updated to refer to the new copy; and then this new copy is modified, this modification not affecting any of the other variables that originally referred to the same entity.[9]

Remark for the advanced reader 3.2.4. The number of references to a given entity may be different from node to node in its scope. Some nodes may need a copy-on-write on modification while others can perform truly in-place updates for the same update operation.

This copy-on-write behaviour of FTIL makes the final outcome of any computation involving elementary entities oblivious to whether different variables refer to the same entity or not. The value of the result would have been the same even if each variable originally referred to a separate entity.

The separation between elementary type entities and their variables in FTIL is therefore purely an efficiency mechanism, allowing values to be copied from variable to variable without any data copying under the hood. (As discussed below, the same is not true for collection types.)

Remark for the advanced reader 3.2.5. The example given above for copy-on-write behaviour used distributed elementary-type entities. It would not have been possible to create an equivalent example with `transmitter`s because `transmitter`s are non-copyable. Every entity of `transmitter` type is referred to by exactly one variable. There is only one set of circumstances in which `transmitter`s do undergo a copy-on-write, namely when a `transmitter` is an attribute of an `object` and that `object` is experiencing a copy-on-write.

For collections

For collection type entities, whether they are immutable is determined by the entity itself rather than by the variables referencing it. Variables referencing collections therefore do not carry an immutability flag.

As was the case with variables referring to elementary type entities, multiple variables can reference the same underlying entity also when that entity is of a collection type. In the case of collections, however, changes to the attribute set do not cause a copy-on-write. If one variable is not immutable and the user creates, deallocates or modifies an entity's attributes through it, any other variable referencing the same entity will be impacted by these modifications.

Remark for the advanced reader 3.2.6. This is true even in the special case of a "`modifying()`" clause.

In fact, copy-on-write for collections only occurs under the following conditions.

- When a collection is labelled (See Section 3.10.1), and
- When an `object` is returned from `__init` and becomes a *module proxy* (See Section 3.14).

The actual rules regarding when a collection is immutable are complicated. In general, collections start off as non-immutable, and can be made immutable through *labelling*. However, there are specific circumstances in which even a labelled collection can be modified. See Section 3.10 for the full details.

[9] We have previously defined an entity to be "immutable" if it cannot be modified in place. For the purposes of that definition, we consider such a modification as is described here to be performed "in-place", regardless of the fact the entity first has to be copied out.

3.2.5 Referencing

Perhaps the most confusing part in introducing FTIL is its referencing system.

FTIL uses a three-tier referencing system for variables.

Identifiers refer to *variables* which refer to *entities*.

Zero or more identifiers may refer to a single variable, and one or more variables may refer to a single entity.

An *entity* is a carrier of information. In FTIL, all core data is ultimately stored in entities, and each entity's data is disjoint.

A *variable* is a way to access entity data. They reference entities and can be used to manipulate either the entities themselves or, for elementary entities, *elements* within these entities.

An *identifier* is a name for a variable. Any number of identifiers may refer to the same variable. Every identifier refers to exactly one variable.

Here are the basic facts regarding entities, variables, elements and identifiers. Much of the discussion here is expanded on in Section 3.8 and Section 3.9.

Entities

Entities were the topic of discussion throughout most of this introduction. They carry values. They have a type and a flavour that are both unchangeable throughout the entities' lifespans.

This is as opposed to variables, which refer to them, that are dynamically typed. FTIL supports a combination of static and dynamic typing, as is explained in Section 3.2.7.

An entity may be, just to name a few examples, an integer, or a list of ciphertexts, or an `object` containing attributes of a variety of types, or be of any of the other data types described in Section 3.5.

For elementary types, different variables referring to the same entity may have different scopes. The true scope of the entity in this case is the union of all scopes of the variables referring to it, because in all other nodes it would have been deallocated. However, nowhere in the system is this "true scope" stored. It is just the entity's effective scope.

For non-elementary types, all variables referring to them necessarily have the same scope. (For collections this is an invariant maintained by the language; for all other such types, the language only allows a single variable per entity.) For all such types, the entity therefore intrinsically knows, from its creation on, what its scope is, and this cannot change throughout its lifespan.

Remark for the advanced reader 3.2.7. It may be a good idea to retain this scope information for non-elementary entities, because it impacts the behaviour of attributes of these entities. However, because the entities are invariably accessed through variables, and because the variables do retain this information, this is not strictly necessary to do, so does not appear in this document as a requirement.

Variables

Variables are reference-counted references to entities: when no variables refer to an entity, that entity is deallocated.

Normally, each variable references exactly one entity. However, in some circumstances variables are uninitialised, in which case they do not refer to any entity. No variable can refer to more than one entity.

In referential-type entities, the attributes are themselves variables.

Like entities, variables, too, have metadata properties that describe them.

As already discussed, if `x` is a variable (and if its underlying entity is one that supports this, which the vast majority of FTIL types do) then "`x.metadata.scope`" provides the information of the variable's definition scope.

Note, however, that this is a special case: it is the only variable information available through "`.metadata`"; everything else stored under "`.metadata`" is a property of the entity, rather than of the variable.

The rest of the information carried by variables is not directly accessible to the FTIL user. It is, briefly, as follows:

- Which entity the variable refers to (if any).
- The variable's reference count (i.e., how many identifiers refer to it).
- Is the variable immutable? (This flag is only available for variables referring to distributed elementary entities or to `transmitter`s.)
- Is the variable protected? (This information is maintained in the form of a "protection counter".)
- Is the variable an attribute of an entity?
- And if so, is that entity's type one of `transmitter` or `catalogue`? If it is, this makes the variable a *dependent* variable. (See Section 3.2.7.)
- Dependent variables also retain the type of entity they can hold.

Remark for the advanced reader 3.2.8. In the case of variables that are dependants of a `transmitter`, the variable's scope indicates the scope of the `transmitter`, rather than of their own underlying entity. This is a unique exception. See Section 3.5.4 for details.

Here are a few examples, to demonstrate the difference between entities and variables. Consider the following FTIL code.

```
a = distribute(7)
b = object()
b.my_a = a
del a
a = b
b.my_a = a
```

- The first command, "`a = distribute(7)`", creates a new entity of elementary type `int`, at the maximal possible scope. This entity is distributed, and its value is 7 at every node. It is assigned to the variable "`a`", which is initially non-protected, non-immutable. As is usually the case at entity creation, there is at this time a one-to-one correspondence between entities and variables.
- The second command "`b = object()`" creates a new entity of referential type `object`, at the maximal possible scope. At creation, the `object` has no attributes. The entity is assigned to the variable "`b`". The one-to-one correspondence between entities and variables remains. The variable `b` is not protected.
- The line "`b.my_a = a`" creates a new attribute for the `object` entity. Because `object` is a referential type, `b.my_a` is a variable. This variable refers to the same entity as `a`. There are now two variables, `a` and `b.my_a`, both referring to the same entity, whose value is 7. Any property of the entity, such as the fact that it is distributed, is common, regardless of how the entity is accessed. Properties of the variables, however, may differ. One is, for example, an "attribute", and the other not. If either variable had now tried to modify its underlying entity's value (e.g., by performing "`a += 1`"), this would have triggered a copy-on-write which would have once again separated the underlying entities.

- The line "`del a`". Deletes the variable `a`. The underlying entity, whose value is 7, loses this reference. However, because it still has a remaining reference (from `b.my_a`) it is not deallocated and remains unchanged. Altogether, the entity switched hands from `a` to `b.my_a`, without any data copying.

- The line "`a = b`" creates a new variable, `a`, and assigns it to the same entity as `b`. This entity is of type `object`, and presently has only a single attribute, named `my_a`, of type `int`, distributed flavour, and a value of 7. The two variables, `a` and `b`, now refer to this same entity. The `int` entity remains a separate entity, because `object` is a referential type, so `b.my_a` is its own variable.

- The line "`b.my_a = a`" reassigns the variable `b.my_a` so that it now refers to the same entity as `a`. The `int` entity that was previously referenced to by `b.my_a` is now deallocated, because it has no remaining references. We are left with three variables, `a`, `b` and `b.my_a`, all referring to the same entity, of type `object`. This entity has a single attribute, called `my_a`, which is a variable that points to the entity itself, creating a circular reference. If one was to now delete `a` and `b`, garbage collection would have been needed to deallocate the `object` entity: it still has a remaining reference, but it is a circular reference. It cannot be reached because it is not connected to any top-level name (i.e., to an identifier). In contrast to the `int`, which was an elementary entity, an `object` is a referential entity, so no copy-on-write is triggered when it is modified.

Elements

The data of elementary entities is composed of elements. Any part of an elementary entity's data (i.e., not its metadata) that can be accessed individually is considered an element.

For example, if "`abc`" is of type `pair` (an elementary type), then "`abc.first`" and "`abc.second`" are both ways to refer to elements within `abc`. The pair `abc` in its entirety is also an element. (Every entity has as an element the "element containing all of the entity's data".)

Similarly, if "`def`" is of (elementary) type `list`, then "`def[3]`" is an element, and if "`ghi`" is of (elementary) type `dict`, with an appropriate key type, then "`ghi[CoordinatorID]`" is an element.

However, this distinction cannot be done based on syntax alone. The types matter. If "`abc`" is of type `object` (a referential type), then "`abc.first`" would be a variable, not an element. This would be the case also for "`def[3]`" if "`def`" is of (referential) type `catalogue`, and for "`ghi[CoordinatorID]`" if "`ghi`" is of (referential) type `transmitter`.

These distinctions are important, because they change the semantics of FTIL commands. Consider, for example, the following command.

```
a = abc.first
```

If `abc` is of an elementary type, then `abc.first` is an element. Because variables can only refer to entities, not to elements within entities, the value of `abc.first` is copied out into its own, separate entity, and this entity becomes the new value of `a`. Any previous value of `a` gets dereferenced, and if it is an entity that no longer has any references to it, it gets deallocated.

By contrast, if `abc` is of a referential type, then `abc.first` is an entity, and the variable identified by `a` (after discarding its previous value, as before) becomes a new variable referring to the same entity. No copying is done.

On the other hand, consider the following example:

```
abc.first = 7
```

Here, if `abc.first` is a variable, then this is simply an assignment to a variable.

Whatever entity `abc.first` originally referred to, this entity loses its reference. If this is the entity's last reference, it gets deallocated. Once that is done, the variable is assigned to an entity of type `int` and value 7.

If we were to now perform

```
abc.first = 8
```

this would have simply redirected the variable `abc.first` to another entity, namely one whose value is 8, and if the original entity had no further references to it, it would have been deallocated.

If, on the other hand, another variable had referenced the same 7, that variable's underlying entity would continue to be the 7. It would not have been affected by the fact that `abc.first` had been rerouted elsewhere.

Now, however, consider how this same code

```
abc.first = 7
```

works if `abc` is an elementary type.

If `abc` is an elementary type, `abc.first` is an element. Elements in FTIL have no autonomy: assignment to an element is merely a modification of its parent entity. Whether or not there is a copy-on-write at this point depends on whether there are any other variables referring to the same entity as `abc`. If there are none, this assignment is an in-place modification. Otherwise, `abc` is first copied out.

For a concrete example, consider the following.

```
a = distribute((3, 4))
abc = a
abc.first = 7
```

The first line initialises `a` as a distributed variable referring to an entity of type `pair<int, int>`. The value of `a.first` is at this time 3.

Next, we have an assignment that makes the variable `abc` refer to the same entity as `a`.

The last command wants to modify `abc.first` which belongs to the entity `abc`. However, this entity also has another variable referring to it (namely `a`), so the entire entity (the entire `pair`) is copied out to a new entity, `abc` is redirected to this new entity, and only then is `abc.first` modified (in-place) to the value 7.

At the end of the process, the system has two `pair<int, int>` entities, one, referred to by `a`, equals $(3, 4)$. The other, referred to by `abc`, equals $(7, 4)$.

Identifiers

An identifier is a top-level name for a variable. Many identifiers may refer to the same variable. Some variables will have no identifier. Every identifier refers to exactly one variable.

The way in which a variable can exist without a top-level name is that it may be known only by a non-top-level name. For example, in

```
x = object()
x.attr = 7
```

there is no top-level name that refers to the 7. The only way to reach it is through the top-level name "`x`". This is because this entity still has a reference, in the form of an attribute of another entity.

If a variable is not an attribute of an entity and is not referenced by any identifier, it is deallocated. Because variables may refer to referential entities which in turn have variable attributes, a cyclic referencing chain may occur. If an entire connected component of variable references does not connect to any identifier, that component needs to be detected and deallocated. The FTIL executor will have a garbage collection mechanism to deal with such cases.

Consider the following example:

```
x = object()
x.y = object()
x.y.z = x.y
del x
```

After the first 3 lines, `x.y` is a variable that can be accessed in an infinite number of ways: it can be accessed, for example, as `x.y`, as `x.y.z`, as `x.y.z.z`, etc.. However, it is not associated with any identifier, because it has no top-level name. It can only be accessed through the variable `x`.

Once the last line is executed, the variable `x` and its associated `object` are deallocated, after which only the circular reference remains, and it will be detected and deallocated by garbage collection.

Identifiers in FTIL can be defined on the command line or as part of a script. FTIL scripts may call each other, including recursively, and by this create an execution stack (i.e., a calling stack). Each execution stack position has its own identifiers, each identifier identifying a variable.

Thus, unlike entities and variables, identifiers have a fixed life-span that is determined at the moment of their creation: they are deallocated at latest when the script that defined them finishes.[10]

In all examples up until now, there was a one-to-one correspondence between top-level variables and identifiers. The importance of identifiers is mainly in parameter passing, which is where this correspondence breaks.

Consider the following code, which defines an FTIL script called `MyCopy` and then invokes it.

```
def MyCopy(src, dest):
  dest = src


abc = 7
MyCopy(abc, xyz)
```

Here, at the top level of the execution stack, only two identifiers are defined: `abc` and `xyz`. Once the script `MyCopy` is called, however, each of the variables referred to by these identifiers gets another name, this being another identifier.

Both "`abc`" and "`src`" now refer to the same variable (whose underlying entity is an `int` whose value is 7), and both "`xyz`" and "`dest`" now refer to the same variable.

The command "`dest = src`" provides the entity referred to by "`abc`" and "`src`" (i.e., the 7) with another reference, this being the variable referred to by "`xyz`" and "`dest`". If the variable identified by "`xyz`" and "`dest`" previously referenced an entity, that entity gets dereferenced, and if its reference count is down to zero, it is also deallocated. (Note, however, that there is no reason for "`xyz`" and "`dest`" to have previously referred to any entity. The line "`MyCopy(abc, xyz)`" may have been the first appearance of "`xyz`", making it an uninitialised variable, not referring to any entity.)

After the assignment, we have one entity (the "7") referred to by two variables, each of which is identified by two identifiers.

After returning from the script, however, the identifiers `src` and `dest` get deallocated, so we are down to one entity referred to by two variables, each of which is identified by only a single identifier.

In the above example, multiple identifiers identified a single variable, but for each variable only one such identifier was usable to the FTIL program at any given time: within `MyCopy`, only `src` and `dest` were addressable, whereas outside `MyCopy`, only `abc` and `xyz` were addressable. This is, however, not always the case. There are examples where a single variable has multiple identifiers identifying it, all addressable simultaneously.

The code below gives an example.

```
MyCopy(abc, abc)
```

Here, inside `MyCopy`, prior to the assignment, there are three identifiers (`abc`, `src` and `dest`) all identifying the same variable, of which two (`src` and `dest`) are simultaneously addressable.

The line "`dest = src`" now assigns a variable to itself.

[10] Advanced note: This deallocation may be done with some clean-up, if the script aborts.

Note 3.2.2. In implementing the FTIL executor, it is important to handle such assignments gracefully, avoiding a situation where the variable of `dest` first gets dereferenced and deallocated, and by this loses the value of `src`, which is to be assigned to it.

This fine point is relevant even without a script call, and, in fact, even without self-assignment. It is equally relevant for handling

```
i = i + 1
```

Generally, the way to handle this is to first assign the right-hand side result to a temporary, nameless variable, and only then deallocate the left-hand side and assign the temporary variable to it, after which the temporary variable can safely be deallocated. (As always in FTIL, such assignment is done by redirecting the relevant variables, and requires no copying of any of the underlying entity's actual data.)

3.2.6 Qualified names

A *qualified name* is a particular way to access a variable. It may be an identifier, like "`a`", or it may be a sequence of attributes starting from an identifier, like "`a.b.c[7].d`".

In this example, the dot notation indicates that these are `object` attributes, whereas the bracket notation indicates `catalogue` attributes. So, `a`, `a.b` and `a.b.c[7]` are all `objects`, `a.b.c` is a `catalogue`, and `a.b.c[7].d` may be an entity of any type. It is important, however, that it is an entity and not an element, because qualified names only relate to accessing variables, not elements.

Some functionality in FTIL depends on maintaining lists of qualified names.

Importantly, when we determine what a particular qualified name is, or when we store it, we always store for `catalogue` and `transmitter` attributes (the attributes referenced by brackets) the actual integer value of the attribute used, not how it was accessed (which may be through indirect addressing by referring to a variable or a computation).

In other words, if the FTIL code refers to any one of "`a.b.c[i].d`" or "`a.b.c[i - 17].d`" or "`a.b.c[3 + 4].d`", FTIL must first resolve which variable is "`a.b.c`", then what the value of "`i`" or "`i - 17`" or "`3 + 4`" is, and only then plug that value, say 7, in, to determine what is "`a.b.c[7]`" and ultimately "`a.b.c[7].d`". When we consider the "qualified name" used in the FTIL code, we always mean the fully resolved "`a.b.c[7].d`".

There are several uses for qualified name lists in FTIL. These are discussed in Section 3.9.3. In this section, we introduce some terminology regarding qualified names. For this, we will take "`a.b.c[7].d`" as our running example.

First, note that while in FTIL one uses the same syntax to refer to elements as to entities (e.g., "`a.first`" is an element if `a` is a `pair`, but is an entity if `a` is an `object`, and "`a[7]`" is an element if `a` is a `list`, but is an entity if `a` is a `catalogue`), the term "qualified name" refers only to names that refer to an entity, not to ones that refer to an element.

The *prefixes* of the qualified name are "`a.b.c[7]`", "`a.b.c`, "`a.b`" and "`a`".

The *extended prefixes* of the qualified name are its prefixes plus itself, i.e. extended to include also "`a.b.c[7].d`".

The *underlying variable* of the qualified name is the variable that "`a.b.c[7].d`" refers to, and the *underlying entity* is the entity that this variable refers to (if any). Note that by definition of "qualified name", all underlying entities of every one of any qualified name's prefixes must be of a referential type. In fact, every one except for the longest prefix must be of a collection type (either an `object` or a `catalogue`). The longest prefix may also be of a `transmitter` type.

The *antecedents* of a qualified name are the underlying variables of its prefixes.

The *root* of a qualified name is the identifier that starts it (which in our running example is "`a`"). Its *derivation* is what comes after the root. In our example, this is "`.b.c[7].d`". We say that a qualified name is *atomic* if it is solely a root, without a derivation.

Note that because a qualified name starts with an identifier, it is inherently tied to a particular execution stack level. A qualified name ceases to exist when its script terminates and that execution stack level is "popped".

Also note that the same variable can have infinitely many qualified names, and even infinitely many qualified names beginning with the same identifier. Consider, for example, the following code.

```
a = object()
a.next = a
```

In this example, there are infinitely many qualified names that the system recognises. They include "`a`", "`a.next`", "`a.next.next`", etc.. These names all begin with the same identifier and all reference the same underlying entity. There are two variables in the system. One is an attribute of an `object` that references the object itself, and one is not an attribute, and is identified by the identifier "`a`". Thus, the qualified name "`a`" refers to one variable (the non-attribute one), but all other qualified names in the system ("`a.next`", "`a.next.next`", etc.) refer to the same other variable.

3.2.7 Dynamic and static typing

In strongly-typed computer languages, an assignment of a variable x, of some type X, into another variable, y, of some type Y, will result in y receiving the value of x, but not always the type of x. If y retains its type and the value of x is in the process converted to type Y, we say the language is statically typed. If, on the other hand, y ends up as type X, without the need for any value conversion, we say that the language is dynamically typed.

FTIL uses a combination of dynamic and static typing.

First, recall that in FTIL y may not be a variable but rather an element. When that is the case, the type of y is determined by its parent entity. On assignment, the value will be converted to the required type if a valid conversion path exists, or the command will fail. This is an instance of static typing.

In the MVP of the Purgles Platform, we do not expect FTIL to support almost any implicit conversions, so having a "valid conversion path" will almost always mean that the types X and Y are the same.

Another FTIL case of static typing is when y is a *dependent* variable: if y is an attribute of a `transmitter` or of a `catalogue`, the parent entity determines the properties of any attribute to be created under it, so static typing is used. Specifically, the attributes of a `transmitter<T>` are of distributed, serialisable, elementary type `T`, while `catalogue` attributes are of type `object` and have the same scope as the `catalogue` itself. (See Section 3.5.4 for a discussion of the scope of `transmitter` attributes.)

Otherwise, FTIL uses dynamic typing.

For a non-immutable, independent, distributed, elementary variable y, FTIL also supports the syntax "`y[]`" to indicate that we are not considering the variable y itself, but rather the element containing all of y's data, in the nodes in which the command is executed.

In such a case, the command "`y[] = x`" is a useful way to force statically-typed assignment from x to y. It can be used whenever statically-typed assignment to an element of y is allowed. For example, it can be used on only a subset of the scope of y, which is a case where dynamic assignment would have failed altogether. (We sometimes refer to this type of assignment as "sub-assignment".)

Remark for the advanced reader 3.2.9. The rest of this section details the nitty-gritty technicalities of static and dynamic assignment, at a level only suitable for a reader already familiar with FTIL. Readers new to FTIL should probably return to it after gaining familiarity with the language as a whole.

Static typing in FTIL means not only that the destination variable's original type remains, but also all its other entity and variable metadata properties.

If a static assignment command is legal, it is essentially executed in parallel on each node in the command's scope, independently.

If y is an element, the assignment updates the value of an existing entity. In this case, the semantics of the command are a modification of the element inside its entity. If the entity has only a single reference, this modification is done in place. Otherwise, the entity is first copied out.

If y is an attribute of a `transmitter` or a `catalogue`, the assignment is still in parallel, but is to a variable. Because this is a static assignment, we first convert the value of x to the necessary target type. If x is not already of the target type (or is an element rather than an entity), this creates a new entity with the converted value. If x is of the target type, x itself is assigned, and there is no data copying. At this point, if the variable y exists, then any entity it may refer to now loses its reference, and the variable is redirected to refer to the new value. If the variable y does not exist (i.e., it is an attribute of a `transmitter` or a `catalogue` that has never been referenced before and never been created), then the assignment itself now creates it.

The command must be one that allows this type of assignment:

- Its scope must be a subset of the scopes of both what is being assigned and the parent entity of y.
- If y is an attribute of a `catalogue`, the scope of the assignment must exactly equal the scope of both x and the `catalogue`.
- The name "y" must be a legal attribute name for its parent entity.
- If y is a a variable, y must be non-protected. If it does not currently exist as an attribute, the underlying variable of its longest prefix must be non-immutable (in the meaning of "non-immutable" relevant to that variable type).
- If y is an element, y's parent variable must be non-immutable, and that variable's underlying entity must be distributed.

If, however, x is assigned to y and y is an independent variable, then this is a dynamically-typed assignment. The variable y receives not only the value of x but also all other properties of its entity. For this to work

- The variable for y, if it exists, must be non-protected.
- If y exists, the scope of the command must be identical to the scope of y.
- The scope of the command must be a subset of the scope of x.
- If x is a collection, the scope of the command must be identical to the scope of x.

Where all these checks pass, the semantics of dynamic assignment are that the variable y is redirected to refer to the same entity as x. If x is not an entity at all but rather an element, an appropriate entity with the same value and properties as x is first generated. Otherwise, no data copying is done. If the original entity that y referred to no longer has any remaining references, it is deallocated. (Note, however, that the number of references may differ from node to node.)

A third possibility, which also leads to dynamic assignment, is when y is an attribute of a parent variable of type `object`, but the attribute is not defined prior to the assignment. In this case

- The parent `object` must be non-immutable (See Section 3.10 for how this is defined for `objects`.)
- The scope of the command must be identical to the scope of the `object` and a subset of the scope of x.
- If x is a collection, the scope of the command must be identical to the scope of x.

3.2.8 Quick cheat sheet

Figure 3.1 attempts to capture the highlights of this section. Some conditions have been omitted for clarity.

Figure 3.1: Example FTIL case handling

If v is...	this syntax...	means...
int	v =	Variable v now refers to the RHS.
	v[] =	If v is distributed and non-immutable: convert RHS to the same type as v (creating a temporary copy if the RHS was not originally of the target type), then make v refer to the result. If no legal conversion is found, the command fails.
rm	v =	As in int.
	v[] =	As in int. Assuming v is distributed and non-immutable, the RHS is converted to array<int, 7> (or fails).
	v[3] =	Position v[3] in v is modified. No change to type, except that the RHS is converted to int. (The command fails if that is not possible.) If another (top-)variable refers to the same entity as v, all of v is first copied to a separate place. Conversion and copy aside, the assignment may be performed in place.
rm	v =	As in int.
	v[] =	As in int.
	v.first =	Position v.first in v is modified. Analogous to example "v[3] =" in "array<int, 7>", above.
rm	v =	As in int.
	v[] =	As in int.
	v[3] =	As in array. If v is shorter than 4 elements, the command fails.
rm	v =	As in int.
	v[] =	Not supported.
	v[x] =	Here, x must be a nodeid in v's scope. If v previously did not have the key x, it is created. Otherwise, what v[x] previously referred to loses the reference (and if it has no more references, gets deallocated). The RHS is converted (if possible) to type list<int>, distributed, copying it if necessary, and then v[x] is made a reference to the result.
	v[x][3] =	If there is another variable referencing v[x], all of v[x] is copied out. The RHS is converted to an int (if possible), and is used to modify place v[x][3] of v[x].
rm	v =	As in int.
	v[] =	Not a legal command for type rm.
	v.x =	The reference v.x is updated.
	v.x[] =	If v.x is distributed (a single rm instance can have both distributed and shared attributes) and non-immutable, the RHS is converted to the type of v.x, and (if the conversion is successful) v.x is changed to refer to its new value.

3.3 The FTIL standard library

Unlike Python, which allows language extensions to be compiled into the language by advanced users, albeit not from within Python itself, and unlike C++, which allows users to define new types that are all but indistinguishable from native types even from within C++ itself, FTIL is a minimalist language that does not provide the user with tools to extend the language itself.

The consequence of this is that, to be effective, FTIL must provide out-of-the-box capabilities that make it suitable for its tasks.

We have therefore divided FTIL conceptually into two tiers.

On the one hand, there is the core language, providing basic, Turing-complete capabilities for distributed processing. All properties of FTIL that have been discussed so far describe this core language layer.

On the other hand, on top of the core language there is a layer that deals with everything to do with privacy-preserving protocols, cryptography, and a specialisation for the financial domain. Though from the perspective of the FTIL user there is no way to tell the two apart, we envision the second layer to be provided by means of what is effectively a "standard library" for FTIL.

Our hope is that future extensions to the language will mainly require extending this standard library, and will only rarely require changes to core FTIL features.

3.3.1 A note on capitalisation

Though the division between core FTIL and the FTIL standard library is ostensibly invisible to the FTIL user, we do differentiate between the two in the capitalisation scheme we have opted for, for each.

All base language keywords and all basic language types are in lowercase, as well as all core built-in commands and the methods of such core built-in types.

Standard library types and commands, on the other hand, are in UpperCamelCase, as are the system constants.

Attributes of standard library types, will be in lowercase with underscores to separate words.

Later on, in Chapter 4, we will write also user code in FTIL. There, we will use the following convention.

UpperCamelCase will be used for script and method names.

Lowercase will be used for user-defined variables, for member attributes of `object`s and for parameters of scripts, with underscores to separate words.

For labels (regarding which see Section 3.10) we use space-separated, capitalised words.

3.4 FTIL variable names

In FTIL one can define, manipulate and delete variables.

In terms of naming, variables obey the following rules:

1. Names of scripts are composed of alphanumeric and underscore characters, but cannot begin with an underscore or a digit. (An exception is the special script named "`__init`", described in Section 3.14.)

2. Names of other top-level variables are composed of alphanumeric, underscore and colon characters. However, they cannot begin with a digit, they cannot begin or end with a colon, and use of colons carries special meaning, as described in Section 3.13.6.

Variables of type `object` may have attributes that are either members (attributes carrying data) or methods (attributes carrying functionality). They are referenced by the `object` name followed by a "`.`", followed by the attribute name. For attributes, the following rules apply:

1. Names of methods follow the same rule as scripts.

2. Names of member attributes follow the same rule as non-script top-level variables, except that they are not allowed to use colons.

Note that scripts are always top-level variables and methods are always attributes.

No variable of any type may have the same name as a reserved word (such as a language construct or a built-in type or function name, whether part of the FTIL standard library or not), nor can a variable share the same name as a system constant.

Within scripts (See Section 3.13), script-local variables have the additional restriction that their names are not allowed to contain colons. Script formal parameters follow the same rule as local variables, but with the added restriction that their names may not start with underscores.

In our examples in Chapter 4, all script-local variable names begin with an underscore, to differentiate them from the script's formal parameters. This is in no way enforced by the language and is merely a naming convention.

3.5 FTIL types

FTIL types broadly divide into five categories:

1. *Fixed-sized* types (Discussed in Section 3.5.1),
2. *Container* types (Discussed in Section 3.5.2),
3. *Collection* types (Discussed in Section 3.5.3),
4. *Special* types (Discussed in Section 3.5.4), and
5. *Pseudotypes* (Discussed in Section 3.5.5).

The fixed-sized and container types are collectively known as *elementary* types. This is because the data contents of entities of these types are data elements. This is as opposed to *referential* types. Entities of referential types have attributes that are themselves variables, referring to other entities. All collections are referential types, as well as the special type `transmitter`.

3.5.1 Fixed-sized types

Fixed-sized types are the most basic building-block of the language. They include

fixed-size integers: ("`int`") Perform integer arithmetic. We only need one size, say 64 bit signed integers.

floating-point numbers: ("`float`") Perform floating-point arithmetic. For future machine learning applications, we may need to support both "float"-sized and "double"-sized values. At the moment, a single type of floating point value should suffice.

Booleans: ("`bool`") Perform Boolean arithmetic. These are needed for the evaluation of conditionals. The literals `True` and `False` are of this type. FTIL does not perform short-circuit Boolean evaluation.

Node Identifiers: ("`nodeid`") Designates the ID of a Segment Manager.

See Section 3.11.15 for notes regarding operations allowed on numerical and Boolean types.

A `nodeid` should support conversions to and from `int`. What node is mapped to which `int` is arbitrary, but converting from an `int` to a `nodeid` for an `int` that does not have a `nodeid` associated with it is an error.

Beyond such atomic fixed-sized types, FTIL also supports composite fixed-sized types. For example, FTIL supports a fixed-size template type, `rm`. This is an array of `k` values, each of a type `T` that is of a fixed size.

Note 3.5.1. Arrays were part of FTIL from the first draft of this document. Their purpose was to facilitate multidimensional data exploration and linear algebra, as well as to avoid the need for an explosion of per-algorithm data types. Despite some surface similarities, they are very distinct from container types like `list` and `memblock` introduced in Section 3.5.2. However, as of this moment, none of our example algorithms presently requires `arrays`, and in general all added value from the use of `arrays` can be gotten—in less convenient and more roundabout ways—also without them. With this in mind, and in view of the significant complexity that `arrays` introduce

into the design, we have decided to reduce `arrays` to Priority 2, though we have kept discussion of them as part of this document.

Readers considering only Priority 1 items can safely ignore all parts of this document discussing `arrays`, and there are many. Note, however, that `arrayiter`, discussed in Section 3.12.2, is still Priority 1.

Arrays support element and contiguous-slice access for read and write, but cannot insert, append or delete elements. They can be initialised from an explicit list (bracketed and comma separated) or from a single element (populating the entire array).

Note that because `arrays` are themselves fixed-sized types, `arrays` of `arrays` are allowed. These can be initialised from an explicit nested list or from from a single element (populating the entire multi-dimensional array).

Entities in FTIL are always initialised at their construction. Explicit lists that are used to initialise arrays, including nested ones, must be of exactly the correct length, or initialisation fails.

If `v` is of an `array` type, "`v.size()`" is the length of `v` and is a value visible both within `v`'s scope and by the Compute Manager (as part of the type information within the Variable Registry). This feature is meant to make the interface of `array` more compatible with that of `list` and `memblock`, introduced in Section 3.5.2. However, because `arrays` are fixed-sized types, this method is not sufficient to fully explore the dimensionality of `arrays`. For example, it does not help with multidimensional `arrays`, with `arrays` used as dictionary keys, etc.. In order to determine these programmatically, given a variable, one should use FTIL's introspection capabilities, introduced in Section 3.5.5.

Arrays also support somewhat more exotic functions like finding the minimum, the maximum, and the index of the first element that is/is not the given parameter. (Priority 3)

Additionally, FTIL provides multiple task-specific types. These include

`rm:` An account number.

`rm:` A BSB number.

Both `AccNum` and `BSBNum` atomic types support conversions to and from `int`. The language is free to decide when an `int` is not a legal source for either of these, in an implementation-dependent way, but any `int` that originated from one must always be convertible back to the same value as the original.

Some of the language's "standard library" fixed-sized types behave as "struct" definitions over existing types. This means that they are composite types, composed of multiple sub-elements, each with its own name and its own (fixed-sized) type. We refer to these as the struct's *attributes*, but note that they are distinct from the entity's metadata attributes. It is not possible for the FTIL user to create such new types themselves.

Examples of predefined structs are

`rm:` A struct concatenating a `BSBNum` (Attribute name: "`bsb_num`") and an `AccNum` (Attribute name: "`acc_num`"). Used to specify a particular account.

`rm:` A struct concatenating a `nodeid` ("`node_id`") and an `AccountID` ("`account_id`"). Used to specify a particular account in conjunction with the RE managing it.

As can be seen in the `AccountAddr` example, structs can be nested.

All structs in the language support explicit constructors that populate all their attributes from constructor parameters.

Additionally, there is a built-in template type "`pair<A, B>`", where "`A`" and "`B`" can be any fixed-sized types and are referenced by the attribute names "`first`" and "`second`", respectively. The syntax "`(a, b)`" constructs a pair, `pair<A, B>`, where `A` is the type of `a` and `B` is the type of `b`. In the created pair, `a` is assigned to the attribute `first` and `b` to `second`.

FTIL has built-in support for multiple cryptographic primitives, including symmetric encryption, public key encryption, one-way hashing, one-way keyed hashing, digital signing and semi-homomorphic encryption. For each of these, FTIL will support at least one specific algorithm. The specific choice of algorithms is at this time TBD.

Note 3.5.2. In future, we may also want to support fully-homomorphic encryption. This is currently Priority 3, but is subject to change with developments in the FinTracer algorithm suite, and will increase to Priority 1 if post-quantum security is required.

Each algorithm supported will require its own, specific types. For example, suppose we wish to support RSA with some chosen parameter set as an FTIL public-key encryption scheme. This will certainly require defining task-specific types for the RSA private key, and quite possible also for the RSA public key, for the RSA plaintext and for the RSA ciphertext. The RSA plaintext will support conversions to and from generic types such as the integer type.

FTIL has special built-in security features for the handling of private keys. Specifically, private key types, regardless of encryption algorithm, will not support commands that allow their values to be copied to any node other than their original one. They will not be convertible to other types, will not be eligible for "`broadcast`", "`transmit`", "`display`" or "`export`", and neither will any containers containing them as either keys or values. (When we later define the "serialisable" types, private keys and containers of private keys will be explicitly excluded.)

With homomorphic encryption the situation is similar, except that here we have already decided on the use of additive ElGamal over the Ed25519 group as our choice of algorithm, and know of specific types and specific functions that this now requires.

Specifically, to support ElGamal over the Ed25519 group, we require the following types.

`rm:` An ElGamal private key,

`rm:` An integer modulo the Ed25519 group size,

`rm:` An Ed25519 group member, being an ElGamal plaintext,

`rm:` An ElGamal ciphertext (in projective representation),

`rm:` An ElGamal compressed ciphertext in affine representation (Priority 3), and

`rm:` An ElGamal compressed ciphertext in folded representation (Priority 3).

The type `ElGamalCipher` is a struct containing two attributes, "`mask`" and "`masked_message`", both of type `Ed25519`.

The ElGamal public key does not require a special type. It can simply be of type `Ed25519`.

The same is not true for the ElGamal private key, however. Like any private key, it must have its own type, and the type will need to be flagged as a private key type, so that it cannot be transmitted.

At the moment, none of our algorithms rely on the ability to compress ElGamal ciphers, for which reason all such capability is presently listed as Priority 3.

Note 3.5.3. An `Ed25519Int` can be constructed from an `int`, but clearly that only covers a very small part of the range possible to an `Ed25519Int`.

An `Ed25519` can also be constructed directly from an `int`, and also with the same caveats. To construct a general member of the Ed25519 group, it can be constructed from an `Ed25519Int`.

The functions that the ElGamal types support include

`rm:` Create two new distributed variables, such that each Segment Manager that is in scope generates its own private-public key pair.

`rm:` Creates a new private-public key pair as shared variables. This is done via a secure protocol that ensures no subset of the parties can control key selection.

`rm:` All FTIL key types can be constructed from a seed value. The idea is that this construction is deterministic (in the sense that the same seed always leads to the same key) and as much as possible entropy-preserving. In the case of ElGamal encryption, the seed type will be an `Ed25519Int`, which has a loss-less, one-to-one conversion to an ElGamal key pair, simply by

taking the seed to be the private key. (As demonstrated here, the key does not need to be some one-way hash of the seed. A direct conversion, which will generally be fastest, works just as well.) For more on the proper use of key seeding, see Section 11.3.10.

Integer arithmetic and integer conversions: On the integer type.

Internal conversions: From the integer type to the group member. (Reverse conversions will be done using comparisons to specific values or using a hashmap; see Section 3.5.2.)

Comparison: All types including `rm` can be compared for equality.

`rm`: Creates a ciphertext from a plaintext and a public key.

`rm`: Creates a plaintext from a ciphertext and a private key.

Summation: Creates a new ciphertext by summing two together. Can also be performed in-place.

Affine summation: Sums together two affine-compressed ciphertexts. (Priority 3)

Folded summation: Sums together two folded-compressed ciphertexts. (Priority 3)

Compression and decompression: Functions for compressing from projective representation to affine and folded, and functions for decompressing back to projective representation. (Priority 3)

Product: Multiply a (projective) ciphertext by a group integer.

Sanitise: Multiply a (projective) ciphertext by a random non-zero group element.

Refresh: Add to a (projective) ciphertext a newly-encrypted zero.

In Section 3.5.2 we introduce iterable containers in FTIL, and in Section 3.12 we detail how such iterables allow efficient execution of (among other things) per-element data-parallel commands on each member of the container separately. For most of the operations listed above, this mechanism is enough to support efficient execution over entire containers of ciphertexts. However, for `Sanitise` and `Refresh` specifically, we believe that the language should provide built-in mechanisms to run these on an entire iterable together. For more on this for refreshing, see Section 3.11.13. For sanitisation, the key property required is that the random non-zero group element chosen is chosen independently for each container element.

Having said this, all operations listed above except the key generation functions are eligible for use in the mass operations syntax described in Section 3.12.

For encrypting and decrypting, working on an entire iterable is largely syntactic sugaring.

Note 3.5.4. FTIL should also support at least one other type of semi-homomorphic encryption (potentially a multiplicative ElGamal algorithm), because additive ElGamal does not allow use of the full plaintext length when decrypting. Though none of our present algorithms require this, we are constantly evaluating other algorithms that do. For example: Multi-Party Computation.

Supporting the other cryptographic primitives mentioned (symmetric encryption, one way hashing, one way keyed-hashing and digital signing) is quite similar, with the main difference being that some or all of these primitives, in FTIL, may not need their own "plaintext" types. Rather, they may be able to act on any plaintext type, `T`.

Specifically in the case of encryption, if the plaintext can be of any type `T`, then the ciphertext type will be a template type such as `rm`, carrying the original `T` that was used as an input type. The reason to keep the information of `T` in the ciphertext type is for decryption: the function decrypting a variable of type `Cipher<T>` will reinterpret the result as a variable of type `T` before returning it.

Note, in particular, that in this case one may encrypt a ciphertext and reach a variable of type `Cipher<Cipher<T>>`.

Hashes should typically be able to work on any plaintext type (except private key types), but do not retain the original type `T` as part of their type definitions.

Where encryption, decryption, hashing, etc. are applied on a fixed-sized type, these actions are eligible for use in mass operations.

3.5.2 Container types

FTIL supports the following template types that can be constructed over any fixed-sized type, whether basic or templatised.

rm: A variable-sized version of the array. May contain any fixed-sized type, but all its elements must be of the same type. Can be constructed from an array or constructed directly from similar inputs as for array construction. A `list<T>` can also be constructed from a `set<T>`, in which case the elements of the `set` are placed in the `list` in an arbitrary sorting order. By "arbitrary sorting order", we mean that the items are sorted, and by this we ensure that no information other than the membership of elements is propagated from the original `set` to the `list`, but the choice of sorting order used is arbitrary and implementation dependent. The only restrictions are that the output `list` must be a direct function of the set of elements in the `set`, and that this function (the arbitrary sorting order) must be the same at all nodes.[11]

rm: Identical to a `list<T>` (or an `array<T, size>`), except that its size is set at construction time and cannot be altered afterwards. This type adds no functionality that cannot be attained by lists, but plays a critical role in implementing fast algorithms. See Section 3.12.3.

rm: A mapping from key-type to value-type. The keys of the dictionary are effectively immutable: they cannot be directly modified, though they can be removed and replaced. Both key-type and value-type must be fixed-sized. Can be constructed from a `list` (or similar) of `pair`s or from a `pair` of `list`s (or similar). Provides both a standard look-up ("`d[k]`") that fails if `k` is not a key of `d` and a safe look-up ("`d.lookup(k, default)`") that returns the default value if `k` is not a key. Also provides mechanisms to check if a given value is a key of the `dict`, similar to a "`set<T>`" (regarding which, see below). Supports a method "`.keys()`" that returns the mapping's keys as a `list<KeyType>`, where the keys are in an "arbitrary sorting order", as defined above. Both types of look-ups are eligible for mass operations (See Section 3.12).

rm: A set of values of type `T`. Similar to a list, but does not allow direct element access and keeps its values unique. Supports union ("`union`"), intersection ("`intersect`") and set difference ("`setminus`"), all of which it supports both as functions generating a new set and as methods modifying a set.[12] It also supports tests for membership. Items in a set are effectively immutable: they cannot be modified directly, though they can be removed and replaced. A `set` can be initialised from a `list`, `array` or `memblock`. Arguments to `union`, `intersect` and `setminus` can be either `set<T>` or `T`. In the latter case, the argument is treated as a singleton set. The result type is `set<T>` in all cases.

rm: Similar in functionality to a dictionary but performs its mapping via a hash table. Supporting this type is a Priority 3 requirement.[13]

Previous versions of this document defined a special variable type "`rm`". Currently, `nodeset` is simply an alias for `set<nodeid>`. The two are the same type. We still use the name "`nodeset`" in

[11] Despite the many references to `rm`s, support of `rm`s is a Priority 1 requirement. They do not depend on the existence of `rm`s.

[12] Using terminology introduced in Section 3.11: though these operations are supported as functions, they are not "proper functions".

[13] There are several technologies for managing maps, and we may want to support others, too, such as "trie"s, for efficient handling of various algorithms, but for the moment only the basic map ("`rm`") is Priority 1 and the rest are all Priority 3.

the document, because this type has special roles in the language (e.g., as a parameter to "`on()`"), but it has no functionality that one would not expect equally from a `set<nodeid>`.

The types `array`, `list`, `memblock`, `dict` and `set` (but not `hashmap`) are *iterable*. See Section 3.12 for a discussion. All iterables support checks for equality (e.g., whether one list equals another list), although such a check can, understandably, be quite heavy.

Note that FTIL is quite restrictive in its container types, allowing only fixed-sized types in them. Lists of lists, dictionaries of lists, etc., are not allowed by the language. The `array` type is, however, a fixed-sized type, not a container, so `array`s of `array`s, `list`s of `array`s, and even `list`s of `array`s of `array`s are all allowed, and can be used to create data structures of arbitrary dimension.

Note also that the type of a container is set at its creation (much like its scope), so even if a `list` or `dict` is empty, it still retains its template type parameters.

The types `list` and `memblock` support element and slice access, as well as the method `rm` for determining the number of elements in the list. The type `dict` supports the `rm` method to determine the number of keys in the dictionary, and for `set`s this is the number of elements in the set. The type `list` supports `append`, `insert` and `del`, but `memblock` does not.

To unify some terminology: when referring to the "keys" of a `list` or `memblock`, we mean the integer range of its valid indices. The items stored in a `list` or `memblock` will sometimes be referred to as the entity's "values".

We refer to fixed-sized types and iterable types, collectively, as *serialisable* types, but exclude from this category any type that is a private key or a container of private keys. Programmatic FTIL communication between nodes is normally done using either "`broadcast`" or "`transmit`". Both commands, discussed in Section 3.11.6, work only on serialisable types (so, in particular, not on private keys or containers thereof). Under the hood, the FTIL executor needs to communicate non-serialisable types as well. The mechanics of such communication were discussed in Chapter 2. The specific mechanics of `string` communication are discussed in Section 3.5.4.

3.5.3 Collection types

Collections are a category of FTIL types that includes `object`s and `catalogue`s. Unlike elementary types, collections are referential: they act like small-scale variable registries, where each attribute is itself a variable.

When a collection is deleted, all the variables within their mini-registries are deallocated as well. If the entities these variables referred to no longer have any remaining references, they are deallocated as well, recursively.

Collections follow different rules to elementary types. Though they have a scope, their attributes have scopes of their own which may be equal to or narrower than (but not wider than) the scope of the collection. This is with the exception of attributes that are themselves of collection type, whose scope must equal exactly the scope of their parent collection.

Collections can only be assigned using "`=`" assignment. When "`a = b`" is executed, the variable `a` becomes another variable to reference the same entity as variable `b`. The entity, in this case, is the mini-registry, not the entities the mini-registry points to. So, for example, a change to `b.attr` will immediately be reflected in `a.attr`. This is in contrast to the copy-on-write policy employed by elementary types.

As explained in Section 3.2.7, such assignments have multiple prerequisites. In particular, if `b` is of a collection type then "`a = b`" can only be executed in exactly the same scope as the scope of `b`.

The assignment itself does not copy any information. It just creates in `a` a variable that references the same collection (the same "mini-registry") as `b`, or redirects such a variable if it already exists. This is a shallow copy. The language does not have any built-in mechanism for deep-copying collections. If one wishes to deep copy a collection, one must program this functionality in FTIL.

In keeping with its status as a "mini-registry", how collection attributes (which are names of variables) are managed in FTIL is essentially identical to how identifiers (which are also names of

variables, specifically top-level variables) are managed. In both cases a new variable with a new (or re-defined) name can be created by assignment, the underlying entity can be manipulated and modified by the same syntax and semantics, and ultimately a `del` command can be used to delete the variable. In both cases, any redirection or deletion of the variable must be done in the same scope as the scope of the variable itself.

When creating new variables, too, their scope is set as the scope of the command that created them. In the case of collection attributes, this scope merely has additional restrictions: it must be a subset of the scope of the collection, and if the underlying entity is itself a collection, then it must be created with the same scope as the parent collection.

All collections support the equality comparison operator "`==`". This operator checks for two collection-type variables whether they reference the same entity. It is not enough for them to be of the same entity type and to reference entities with identical values. The negation of this operator, "`!=`", is also supported.

For example:

```
a = object()  # Creates an empty object.
b = object()  # Creates another empty object.

# At this point, a != b holds.

a = b

# Now, a == b holds.
```

Collections can be *labelled*, a process explained in Section 3.10. Labelling changes both the entity's metadata and the variable pointing to it, and is one of the only cases where a referential entity may experience copy-on-write. However, this copy-on-write, too, is only required if there are multiple variables referring to the same entity. Otherwise, labelling is done in place.

(The only other copy-on-write case for collections is when a variable is returned from "`__import`", but such a variable is necessarily of type `object`. The process is described in Section 3.14.)

Copy-on-write for a collection means that a new entity with the appropriate properties is created, all of its attributes are copied from the original collection in the sense that they are new variables that refer to the same entities, and the variable referencing the collection is redirected to the new entity. (However, see Section 3.5.4 for how `transmitter`-type attributes are handled in this case.)

When a collection is labelled, a new attribute of type shared string is added to its metadata, as "`.metadata.label`".

Note 3.5.5. Strings are discussed in Section 3.5.4. Shared strings are discussed in Section 3.5.5. They are distinct types.

In addition to the label itself, a labelled collection retains the following metadata properties.

- Whether the label is *strong* or *weak*, and
- Whether the label is *suspended*.

These properties are used by the Compute Manager to determine the legality of operations, but are not accessible programmatically through the `metadata` property.

The label is strong if labelling was done in privileged mode (See Section 3.15) and is weak otherwise. This information can be different from node to node, but does not change after it was set at labelling.

The label's initial status is "Not suspended". See Section 3.10.3 for how this can change later on.

Once a collection has been labelled, no change can be made to the collection's "mini-registry": no new attributes can be defined, existing attributes cannot be deleted, and all existing attributes

are, themselves, marked as protected and, where relevant (i.e., for distributed elementary-type and `transmitter`-type attributes), also immutable.

We say that the collection by this becomes immutable.

The idea of "suspending" the label, discussed in Section 3.10.3, is to allow changes in the "mini-registry", in controlled ways, even after labelling. When a labelled collection allows such changes, we say that it is non-immutable, just as a non-labelled collection is non-immutable.

All collections `x`, both labelled and not, have the metadata attribute "`x.metadata.id`". This attribute, of type `int`, is a nonzero value such that `x.metadata.id == y.metadata.id` if and only if `x == y`, i.e. it identifies a collection uniquely. Its value must remain constant throughout the lifespan of a given collection, barring copy-on-write events (labelling and return from `__init`). The value is computed locally at the coordinator and is passed by value to the peer nodes. This is done when a collection is first created, and then again whenever it experiences a copy-on-write event. A collection's `id` must not convey any information beyond its equality or otherwise to other `id` values, which is why transmitting it to the nodes requires no special privileges. If the `id` value is computed as a keyed hash of a meaningful value (e.g., the location where the collection's data is stored in the Variable Registry), that should satisfy this information criterion.

Note 3.5.6. Because all variables referring to the same collection-type entity must all have the same scope, all nodes holding such an entity will agree on all significant events during the lifespan of the collection: when it was created, what its reference count is, whether copy-on-write events occurred at any point, etc.. For this reason, no information is leaked to any node by the knowledge of when a collection's `id` has changed.

However, note in particular that the label must not change due to re-labelling at the end of a `modifying` block (See Section 3.10.3), which is not an event with equal visibility to all participants.

Collections support "`y = x`" assignment (but not "`y[] = x`"). This simply creates in `y` a variable that refers to the same entity as `x`. As is always the case in assignments, the new variable is not protected, even if `x` is.

Collections are neither serialisable nor iterable.

Remark for the advanced reader 3.5.1. The restriction that collection attributes that are themselves of a collection type must have exactly the same scope as their parent collection has substantial impacts on FTIL at large. It imposes on the FTIL programmer some unnatural constraints, but is required for upholding the invariants maintained by the language, necessary for FTIL to meet its design goals.

I am hoping future versions of FTIL might offer better solutions for this issue, but at the moment this is how the language is defined. See Section 11.2.3 for a more detailed discussion of this topic.

We now describe the properties that are unique to the specific types of collections, namely `objects` and `catalogues`.

Objects

FTIL supports an `rm` type, which provides the user with a means to create variables that carry both data (as member attributes) and functionality (as method attributes). The `object` type is the only FTIL type that supports user-defined methods.

Object instances are created by the following command:

```
a = object()
```

The resulting new variable, "`a`", is a variable whose scope is the scope in which the command was run.

Variables of type `object` are collections whose attributes are named and can be accessed through dot notation, as in "`object_name.attribute_name`". The attributes themselves may be

- Of any elementary type,

- Of any collection type,
- Of any `transmitter` type, or
- Methods.

The scope of attributes must be a subset of the scope of the `object`, and may be different from attribute to attribute. Attributes that are of collection types and attributes that are methods must have the same scope as the parent collection. The same `object` can carry both shared and distributed elementary-type attributes, as well as attributes that are a mix of both immutable and not immutable, protected and not protected.

Attributes of any `object` must have distinct names.

Remark for the advanced reader 3.5.2. Suppose that the FTIL user already defined an attribute called "`name`" in some scope, `X`, and suppose that the user now tries to define another attribute of the same name in another scope, `Y`, such that `X` and `Y` are disjoint. It may not be possible for any node other than the coordinator to detect that creating the new attribute would be illegal in FTIL, so it is the responsibility of the Compute Manager, using Variable Registry data, to detect the situation and cause the command to fail.

Suppose, however, that the coordinator node was derelict in its duties, and failed to signal the error. Can the peer nodes nevertheless detect the problem? (Noting that a key issue with the FinTracer system setup is that computing parties do not fully trust each other.)

The answer is that while the FTIL user may be able to illegally define such an attribute (under the assumption that the Compute Manager does not stop it), from the perspective of the peer nodes, any computation involving either one attribute or the other can proceed as usual, as though their names had been different. As long as the commands involved all run either on a subset of `X` or on a subset of `Y`, the fact that there is a name clash has no effect. However, if reference to "`name`" is made in any context that includes nodes from both `X` and `Y`, any such nodes can easily detect in prerequisite checking that the scope of the command does not match the scope of the attribute. The nodes can at that point cause any such command to fail, aborting its execution.

When first created, `objects` contain no attributes (other than their `metadata`). Attributes are created by commands assigning values to them, as in "`a.attr = 7`". The new attributes are created at the scope of the command that created them. If this scope is not legal for the `object` or the attribute, the command fails. The command may also fail for other reasons, such as if the `object` is labelled and the label is not suspended.

For methods, `objects` support the method calling syntax

```
object_name.method_name(parameters)
```

Further discussion on methods can be found in Section 3.5.5.

Catalogues

Catalogues are a type of collection. Catalogues only have attributes that are of type `object`. In particular, their attributes cannot be methods, `transmitters`, elementary types or other catalogues.

The scope of all attributes of a `catalogue` must equal the scope of the `catalogue` itself, this being simply a special case of the general rule that all collection attributes that are themselves of a collection type must have the same scope as their parent collection.

Catalogues are initialised by

```
c = catalogue()
```

This creates a `catalogue` with no attributes (other than its `metadata`), and assigns it to the variable named `c`.

The attributes of a `catalogue` are signified by integers and referenced through bracket notations ("`[]`"), like `dicts`. For example:

```
c[7] = object()
c[3] = c[7]
c[3].attr = True
x = (c[3] == c[7])
```

Here, a new `object` is assigned to `c[7]`, this is then further assigned to `c[3]`. The `object` `c[3]` is then modified by the introduction of a new attribute, and finally the Boolean `x` compares whether `c[3]` and `c[7]` are equal.

As a matter of fact, they are: because `catalogue`s are collections, so referential entities, their attributes are variables, not elements. For this reason, the assignment "`c[3] = c[7]`" is a reassignment of a variable, not a modification of data. The two variables now refer to the same entity of type `object` (the same "mini-registry"), so are equal. In particular, at the end of the code above, `c[7].attr` is defined and equals `True`.

Note that the attributes of a `catalogue`, because they are constrained to be of type `object`, are considered *dependent* variables: assignments to them must be of type `object` or they fail. (This is because there is no implicit conversion path to an `object` type.)

The integer indices of a `catalogue` are not necessarily consecutive. The `catalogue` behaves in this regard more like a `dict` than like a `list`.

The `catalogue` type supports the method "`.keys()`", which returns a `list<int>` of the integer keys of the `catalogue`. This `list` is sorted, and is a shared variable with the same scope as the `catalogue`.

Note 3.5.7. It is critical that the list is sorted, because a shared variable must have the same value across all nodes where it is defined.

The `catalogue` type also supports the method "`.empty()`" to empty the entire `catalogue` in one go, as well as the method "`.size()`", which is semantically equivalent to "`.keys().size()`".

Both "`.keys()`" and "`.size()`" return shared values (within the scope of the `catalogue`), and both rely only on information that exists in the Variable Registry, so are also available to the Compute Manager when making flow-control and scoping decisions.

For it to be possible for the Variable Registry at the coordinator node to keep track of a `catalogue`'s keys even if that `catalogue` does not have the coordinator node within its scope, it is important that any use of the bracket operator in order to refer to a specific `catalogue` attribute, e.g. "`c[i]`", *must* be done with an index value, `i`, that is

- Available at the coordinator node, and
- Is shared among all nodes within the scope of the command, if that scope includes any node other than the coordinator node.

Using any other type of index into a `catalogue` is an error.

Examples of index values that can be used are:

- The `.metadata.id` attribute of a collection at the same scope as the `catalogue`,
- The `size()` of the `catalogue`, and
- A variable `i` that is a `for`-loop variable, looping over the `catalogue`'s "`.keys()`".

Despite the cursory resemblance to the `dict` type, `catalogue`s are collections, not containers, and are not iterable. In other words, they are not eligible for mass operations, and are not serialisable.

Module proxy objects

The *module proxy object*, or *module proxy* for short, is not its own separate type. If `p` is a module proxy object, its FTIL type, as reported in `p.metadata.type`, is "`object`", same as any other `object`.

Module proxy objects are `objects` that are the outputs of a module's `__init` script. They play a critical role in importing new modules, as explained in Section 3.14. Module proxy objects, however, have special restrictions beyond the standard restrictions for `objects`, and also have the `metadata` property "`signature`", which `objects` otherwise do not have.

Note 3.5.8. In previous versions of this document, module proxy objects were known as "module objects". That name is now no longer in use, due to potential confusion with "modules" (regarding which, see Section 3.5.5).

Any entity returned from any script named "`__init`" must be an `object`, must adhere to the following extra conditions, and is automatically made into a module proxy thereafter.

- Its scope must be the scope in which the `__init` was executed (which in turn is the scope in which the `import` command was executed).
- Its attributes must all be either methods or other module proxies.
- It must be labelled.

Once it is returned from the `__init` script, FTIL adds to the `object` the metadata property `.metadata.signature`. This is a shared string, whose value is the signature of the module.

Module proxies also differ from standard `objects` in that they cannot be used as parameters in a `modifying` block.

A module proxy is the FTIL user's gateway into the functionality of imported modules. This being the case, it is recommended that re-running the same import command will consistently lead to the same module proxy object. However, this is in no way policed by the FTIL language.

Remark for the advanced reader 3.5.3. Unlike methods, module proxy objects do not contain, directly, a reference to their module. However, if they have method attributes, those method attributes will have such references.

3.5.4 Special types

In addition to the elementary and collection types described above, FTIL supports a handful of non-serialisable types that have special roles in the language. These are the `transmitters` and `strings`.

Transmitters

FTIL supports a special template type which we refer to as a *transmitter*, which facilitates communication of the information of serialisable, distributed variables between FinTracer nodes, though it is not serialisable itself.

A transmitter, "`rm`" is a template type that looks like a `rm`, except that (unlike in regular dictionaries), `T` does not have to be a fixed-size type. Instead, any serialisable type is allowed.

This is a referential type, so for a transmitter `t` every "`t[n]`" is its own variable. However, like the attributes of a `catalogue`, these attributes of a `transmitter` are *dependent* variables, meaning that assignments into them always follow static-typing rules: they are converted to type `T` before being assigned. (Given that hardly any implicit type conversions will be supported for the MVP, this essentially means that the type will almost invariably have to be an exact match.)

Also, `transmitter` keys are always distributed, so if a shared entity `x` is assigned to a transmitter key, the command's behaviour is as though what was assigned is "`distribute(x)`".

The legal `n` values for "`t[n]`" are exactly the `nodeids` in the scope of `t`.

The command `rm`, which takes a `transmitter` as a parameter, as in "`transmit(t)`", can only be run at the exact scope of `t`, and is privileged, in the sense that all nodes in its scope must be in privileged mode in order to execute it.

The `transmit` command modifies the `transmitter`, `t`, such that any value stored at `t[n_2]` on node `n_1` before the operation will be stored at `t[n_1]` on node `n_2` after the operation. The

"`transmit`" command offers a way to communicate between Segment Managers. One can think of `t[n_2]` on `n_1` as a setup area, where `n_1` prepares any value it wishes to communicate to `n_2`. After a "`transmit(t)`", this value moves to `t[n_1]` on `n_2`, i.e. to a landing pad on `n_2` designated for storing values that came from `n_1`.

Note that the type `T` is declared as part of the definition of `t`, and cannot be changed without dereferencing `t` itself. This means that attributes of `t` are not just statically typed, but that their type is determined before their creation, even when `t` is first declared by a

```
t = transmitter<T>()
```

which is a command that creates an empty `transmitter` at the scope at which the command was run.

The `transmitter` type is not copyable: one cannot, through an FTIL command, create a duplicate of a `transmitter`, nor can one create two variables that both reference the same `transmitter`.

A `transmitter` can be sent as a parameter to a script, and can be an attribute of an `object`.

When an `object` is labelled, making any `transmitter` attribute of it const, the `transmitter` also makes its own attributes const.

If, as part of labelling or another copy-on-write operation, the parent `object` performs a copy-on-write, the `transmitter` entity is first also copied into the new copy of the `object`, and only then makes its own attributes const. At the end of the copying, the new `transmitter` has the same attributes as the original `transmitter`, each of which is a variable that refers to the same entity as the same-named variable in the original `transmitter`. The underlying entities are not copied.

This is a special case in which `transmitter` entities do get copied, even though the type itself is not copyable, in the sense that one cannot copy it through FTIL assignment.

Transmitter attributes have special handling regarding their scope.

Normally, a variable is either non-initialised (in which case it does not refer to an entity at all) or is initialised (in which case it refers to the same conceptual entity everywhere in its scope). Transmitter attributes are the sole exception to this rule. When a `transmitter` is initially created, it does not have any attributes, but each time an attribute is created, that attribute's "`.metadata.scope`" property (serving largely as a place-holder) is set to the same value as the scope of its parent `transmitter`, whereas the attribute's actual value behaves as though it was a separate variable in each node, following FTIL's standard rule for such variables. In each node separately, if it is given a value, it stores that value, whereas if it wasn't given a value it may either not exist at all or be uninitialised (depending on the specifics of the executed FTIL code, following the rules standard variables follow).

It is possible to assign to such attributes using commands that run at any scope that is a subset of the scope of the `transmitter`, and such commands are interpreted as though each node in scope performs the assignment separately. Furthermore, they are interpreted as subassignments, performing type conversion and conversion to "distributed" where necessary.

It is also possible to assign such attributes to other variables using commands that run at any scope that is a subset of the scope of the `transmitter`, and this is interpreted as though it was an assignment of a distributed variable. However, the `transmitter` attribute must be defined and initialised in any node where the command is run. Otherwise, the command fails. The command will fail in such a case even if this is an assignment from one `transmitter` attribute to another `transmitter` attribute.

Strings

A variable of type *string* stores a textual string. In FTIL, strings are meant for communication with users, rather than for inter-node communication, for which reason they are non-serialisable. For example, strings are used in requesting input from the user, including from the peer-node-side user, e.g. by use of the "`Read<T>(s)`" function, described in Section 3.11.1.

Because strings are not serialisable, it is not possible to `broadcast` or `transmit` them, so it might not be immediately obvious how, in the example above, the information in "`s`" can be transferred to the peer node where the "`Read`" is to take place.

To solve this dilemma, FTIL uses special semantics with strings.

Strings in FTIL are *local* variables, in the sense that they exist, both as variables and in terms of their values, only in the Variable Store on the coordinator node.

Operators that manipulate strings or that create strings, such as the `str()` function, are executed locally at the coordinator, even for commands in which the coordinator is not part of the scope of the command. (This means, in particular, that all input to `str()` is parsed and executed by the Segment Manager on the coordinator node, even if that node is not in scope for the rest of the command.)[14]

Any assignment of a string to a variable must happen exactly at the scope of the coordinator.

The FTIL language also supports another type for textual values, this being the *shared string*. This is distinct from a string and is explained in Section 3.5.5. It is a shared type that exists on the executing nodes, but is only used for metadata strings and for explicit literal constants.

When the executor needs to send a string to a built-in function or to run any operator that does not have a string as its output type (which, at the moment, means any operator other than "`+`" between two strings), this is an operation that terminates the special handling for string. The string is at this point converted, implicitly, to a shared string at the scope executing the command, sharing the value of the string (by value) with the appropriate nodes, and the command is then executed with the shared string treated as though it was a literal constant.

This is a rare case of implicit conversion in FTIL.

Implicit conversion in the opposite direction happens when passing an explicit literal string (which is a shared string) as a parameter to an FTIL script: on the receiving end, the variable accepting the shared string's value is a normal (non-shared) string, and the shared-string is implicitly converted to this. Such an operation requires the coordinator node to be in scope for the script. Similar can be said to occur when an explicit literal string is assigned to a variable (though this will necessarily happen only when the command is run at a scope that is exactly the coordinator node).

Implicit conversion from string to shared string is also a case where data may be transmitted from the coordinator to the peer nodes via non-privileged commands. The general rule regarding this is as follows. If a string needs to be converted to a shared string, and then that shared string is manipulated in any way, this is a privileged operation (requiring the coordinator node to be in the script's scope and operating in privileged mode), as one would expect from this kind of communication operation. If, on the other hand, a string is converted to a shared string and then immediately used as a parameter in a top-level built-in command (such as, for example, in "`Read<T>(s)`"), then this usage receives in FTIL special treatment, in that the action is non-privileged and does not even require the coordinator node to be in the scope of the script.

To explain this special handling of top-level built-in commands: though such a conversion (and resultant data communication) is a potential source for data leaks (albeit probably only small-scale ones), we believe that this potential issue can be managed through governance. The alternative, of requiring each `Read` instruction to be individually certified, we deem as impractical, because each user query will require a different string.

In any case, this policy does not create any danger of data leaks from any peer node.

A special case for strings is the built-in function `limit`. This obeys the following syntax:

```
limit(conditional, message_string)
```

When handling `limit`, the conditional is evaluated by each Segment Manager in the command's scope. If for any such Segment Manager the conditional evaluates as `False`, the command fails,

[14] Advanced note: However, if the input to `rm` refers to variables that are in the coordinator node Segment Manager, necessarily the coordinator node must be in scope for the script that the command belongs to. If it is not, the coordinator node does not even participate in the execution stack at this time, so cannot have any local variables of its own.

and `message_string` is sent back to the FTIL user at the coordinator node as part of the resulting error message.

Where `limit` is an exception is that the `message_string` is at no point evaluated by the Segment Managers running the `limit`. If it needs to be evaluated and displayed, this is done by the Compute Manager as part of outputting the error message. The `message_string`, to the extent that it cannot be determined independently by the Segment Managers, is hidden from them. If the `message_string` is a string variable (rather than a shared string, such as an explicit literal), it is at no point converted to a shared string.

Remark for the advanced reader 3.5.4. Any use of strings in flow control and scoping commands follows the standard rules for such commands. These rules are explained in detail in Section 3.7. Readers should probably skip the rest of this section on first reading, and revisit it only after gaining familiarity with these special rules.

To the reader versed with FTIL's handling of scoping and flow-control, the important aspects for the interpretation of string mechanics are the following.

- The values of both regular strings and shared strings is available to the coordinator, so can be used in both flow-control and scoping.
- Any use of shared strings is not considered privileged, because they are shared by all nodes in scope.
- If the scope of the command is not just the coordinator node, any conversion from string to shared string requires the coordinator node to be executing in privileged mode (which, in particular, means that the coordinator node must be in the scope of the executing script), unless the shared string is used directly as an argument for a built-in top-level command.

Let us consider a few example lines of FTIL code (which we've numbered for convenience):

```
1. if str1 in obj:
2.   on(nodes) b = ((str1 + str2) in obj)
3. on(obj.metadata.scope) if obj.metadata.label == str1:
4.   b = (obj.metadata.label == str1)
```

This code assumes the existence of strings called `str1` and `str2`, a `nodeset` called `nodes` (which we'll assume not to include the coordinator node), and a labelled `object`, `obj`, whose scope we will assume to be `nodes`. The variable `nodes` itself, we will assume to be a shared variable whose scope is all nodes.

Let us now consider, line by line, how this code is executed. We will assume that the code is part of a module script.

Flow control is handled by the coordinator alone, so line 1 is exclusively evaluated by the coordinator. Conceptually, it is run by the Compute Manager. Because `str1` is a string and therefore local to the coordinator node, there is no problem evaluating the conditional. The result is True if the `object obj` has an attribute called `str1`. Because this is evaluated by the Compute Manager, over Variable Registry data, the result will be true if the attribute exists regardless of the scope of its definition. (In fact, we assumed that `obj` does not have the coordinator node in its scope at all.)

To handle line 2, first the Compute Manager evaluates "`nodes`" (using Variable Store data available at the coordinator node) to determine the scope of the command. The rest of the command is handled as follows. The coordinator locally (in this case, using Variable Store data) performs `str1 + str2` leading to a string result. This result, implicitly converted to a shared string, is then passed by value to nodes in `nodes` in order to serve as input to "`in`". The "`in`" and subsequent assignment are both handled by each Segment Manager in `nodes` separately. In particular, `b` is now a distributed variable of type `bool`, whose actual value may be different from node to node in its scope, because even though `obj` itself is defined everywhere in `nodes`, that might not be the case regarding each of its attributes.

Line 3 is more complicated. It is a flow control operation, so conceptually handled by the Compute Manager at the coordinator node, but is regarding a comparison where the left-hand side is a shared string whose scope is `nodes`, and therefore does not include the coordinator.

If such a command—still executing on the coordinator node—had not been a flow-control instruction, and so conceptually would have been executed by the coordinator node's Segment Manager rather than by the Compute Manager, this would have caused the command to fail because the left-hand side variable cannot be accessed from the Variable Store. However, this information does exist in the Variable Registry, so the Compute Manager can evaluate it. The Compute Manager evaluates the conditional without needing to convert either side.

Note that the "`on()`" clause also requires Variable Registry data, and is also handled by the Compute Manager, and that this "`if`", therefore, is necessarily and exclusively handled by the Compute Manager (at the coordinator node) even though the scope of the command does not include the coordinator node in it.

Other nodes do not get the value of `str1` as part of this computation. The Shadow Compute Managers receive (by value) the value of `obj.metadata.scope` in order to determine which nodes will be in the command's inner scope, and only those nodes (namely the nodes in `nodes`) receive any information about the `if`, and even then only in the form of the Boolean value of the conditional.

Finally, assuming the conditional is "`True`", line 4 is executed on the Segment Managers at the nodes of `nodes`. They now need to resolve an equality between a shared string and a string. The string is evaluated by Variable Store data at the coordinator (even though that Segment Manager is not in the command's scope), and the value of the result is then sent to the nodes in `nodes`, promoting it implicitly to a shared string. From this point on, it is the individual Segment Managers on `nodes` that perform the comparison and subsequent assignment.

We note that while in both line 2 and line 4 the resulting b is of type `bool`, in line 2 it is a *distributed* variable while in line 4 it is a *shared* variable. The reason is that in line 4 all source variables to the computation are shared: if the inputs are all shared strings, the output is a shared Boolean. By contrast, in line 2 we perform an "`in`" between a (shared) string and an `object`. The `object` is not a type that can be shared, so the result is distributed.

Assuming the example code is part of a module script, as written, all four lines are privileged in the sense that they require the coordinator node to be running in privileged mode. (In particular, this means that the coordinator node needs to be in the execution scope of the script. The assumption is that `str1` and `str2` are local variables of the script, defined on the coordinator node.)

The reasons this privilege is required are as follows.

- On line 1, the Boolean value of "`str1 in obj`" needs to be broadcast by the Compute Manager to all nodes.
- On line 2, the string value of "`str1 + str2`" needs to be converted to a shared string in order to run the "`in`" operator on the individual Segment Managers.
- On line 3, the `nodeset` value of "`obj.metadata.scope`" needs to be broadcast to all nodes, this being the outer scope of the command.
- Also on line 3, the Boolean value of "`obj.metadata.label == str1`" needs to be shared with all nodes in `nodes`.
- On line 4, the string `str1` needs to be converted to a shared string in order to participate in the "`==`" comparison.

If the strings `str1` and `str2` in the example had been replaced by literal string constants (which are shared within the command's entire execution scope and are also available to the Compute Manager), then this would have voided reasons 2, 4 and 5. Regarding reason 2, the result of concatenating two shared strings is already a shared string so needs no conversion. Regarding reason 4, the Boolean value of the result would be intrinsically a shared Boolean (because it is computed only from shared inputs), and its scope encompasses all nodes in `nodes`, so no privilege

is needed. Finally, regarding reason 5 the literal string constant is already a shared string, so no conversion is necessary.

This would have made lines 2 and 4 ones that require no special privilege. The other lines require the coordinator node to be running in privileged mode (and, in particular, be in scope for the executing script).

As a reminder, privilege in converting from string to shared string was only required in the previous example because the string is then further manipulated (on line 2 using "`in`", on line 4 using "`==`"). Had the usage of these strings been to directly send them as parameters to a built-in, top-level function, no privilege would have been required.

As a final point in analysing the example code, consider what the situation would be if the (original) code was part of a non-module script.

Here, lines 1 and 3 would have not been communicated to peer nodes at all and therefore would not require any special privileges. Lines 2 and 4 would have been communicated (not including the scoping portion of line 2), but only to the nodes in `nodes`. Both line 2 and line 4 would still be privileged in this case, for the reasons given above. Note, however, that the scoping portions of commands are not communicated at all to the peers in this model, except by value, so the peers do not know that one was specified as "`on(nodes)`" and the other as "`on(obj.metadata.scope)`". In non-module scripts, scoping (like flow control) can never require privilege.

Combining the two analyses: if the above code had been part of a non-module script *and* strings `str1` and `str2` had been replaced by explicit literal constants, no privilege would have been required to run these lines at all.

3.5.5 Pseudotypes

Pseudotypes are FTIL constructs that look like types but have only very little type-like behaviour. They cannot be assigned into variables, cannot be copied, and support very little that constitutes meaningful manipulation. Some of them can be deleted using a `del` command, others not.

It is an implementation choice of how much these types should or should not use the general FTIL type mechanisms in their implementations.

Some pseudotypes (methods and modules) have some metadata properties that can be queried regarding them (though they do not use for this the `metadata` attribute), but none of the pseudotypes have the `metadata` attribute. They cannot be queried regarding properties that all types have, like `type` and `scope`.

Shared strings

FTIL has two types of strings: strings that are local to the coordinator node, and strings that are shared. The local version was already described in Section 3.5.4. The exact conversion rules between local strings and shared strings were also described there. Conceptually, however, the general idea is that when strings need to be stored as variables, this is done as local strings, but as soon as they need to be passed on to the nodes for processing they are converted to shared strings at the scope of execution.

There is one type of shared string that does get stored, however. This is the string-typed values that are part of an FTIL entity's metadata. This includes metadata fields like "`label`" and "`signature`". Like all `metadata` fields, these fields are shared at the scope of the entity they belong to, and can also be accessed by the Compute Manager through the Variable Registry.

Quite often, strings will appear in FTIL code as explicit literals. This is a second case where the "shared string" type is used. In the case of explicit literal strings, too, their scopes is the command's execution scope and they are also available to the Compute Manager (making them usable in every context).

In terms of the format for explicit literal strings, FTIL supports both normal strings beginning and ending in double quotes (and supporting standard escape sequences) and multi-line strings, beginning and ending in three consecutive double quotes. (In Priority 2, FTIL should also support explicit literal strings that begin and end in single quotes.)

FTIL uses standard backslash-based escape sequences for string literals. Which escape sequences are to be supported is TBD.

Given that FTIL input requests may at times include names, it is probably a good idea for string literals to also support Unicode.

Shared strings are considered shared types for purposes of determining whether a result type is shared.

Scripts

A *script* is a textual representation of a sequence of FTIL commands, potentially parameterised. Scripts are created directly in FTIL via a "`def`":

```
def MyScript(param1,...):
  command1
  command2
  ...
```

This pseudotype is distinct from "module scripts", regarding which see below.

Scripts are never communicated to any of the other nodes. They are interpreted locally. Only the underlying commands are sent to the nodes that need to execute them.

Scripts are immutable and not copyable.

Scripts can be deleted using a "`del`" command, like other types of variables, but whenever they are referenced without being invoked, i.e. without being followed by a "(", their names must be preceded by double colons. For example:

```
del ::MyScript
```

Note that scripts are referenced by their base name: there is no script overloading in FTIL. This is true also for module scripts.

Scripts are discussed in Section 3.13, where we delve into the details of both script execution and script parameter passing.

Modules

Modules are special types that carry multiple scripts, known as "module scripts". Their details are discussed in Section 3.14.

A module acts very much like an immutable, shared variable. However, it has no name (no identifier) of its own, and cannot be explicitly deleted.

Modules are created by running an "`import`" command. This command creates both a module entity and a module proxy object, both at the scope of the command. The module proxy object may have method attributes, and these, in their metadata, reference the module entity.

Like all other FTIL variables, modules are reference counted, and when the count drops to zero, they deallocate themselves. In the case of modules, this works as follows. As long as any method remains, at any given node, that references a particular module entity, that module remains on that node. When no reference for it remains on a given node, it is deallocated. Modules can be deallocated only on some nodes but not others. (As a special case, if the created module proxy object has no methods, the module is deallocated immediately from all nodes.)

The module information exists also in the Variable Registry, where it can be used by the Compute Manager. Like all Variable Registry data, this information is deallocated once every peer node has deallocated the module.

Modules are not copyable.

Module scripts

A *module script*, also known as a *signed script*, is a script that exists inside a module. It is in general identical to a regular script, except in details of its execution, regarding which see Section 2.11.3.

Module scripts do not exist as independent variables, other than that they can be assigned to `object` methods, using:

```
obj.MyMethod = ::MyScript
```

This line must be executed from within a script of the same module, as that is the only context in which "`::MyScript`" is recognised. Also, the scope of the assignment must exactly equal the scope of `obj`, which, in turn, must be a subset of the scope of `MyScript`'s module.

Non-module scripts cannot be assigned into `object` methods.

When invoking the script directly, (e.g., via "`MyScript(...)`"), this can also only be done from within the module that defines the script. The name `MyScript` is not recognised outside its own module, and does not clutter any other namespace: multiple modules can define a script of the same name, without this causing any clashes.

In order to invoke a script outside its own module, use method call invocation. In the example above, "`obj.MyMethod(...)`" can be called whenever `obj` is in scope (subject to conditions discussed in Section 3.10.2).

Module scripts cannot be individually deleted. They are deleted when their module is dereferenced.

Effectively, their scope is the same as the scope of their module.

They are immutable and non-copyable, like non-module scripts. However, because their existence is tied with the existence of their module, they are better described as const, rather than just "immutable".

Object methods

Object instances in FTIL can have attributes that are *methods*, i.e. behave as bound scripts. Methods are an immutable variable type distinct from scripts or signed scripts. They are not copyable.

Methods are created by use of the "`=`" operator. If "`a`" is a variable of type `object` and "`MyScript`" is a module script that accepts at least one parameter, the syntax

```
a.MyMethod = ::MyScript
```

when run from a script in the same module, creates in `a` a new attribute "`a.MyMethod`" of type method.

When assigning a module script into a method, the method retains two metadata attributes, to associate it with the module script. These attributes are

- "`.module`": a reference to the module, and
- "`.scriptname`": the name of the module script, as a shared string.

Methods do not use the `metadata` keyword.

The reference that a method carries to its module behaves in the same way that an attribute of a referential entity does: the module is its own, reference-counted entity, separate from the method, and the method merely refers to it. However, this is a full, variable-like reference, e.g. in the sense that, in the example above, "`a.MyMethod.module.signature`" is the `.signature` attribute of the module entity.

The method created in the example, when called as

```
a.MyMethod(param1_value, param2_value,...)
```

invokes under the hood the script `MyScript` with the same parameters, other than that the invoking `object`, `a`, is inserted as a first parameter. In other words, the underlying module script is called with parameters `(a, param1_value, param2_value,...)`.

During the execution of `MyScript`, `object` instance `a` cannot be reassigned or deleted. (However, if it was previously non-immutable[15], it remains so.) See Section 3.13 for a discussion. The

[15] This may be either because it is unlabelled or because it is in the process of being modified.

method `MyMethod`, itself, is also protected from reassignment and deletion as long as it is on the execution stack.

The ability to reassign, overwrite or delete a method at all, once it has been created, is Priority 2. If it is supported, deleting a method can be done by

```
del a.MyMethod
```

Note that unlike when deleting scripts, deleting methods does not require or permit any use of "`::`".

As a pseudotype (not a full type) methods do not have a "protection counter" of their own. If method deletion is supported, methods will use their parent `object`'s immutability status (whether it is labelled and whether that label is suspended) to determine whether method deletion, reassignment or overwrite is allowed at any given point in time.

Even if methods are not individually deletable, they can always be deleted by deleting their parent `object`.

Methods are shared variables, whose scope is the scope of the `object` to which they are bound. They must be created by an assignment with this scope exactly, or this is an error. For the assignment to succeed, its scope must also be a subset of the scope of the module of the script being assigned.

Metadata

Every initialised variable in FTIL that is not of a pseudotype nor of string type has a `metadata` attribute. A variable's `metadata` is a shared attribute, in the sense that it is always equal on all nodes in the variable's scope, as are all its attributes, including recursively.

Most of the information populating the `metadata` attribute comes from the entity that the variable refers to. The one exception is "`.metadata.scope`", which is a property of the variable.

The `metadata` attribute looks like an `object` in that it has attributes of its own that are accessible through a "dot" notation and are of various types. For example "`a.metadata.scope`" is the scope of variable `a`, which is a value of type `nodeset`, whereas "`a.metadata.label`" (if `a` refers to a labelled `object`) is the label of the entity referred to by `a`. Its type is shared string. All string-type information in the `metadata` is held by shared strings.

An `object`'s `metadata` cannot be directly manipulated by the FTIL user. It is in that sense a "constant" attribute, and its own attributes are also constant.

The `metadata` attribute can also not be copied into a variable.

The full contents of the `metadata` attribute are described in Section 3.8.1.

This content is all available on the Variable Store for the Segment Manager of every node in scope, and is also in the Variable Registry for the Compute Manager.

In terms of FTIL code running on the Compute Manager (e.g., scoping commands and flow-control commands), `metadata` attributes that are in the Variable Registry are accessed via exactly the same syntax as though the entities were in scope and the data was in the Variable Store. For example, if on *some* node the variable "`a.b.c[7].d`" is defined, the Compute Manager can evaluate "`a.b.c[7].d.metadata`". This uses the fact that both collection data and entity metadata are all available on the Variable Registry.

Methods, modules and `type`s in FTIL also have some metadata properties, even though they are not FTIL types but rather pseudotypes, but the metadata properties of these are accessed through the "dot" syntax without use of the `metadata` keyword.

The "type" type

The last FTIL pseudotype is the `type` type. It exists in two forms: first, it can exist in the metadata, as an entity's "`.metadata.type`" attribute, and second, it can appear as an explicit literal constant.

The "`.metadata.type`" attribute, like all other metadata attributes, is constant and shared at the scope of the entity. It signifies the entity's type.

The explicit literal, like all explicit literals, is constant and shared in both the scope of the command and by the Compute Manager.

This type cannot be assigned, copied or modified, but can be used in comparisons. For example, in order to execute "`a.metadata.type == int`".

If `t` is a non-literal of type `type` (necessarily, a metadata attribute), then "`t.name`" returns a human-readable shared string that provides the type name.

This concludes all Priority 1 requirements from `type`.

In Priority 2, the `type` type should also allow introspection into FTIL variable types, as discussed below. The features below only begin to be really useful if `array`s are supported, and none of them are used in any of our example code.

For introspection, the `type` type should have additional attributes accessible through dot notation, as follows.

Containers and composite fixed-sized `type`s should have a `basename` attribute, also of type shared string, that provides the base of the type, e.g. `"pair"`, `"AccountID"`, `"AccountAddr"`, `"ElGamalCipher"`, `"array"`, `"set"`, `"dict"`, `"list"`, `"memblock"`, etc..

Such types should have additional attributes for their constituents. For example a "struct" type, like `pair`, should have an attribute for each of its struct elements, each (of type `type`) providing the type of the element. In the case of `pair`, these attributes will be named `first` and `second`. This should also be done with struct types that are not template types, like `AccountID`.

Atomic (i.e., non-composite) fixed-sized `type`s should have a `bytesize` attribute, of type `int`, that provides the type's storage size, in bytes.

Container types should have an attribute "`element`" (of type `type`) that provides the type of their underlying elements, except for `dict`s and `hashmap`s, which should have both "`key`" and "`value`".

Array types should have both the "`element`" attribute and a "`size`" attribute, of type `int`.

Like all `metadata` attributes, the attributes of `type` are all shared in the entire scope of their variables, and are also visible to the Compute Manager.

Though entities of the `type` type cannot themselves be assigned into variables, if an attribute of a `type` is itself of a type that can be assigned into a variable, this assignment is allowed. Specifically, it is allowed to assign the `size` attribute of an `array` into a variable.

3.6 Constants

FTIL provides several constants, which we refer to as *system constants*. Unlike standard variables, these can be referred to from scripts freely, with no need to send them explicitly to the scripts as parameters.

The following is the list of such constants.

`rm`: A distributed constant of type `nodeid`. Holds for each Segment Manager the identity of its own node.

`rm`: A shared constant of type `nodeid`, identifying the node hosting the Compute Manager. This is shared among all nodes.

Additionally, each node will have its own shared constant, similar to `CoordinatorID`, which will provide its name to all participants.

3.7 Anatomy of an FTIL command

Readers of Chapter 2 may remember that FTIL's execution paradigm is one where one node, the coordinator, determines what all FinTracer nodes do, and all nodes execute the commands.

Conceptually, we separate actions to one of two jurisdictions:

The Compute Manager (which exists only on the coordinator node) determines which commands are to be run at any point in time, and

The Segment Managers (which exist at every node, including at the coordinator node) perform the computations.

Moreover, as discussed in Section 2.4, the Segment Managers work based on data that exists in their respective Variable Stores, whereas the Compute Manager has its own storage area, namely the Variable Registry, for its own use.

The exact contents of the Variable Registry are discussed in Section 3.8 and the exact contents of the Variable Stores are discussed in Section 3.9. For the purpose of the present discussion, only the following points are important:

1. When analysing an FTIL command, it is important to understand which parts of it are interpreted by the Compute Manager vs by the Segment Managers, because these work off of different information.

2. The peer nodes also have a secondary role which we refer to as *Shadow Compute Managers*, where each node oversees the Compute Manager, to determine that its instructions are correct, and are valid. This role is performed using each node's own Variable Store data.

Our use of the term "Compute Manager" in this document should not be taken as representing a software component. Rather, it is shorthand for saying that Variable Registry data is being used. Because what the Compute Manager computes using Variable Registry data often needs replicating by the Shadow Compute Managers using node-individual Variable Store data, it is, in fact, very likely that in a software implementation much of the Compute Manager's functionality will be off-loaded to the coordinator node's local Segment Manager to execute (by which we mean, to whatever software component handles standard execution).

Importantly, in the FTIL syntax as described here, there is no distinction between access to data by the peer nodes vs by the coordinator node, by Segment Managers vs by the Compute Manager, or from the Variable Store vs from the Variable Registry. The distinction is merely in the context, e.g. whether an expression is part of a flow control instruction or a standard instruction.

For reasons to do with ease of implementation, maybe this choice will be revisited at a later time, but at the moment we chose it to reflect the fact that in the peer nodes no distinction exists, nor *should* any distinction exist, between access to the local Variable Store that in the coordinator node is handled by the Segment Manager and one that in the coordinator node is handled by the Compute Manager.

Note 3.7.1. The above statements refer to the portions of the Variable Registry that are programmatically accessible from FTIL. The Variable Registry also includes much information that cannot be accessed by the FTIL programmer directly. This is all explained in Section 3.8.

In this section we describe the FTIL-visible portions of how individual FTIL commands are interpreted and what data is available at each step.

The general principles at work here are the following.

- While the Compute Manager, at the coordinator node, fully determines what is run, the Segment Managers must individually be able to understand and semantically verify each command. This creates tension between what the Compute Manager may want to do (potentially anything) and what the Segment Managers would allow (only what they can fully verify). FTIL balances these needs.

- The FTIL computation model is fully parallel, meaning that the same coordinator instruction runs on multiple nodes simultaneously and must have the same semantic meaning in all. This constrains FTIL so as not to allow meanings of the same command to diverge between nodes.

Note 3.7.2. We exclude from this discussion commands that explicitly work on Variable Registry data, such as `dir` and `help`. These are locally-run commands that do not admit a scoping directive, and are handled beginning to end by the Compute Manager, using the data appropriate to them.

3.7.1 Flow control

The first task of the Compute Manager is to determine what the next command to be executed is. While on the command line, this is straightforward to do: the next command input is the next command to be executed, and only after its execution completes is the next command read.

In scripts, however, commands follow more complicated flow control: there may be (potentially nested) script calls, there may be conditional execution, and there may be loops.

Flow control in FTIL scripts is detailed in Section 3.13.4. For the purposes of segmenting Compute Manager from Segment Manager operations, however, important is the following.

The Compute Manager evaluates what the range of a loop is and what the conditional is in an "`if`" or "`while`" statement. (We refer to these as the *parameters* of the flow-control instruction.)

These evaluations are done entirely at the coordinator node. The evaluations may use any data available at the coordinator node that is programmatically accessible from FTIL, whether it is in the Variable Store or the portion of the Variable Registry directly accessible to the FTIL program.

If the script is an unsigned script, the process is as described in Section 2.11.2: the other nodes are not even aware of the existence of the flow control statements, and do not see their parameters. They only know the scope of commands they are actually called on to execute.

If the script is a signed script, however, the process is as described in Section 2.11.3, and is essentially as follows.

The flow-control parameters, once evaluated by the Compute Manager, are broadcast out to the peer nodes that are in the command's (outer) scope.

Conceptually, signed script flow control decisions are taken by the Compute Manager, with the Shadow Compute Managers following along, using only the data available to them individually.

If the scope of the command is just the coordinator node, no further data communication is necessary, and the coordinator executes the command as usual.

Otherwise, two possibilities may occur.

The first case is that the values derived by the coordinator node are shared and accessible also to the nodes in scope. In this scenario, the Shadow Compute Managers verify that the values broadcast from the Compute Manager match the local-node result, or the command fails.

The second case is one where the values derived by the Compute Manager are either not shared or shared but not at the full required scope. In this case, any Shadow Compute Manager that still can check the value does this as before. All other nodes use the result received from the coordinator node. In this case, the parameters are sent from the coordinator as though by an implicit `broadcast` instruction, and exactly as in normal invocations of `broadcast`, in this case the operation is privileged for the coordinator node.

3.7.2 Scoping

As previewed in Section 2.10, other than very few, exceptional FTIL commands that are local, meaning that they can only be executed entirely on the coordinator node and do not involve any form of inter-node communication, all FTIL commands are structured as

```
on(scope) command
```

where the "`on(scope)`" part is optional. Within scripts, it is also possible to use "`on()`" scoping as a block-defining directive like so:

```
on(scope):
  command1
  command2
  ...
```

Also, scripts can be used on commands that themselves create blocks, such as in:

```
on(scope) if condition:
  command1
  command2
  ...
```

The scoping part says *where* a command is to be executed, and the rest is *what* the command is. The "`scope`" may be a `nodeset`, or may be a `nodeid` that is treated as a singleton `nodeset`. The `scope` may or may not include the coordinator node.

A command's scope may be given as a system constant, a variable, or an expression to be evaluated.

A scoping directive on a command can only narrow the execution scope, never expand it. Thus, when executing nested scripts, one is also employing nested scoping, because each script invocation may be done in its own scope. Similar nesting of scopes happens when executing inside a scoping block, or when scoping is applied to a (flow-control) command that creates a block.

Running a scoping command with a scope that includes any node not within the present scope is an error, and will cause the command to fail.

At the end of the scoped command, scoped block, or scoping block, the scope returns to what it was prior to scoping.

Where the scoping directive is missing, the command executes at the widest possible scope given the context. On the command line, this means that the command executes at the scope of all FinTracer nodes.

In terms of command structure, the two parts of the command are interpreted differently.

The scoping portion is interpreted by the Compute Manager in the same way as a flow control command is interpreted: any information accessible to the Compute Manager, whether it is in the coordinator node's Variable Store or in the portion of the Variable Registry programmatically accessible to FTIL, can be used.

If scoping is done as part of a non-module script, the process for this is as described in Section 2.11.2. Essentially, each node is only made aware of the individual commands it is asked to perform[16], and the scope in which that command is executed. Otherwise, it is oblivious to all scoping commands, and is not even aware that they are executed.

If scoping is done as part of a module script, however, the process is as described in Section 2.11.3: it is the Compute Manager, alone, that does the evaluating, but then the scope is broadcast, by value, to all nodes in the command's *outer scope*, by which we mean the scope before it was narrowed down by the scoping command.

It is then the nodes in the command's *inner scope*, the scope after narrowing down, that execute the command, and it is only they that do prerequisite checking (regarding which, see Section 3.7.5) to verify that all variables referenced by the command are, indeed, known to the node's Segment Manager.

In executing a signed module, the behaviour of the nodes receiving a scoping parameter can be one of three.

In the first case, the scope of nodes with which the command needs to be shared is exactly the coordinator node alone. In this case, no further communication is necessary, and the coordinator node Segment Manager proceeds with executing the command as usual.

In the second case, the resulting scope value, as evaluated at the coordinator node, is a shared value among all nodes with which the command needs to be shared. Examples of such shared values are:

- A system constant,
- A `for`-loop variable,
- A variable shared within the necessary scope or beyond it, as well as at the coordinator node,

[16] Where "individual commands" covers in this case also the start and end of the execution of any script, to allow nodes to follow the execution stack.

- A `metadata` attribute shared within the necessary scope or beyond it (whether or not it appears in the coordinator node's Variable Store), or
- A computation result whose inputs are entirely of the above.

Regarding the last case: barring a few exceptions (such as randomness-generating functions), functions in FTIL whose inputs are shared within a scope are considered to also return values that are shared within that scope.

Note 3.7.3. We discuss such evaluation here in the context of scoping (which is identical to the case of flow control). In the context of standard execution, variables are computed in the (inner) scope of the command that creates them. If, in some scope, `n`, we compute "`a = 1 + 1`", the created variable `a` will be created as a shared variable in the command's scope, `n`. It does not matter that the same command would have been legal to compute also in a wider scope.

If the resulting scope value is a shared value among all nodes with which the command needs to be shared, as in this second case examined, the resulting scope is still `broadcast`, by value, to all these nodes, but their Shadow Compute Managers also compute it independently, in order to verify that the value is correct.

The third and final case is one where the resulting scope value is not a shared value or is shared, but not at the full desired scope. In this case, the value is computed by the Compute Manager and `broadcast` by value to the necessary nodes, but then the command becomes privileged for the coordinator node, meaning that it needs to run in privileged execution mode (See Section 2.11.4), in order to execute the command. It may be the case that the value is shared in some portion of the scope but not elsewhere. In this case, any node that is inside the shared value's scope does independent checking of the value received from the coordinator node, as usual.

All this is regarding the scoping portion of the command.

The root command, however, is interpreted differently. It is interpreted according to the nature of the command. If the command is a standard command, for example, it will be handled as per standard execution rules: the Segment Managers within the inner scope execute the command, using only Variable Store data (See Section 3.7.5). If, on the other hand, the command is of a different type, such as, for example, a flow control instruction, it is handled according to the rules of that type.

3.7.3 Strings

Having handled the clearly-delineated flow-control and scoping portions, the FTIL executor now needs to work on standard commands.

Its first task is to handle strings.[17]

Why are strings different?

Strings in FTIL are one of two "special" types, and their handling is very different to what is done with elementary types. Before delving into the actual solution of how FTIL's string handling works, it's worth mentioning briefly why we do not want to use with strings the same kind of logic as used for fixed-size or for container types, as this will clarify why string handling in FTIL works the way it does.

Strings in FTIL have two uses:

1. They can communicate with human users (for example, they may prompt a peer-node user for specific manual inputs); and
2. They may carry SQL query strings (for querying the Auxiliary DB or the Transaction Store).

[17] We stress again that this entire section relates only to standard commands, not to flow-control or scoping. Any strings encountered in flow-control or scoping decisions are handled purely by the Compute Manager as per the process described above, and completely bypass everything described here.

In both cases, the role of strings is quite unique:

- They behave more like code than like data, in that they carry instructions from the coordinator node to the peer nodes, but
- They behave more like data than like code, in that in realistic usage their value will be unique to each invocation, so they cannot undergo certification in the way scripts do.

As usual with FTIL, this presents us with almost-contradictory requirements: on the one hand, the above suggests that string transfer between nodes needs to be allowed, and, in fact, be quite unregulated. On the other hand, even the passing of a single string between nodes may present significant data leaks, because strings are not fixed sized types, and a single string may pack significant amounts of data.

Conceptually, FTIL's way to reconcile these nearly-contradictory requirements is to restrict what can be done with strings to the point that string manipulation can essentially only follow the same patterns and pathways as code, to allow as much regulation as possible of these pathways via governance mechanisms (much as is done with code), and to make the minimum possible allowances in order to enable the use of unique strings with every invocation in order to allow code reuse without constant re-certification.

The solution

If the scope of the command is exactly and solely the coordinator node, this handling is very simple: everything is done by the coordinator node's Segment Manager, using the data available to it (and no special privileges are needed).

This is because most of the complications in handling strings stem from disparities between the view of the coordinator node Segment Manager (which handles strings) and the Segment Managers that are in the execution scope of the command (which handle the rest of the command, and are a set of nodes that may not even include the coordinator node), with the Compute Manager on the coordinator node using Variable Registry data to bridge between the two. When the execution scope is exactly the coordinator, no such disparity exists.

This "simple case" immediately deals with all commands that assign a string (or a shared string) to a variable, because such assignment can only happen at the coordinator node scope. So, we can safely exclude this type of command from all further discussion. Assignment of a shared string to a variable requires an implicit conversion from shared string to string, but this requires no special privileges.

A similar case that we carve out of our discussion is handling of the optional message string parameter in the FTIL built-in top-level command "`limit`". This parameter is not evaluated at all unless `limit` triggers a command exit. If an exit is triggered, the message parameter needs to be evaluated because it is displayed to the user as part of the FTIL exit message. When this happens, the parameter is evaluated solely on the coordinator node, and its value is never shared with any peer node, making this into a special case of the "simple case" handling.

Excluding the "simple case" where the scope of a command is exactly the coordinator node and the related handling of the `limit` message parameter, the following is how FTIL handles string operations inside standard (non flow control, non scoping) commands.

Consider the execution tree of the command in question. We begin by isolating those sub-trees of the execution tree that compute strings. This includes the execution sub-trees whose roots are proper built-in functions that return a string. (There are presently only three such functions in FTIL: "`str()`", "`dateoffset()`", and the string concatenation operator "`+`".)

We stress that we are only interested in *partitioning* the execution tree. The end result will be a partition where each part is a string-generating sub-tree, except for one part that encompasses "all else". In particular, if some function outputs a string value that is later used by another string-generating function, only the latter's sub-tree is important to us. The sub-tree under the former will be contained inside the partition part represented by the larger sub-tree.

Note that in FTIL the root of the execution tree may not be a proper built-in function at all. It may also be a script call, or, alternatively, a built-in top-level command (a command that either returns no value or returns a value that can only be immediately assigned to a variable). At the moment, we consider only the parts of the execution tree that are proper built-in functions, leaving any top-level command or script for later handling. Such top-level commands / scripts are therefore not mapped into any of the execution sub-trees we now isolate, even if they accept string inputs.

Some of the execution sub-trees thus mapped will contain only leaves: a string that already exists in the system, such as a string variable, a metadata attribute or a literal constant, will be used directly. These we leave for now as-is.

The remaining execution sub-trees that generate strings are now executed. This computation is done entirely by the coordinator, regardless of the scope of the command. Information that can be used here is

1. Any information that is available to the coordinator node Segment Manager (if the command is run from the command-line at a scope that includes the coordinator node, or if the command is run as part of a script whose execution scope includes the coordinator node), and

2. Any Variable Registry information that is shared among all nodes that are in the command's scope (whether or not that scope includes the coordinator node).

For example, if any non-string input is a variable, the variable's value at the coordinator node is used. If the variable is not defined at the coordinator node (even if it is shared among all nodes in the scope of the command), this is an error.

To understand the state of the system after the sub-tree computations occur, let us first introduce some terminology.

We say that an FTIL entity is *widely shared* if its value is known and is known to be equal everywhere in the scope of the running command as well as at the coordinator node.

To satisfy the condition of known equality everywhere in the scope of the running command, the value may be any shared elementary entity or any shared string, as long as they are defined in the command's entire scope. Alternatively, a literal constant would also satisfy the condition.

To satisfy the condition of known equality for the coordinator node, one alternative is that the coordinator node is part of the command's scope. Alternatively, any value that exists in the Variable Registry, such as metadata attributes and `for`-loop variables also works. Literal constants, too, are visible to the coordinator node, regardless of the scope of the command.

We say that an FTIL built-in proper function is *stable* if it will always return the same outputs for the same inputs, regardless of when it is executed or on which node. Most FTIL built-in proper functions are stable. Examples of *unstable* FTIL functions are ones that generate random values. Stability is a critical feature in FTIL, because we compute functions over shared inputs, and expect the output (because it is also shared) to be equal in every node where it was calculated.

There are, however, functions that are harder to categorise, which we refer to as *in-principle stable*. The idea of in-principle stable functions is that if two FinTracer nodes are running the same software on the same hardware, invocations of these functions with equal inputs on these nodes will yield equal outputs. However, differences in hardware and software implementations may cause a divergence of the results. Examples of in-principle stable functions that are not necessarily stable are the string manipulation functions "`str()`", "`datediff()`" and "`dateoffset()`".

With this terminology in place, consider the following.

After the coordinator finishes processing the sub-trees producing strings in a command's execution tree, the part of the execution tree that still remains to be executed only has strings on its leaves. These strings can be divided into two categories:

- Some strings were from the start leaves in the execution tree, in which case they keep their original identities: what was a string remains a string, what was a shared string remains a shared string.

- Some strings are the result of the coordinator's computation. They are initially in string form, not as shared strings, but all get subsequently converted to shared string form and shared with the nodes in the command's scope.

Regarding the strings that are computation results and now need to be converted to shared strings, normally such a conversion would require privilege at the coordinator node (because this is essentially a `broadcast`-like operation). However, in two cases no special privileges are needed.

1. If all the leaves in the sub-tree that computed the string are widely shared, and if all functions in that sub-tree are stable, no privilege is required. The reason for this is that we consider these as a case where shared inputs lead to shared outputs. The coordinator node was technically the party to perform the computation, but conceptually this is work that was merely offloaded to the coordinator, and truly belongs on the individual nodes.

2. Some computed strings are at this point immediately used as parameters to a top-level FTIL built-in command. They are not manipulated further through additional proper functions. In this case, in the interest of allowing code reuse without requiring constant re-certification of code utilising strings, the conversion from string to shared string does not require privilege. (We remark that this opens up the potential to a minor data leak from the coordinator node to the peer nodes, if sensitive information is encoded in a string sent in this way as a direct command parameter, but we believe the risk of this is small and can be managed through governance. In any case, no data can leak from peer nodes due to this concession.)

Remark for the advanced reader 3.7.1. Regarding the first item, note that we require that the functions be stable. If they are in-principle stable, that is not enough, because FTIL does not allow shared variables to have different values between nodes.

Some complications in this section can be avoided if the FinTracer system is implemented in such a way that all in-principle-stable functions listed above are guaranteed to always return the same results on all nodes. This will make these functions from "in-principle stable" to just "stable".

Regarding pre-existing strings that were not computed by the command itself, if the string was originally a shared string, it can be used as a shared string without further complications. If it was originally a non-shared string, it will need to be converted to a shared string, and this will require the coordinator node to be operating in privileged mode, before the string can be used as a parameter either to a top-level command or to a proper function.

For pre-existing strings, however, there is also the possibility that they will be used as parameters to an FTIL user script. If they are non-shared strings, they can be used directly: the string will get another identifier, this being the formal parameter, as per the usual script invocation process described in Section 3.13.3. If they are shared strings, they need to be converted to non-shared string form before this can be done. This is a process internal to the coordinator node, and does not require privilege. However, it does require the coordinator node to be part of the scope in which the script is invoked, or else script invocation will fail.

Once all conversions from string to shared string and vice versa have been handled, command execution can continue (and conclude) through the standard execution pathway, as described in the next sections. In other words, all else, with the exception of the command parts described in Section 3.7.4, is handled by the individual nodes in the command's scope.

Handling in-principle stable functions

Unrelated to our previously-described reasons for special handling for string functions is the need to provide special handling for all in-principle stable functions that are not guaranteed to be stable, in order to ensure parity for all shared outputs computed by them. This is yet another constraint for any FTIL solution.

Because all in-principle stable FTIL functions that are potentially unstable are string-handling functions, we describe their handling here.

Specifically, there are three functions in FTIL that fall under this category, namely "`str()`", "`dateoffset()`" and "`datediff()`". Of these, the first two return strings, and are simply handled by the mechanism already described: because the result is calculated by the coordinator node and then propagated out to the peer nodes, it is guaranteed to equal across all nodes.

If FTIL is implemented in a way that guarantees parity of the results for "`datediff()`", no further complication is required.

Otherwise, computation of "`datediff()`" must also be offloaded to the coordinator node: the inputs to `datediff` are sent to the coordinator, the coordinator performs the computation, and returns the result. No special privilege is required for this.

Notably, this may lead to a situation where a computation begins at the coordinator node, leads to a string that is shared with the nodes in scope, which then send it immediately back to the coordinator for computing "`datediff()`". This is completely by design, as it provides the peer nodes with full visibility and auditability into the computation thus offloaded to the coordinator.

An executor view of string handling

From the perspective of the executor, too, strings require special handling.

First, on any conversion of a string to a shared string (whether or not privilege is required) it is important for the peer nodes receiving the value to log the value of the string. This is because it is effectively part of the code to be executed.

We log the string value just as we log the executing FTIL command. The only difference is that logging of the string value should preferably use some form of masking, as the string value is likely to be much more sensitive than the raw FTIL command. (This is because the command is generic but the string value is use-cases specific.)

Second, we note that if a string was computed based on an execution tree that started from widely shared inputs and contained only stable functions, then each of the peer nodes has enough information to compute the result independently. In practice, the shadow compute managers should perform this computation and verify that the result received from the coordinator node matches the locally computed one. If there is no match, the operation should fail.

Note 3.7.4. This is not an unnecessary overhead. In this way, peer nodes can ascertain both that the value is correct and that the same value is used in all other nodes.

If a string is computed based on an execution tree that started from widely shared inputs and contained only in-principle stable functions (though some that are potentially unstable), the shadow compute managers nevertheless perform the mirror computation and compare the local result with the received result. In this case, a mismatch should not abort processing, but it should nevertheless be logged as a warning.

3.7.4 Indirect addressing to catalogues

For the most part, the core portions of a command (after dealing with scoping, flow control and strings) are handled by the standard execution pathway (regarding which, see Section 3.7.5). There is, however, one exception.

Consider indirect addressing: "`x[i]`".

In general, FTIL supports indirect addressing. The type of syntax shown above works when `x` is of type `array`, `list`, `memblock`, `dict`, `transmitter` and `catalogue`, and in all of these except `catalogue`, the value of "`i`" is evaluated through the standard execution pathway.

The problem with allowing the same for `catalogue` is that very quickly FTIL code can diverge between computation nodes. Consider, for example, the following (illegal!) code snippet.

```
a = object()
a.MyScript = ::ScriptA  # This is some previously-defined module script.

b = object()
```

```
b.MyScript = ::ScriptB  # This is another previously-defined module script.

c = catalogue()

i = Read<int>("Enter either '0' or '1'.")
assert(i == 0 or i == 1)

c[i] = a    # THIS IS AN FTIL ERROR.
c[1-i] = b  # THIS IS ALSO AN FTIL ERROR.

c[0].MyScript()
```

This code assigns one of the `objects` `a` and `b` into `c[0]` and the other into `c[1]`, and then invokes the method `c[0].MyScript()`. What code should be executed, however? Is it `FunctionA` or `FunctionB`?

Had this code been allowed in FTIL—and it isn't—the choice of either `FunctionA` or `FunctionB` would have been individual to the nodes, depending on a value, `i`, locally input on each node by the node's user, and unknown to the Compute Manager. The Compute Manager, at this point, would have lost control of the execution, and the computation model would have devolved from parallel to merely distributed.

Instead, FTIL restricts what values can be used as indirect addresses into `catalogues`, and these values are not resolved using the standard evaluation pathway.

To resolve indirect addressing to `catalogues`, the Compute Manager and the Segment Managers separately resolve the address. However, the Compute Manager and Shadow Compute Managers both verify that this address is a value that is both

1. Known at the coordinator node, and
2. Shared among all participants in the computation.

This can be, for example, a shared value whose scope includes the coordinator, or may be a metadata value, read from the Variable Registry, for an entity that may or may not have the coordinator in its scope, or it may be a `for`-loop variable.

If the `catalogue`'s scope is solely the coordinator node, there is no need for the value to be shared in any way. It is enough that it is known at the coordinator node.

As a result, the code snippet that was presented above as illegal can be made legal by enclosing it inside an "`on(CoordinatorID):`" block.

3.7.5 Standard command evaluation

Once flow control, scoping, strings and indirect addressing into catalogues have all been resolved, execution is down to the command's core.

This has two stages:

1. Prerequisite checking, and
2. Actual command execution.

The process of prerequisite checking is meant to make sure that every variable accessed by the command is adequately defined and is of a satisfactory type/flavour/scope/etc.. It is described in detail in Section 2.11.

In terms of data that can be used for prerequisite checking, here the Compute Manager can use all data available to it via the Variable Registry, but no local Variable Store information. Importantly, the Compute Manager is not restricted—as it was elsewhere in the execution pathway—to only use elements from the Variable Registry that are programmatically accessible through FTIL. Any information within the Variable Registry can be used.

The individual Shadow Compute Managers that are in scope repeat this prerequisite checking individually, each using the data available on their nodes.

In general:

- Variables that are read need to be predefined before the command is executed.
- Variables that are written to do not have to be predefined ahead of time. Their values may be given to them by the assignment. If they do exist previously, however, it must be possible to overwrite the previous value. For example, if we perform `a = b` on one node (assuming that this is dynamic assignment of a variable `a`) and `a` has previously been defined on all nodes, this would be an error: the previous value would first need to be deleted on all nodes, before `a` can be redefined in a new scope.
- Calling a script is considered neither read nor write in terms of the parameters sent to it: one can send to scripts that run on all nodes parameters that have only been defined on a single node.

Once all prerequisite checking concludes successfully, the command is executed: the individual Segment Managers in the command's scope execute the command individually, using only their own Variable Store data.

Note that even if a command did not fail during prerequisite checking, it may still fail during this final execution step.

Consider, for example, the command

```
a = 1.0/c[7].t[CoordinatorID][3]
```

Let us assume that `a` was not previously defined, and that `c` is of type `catalogue` and `c[7].t` is of type `transmitter<list<float>>`.

These assumptions can all be verified by the Compute Manager and by the Shadow Compute Managers. The Compute Manager can also, as part of this, determine that "`7`" is a key of `c`.

However, the Compute Manager cannot determine whether "`c[7].t[CoordinatorID]`" is initialised, or whether this list is long enough for its index "`3`" to be defined, or—if it is defined—whether it does not equal zero, making the whole expression a division by zero. These are all checked by the Segment Managers that are in scope.

It does not matter for our purposes whether these checks can be done in advance (as in the example above) or whether they cannot. (For example, a call to `AuxDBRead` may have a query parameter that is—as an extreme example—a legal SQL expression that refers to a table that does not exist in the Auxiliary DB. It is not possible to determine that this command will fail without actually trying it out.)

Because prerequisite checking is done solely using Variable Registry data, whereas actual command execution is done solely using Variable Store data, the distinction between what is in the Variable Registry and what not reflects what can be checked as part of a command's prerequisites and what can only be checked as part of the command's execution.

To highlight this, consider the following, different example. Consider the same line of code as before, but this time let us now assume, regarding the same code, that the variable `a` *does* exist. If all other parts of the command are legal, it is the properties of `a` that determine how to handle the assignment. In particular, if `a` is protected the assignment is not legal. For this reason, it is important for each variable's protection status to be part of the Variable Registry, even though it might be different from node to node.

The following two sections, Section 3.8 and Section 3.9, detail exactly what information exists in the Variable Registry, and what information is only in the Variable Store.

3.8 Variable Registry data

In Section 2.4, it was explained that variable data in the FinTracer system is stored in two places: the Variable Store (where each node stores information about its own variables) and the

Variable Registry (where the coordinator node stores information about all variables in the system, regardless of their scope).

In Section 3.7, we discussed how and when each repository is used, and demonstrated that the choice of "what data is stored where" impacts what code the language can execute.

In this section, we recount in detail the various types of data that exist in the Variable Registry. Essentially all of them are also kept track of in the Variable Store by each Segment Manager at each node for the variables visible at that node.

Almost all information that is in the Variable Registry, and certainly all parts of it that are programmatically accessible to the FTIL user, is identical and shared among all nodes within the scope of the relevant variable/entity/etc.. However, some items stored in the Variable Registry are different from node to node, in which case they are stored in the Variable Registry on a per-node basis. Such items can only be used for prerequisite checking, and not for any other use.

Variable Registry information includes

- Entity metadata,
- Entity Variable Registry data,
- Variable properties, and
- Identifier properties.

The portion of the Variable Registry that is "programmatically accessible to FTIL" is the "Entity Variable Registry data" and the portions of the "Entity metadata" and the "Variable properties" that are flagged specifically as accessible through the `metadata` property, or otherwise using attribute syntax, in the case of pseudotypes.

Note 3.8.1. By design, all programmatically-accessible portions of the Variable Store are shared among all nodes in the scope of the affected variables/entities, and all operations that affect them must run in the full scope of these affected variables/entities.

Note 3.8.2. Explicit literal constants are taken in FTIL to be shared variables whose scope is the command's execution scope, and which are also visible to the Compute Manager. This ensures that they can be used in all scenarios.

The guiding principles that determine what goes into the Variable Registry and how are as follows for information that is shared among all nodes in scope.

- The Variable Registry keeps a graph whose vertices are the identifiers and the referential entities in the system (with their attributes) and whose edges are the variables linking them. Each Variable Store has a sub-graph of this graph, consisting of those entities and variables that are in scope for its node, and the Variable Registry graph is the union of all these sub-graphs.
- Elementary entities are also stored, as leaves of this graph, and also maintain their attributes, but in the case of elementary entities the mapping between Variable Registry and Variable Store is not necessarily one-to-one. Also, the Variable Registry only stores metadata attributes of elementary entities, not their data payload.
- For `transmitter` attributes, the Variable Registry does not keep track of whether they have been initialised or not.

Note 3.8.3. The reason that it is important that there is no one-to-one correspondence between elementary entities in the Variable Stores and in the Variable Registry is that this allows each to only copy-on-write when it needs to, irrespective of what happens in any other Variable Store/Registry. For example, when the value of an elementary entity is modified on only a single node, and there is another variable referring to the same entity, that node must copy out the entire elementary entity (regardless of the size of the change), but the nodes that did not participate in the modification do not have to, nor does the Compute Manager.)

The reason for the asymmetry in the handling of `transmitter`s versus `catalogue`s is that indirect addressing into `catalogue`s is restricted to what can be determined by the Compute Manager (and therefore managed in the Variable Registry). By contrast, `transmitter`s have no such restrictions. It is possible in FTIL, for example, to read a `nodeid` value, `n`, using a `Read` command (where this value is read manually, separately at each node) and then to initialise a `transmitter` attribute `t[n]` from it. The knowledge of which attributes of `t` were initialised purposefully does not exist at the coordinator node, and therefore is inaccessible to the Compute Manager and cannot be managed in the Variable Registry.

In particular, the following is *not* managed in the Variable Registry.

- The data of elementary-type entities (i.e., the values and number of elements in them) does not exist in the Variable Registry,
- The Variable Registry does not keep track of whether `transmitter` attributes are initialised (i.e., refer to entities) and if so, which entities, nor even whether the attribute exists (as long as its name is in the scope of the `transmitter`); and
- There is no one-to-one correspondence between elementary-type entities in the Variable Stores and in the Variable Registry. (Although, if an elementary-type entity is the underlying entity for a particular qualified name whose longest prefix has a collection as its underlying entity then the same qualified name will be mapped in the Variable Registry to an elementary-type entity of the same type and metadata properties.)

The principles above can be described in far greater detail and specificity as follows:

- The Variable Registry keeps track of every non-elementary entity in the system, in the sense that there is a one-to-one mapping, m_n, from the non-elementary entities in the Variable Store on each node n to the non-elementary entities that are in the Variable Registry, and every non-elementary entity in the Variable Registry is mapped from a non-elementary entity in at least one of the Variable Stores. The mapping preserves the entity's type and properties (both attributes and metadata).
- The Variable Registry keeps track of every attribute of every collection in the system, in the same sense that if collection c on node n maps to $m_n(c)$ in the Variable Registry, then every attribute of that collection on n is mapped to a same-named attribute of $m_n(c)$ in the Variable Registry. This provides a similar one-to-one mapping, for variables.
- Where a variable v on node n that is an attribute of a non-elementary entity c_1 refers to a non-elementary entity c_2, in the Variable Registry the same-named attribute of $m_n(c_1)$ refers to $m_n(c_2)$ and has the same properties as the properties of v that are shared throughout its scope. In other words, if the non-elementary entities in the Variable Registry and the variables connecting them are taken to be an attributed digraph, the mapping m_n is an attribute-aware mapping of a subgraph into n's Variable Store data.
- Where a variable v on node n that is an attribute of a collection c refers to an elementary entity e, in the Variable Registry the attribute of $m_n(c)$ with the same name as v also refers to an elementary entity, and this entity shares all of the properties of e that are shared throughout the scope of e (with the exception that if e is a shared entity, the actual value of e is not shared with the Variable Registry).
- The Variable Registry also keeps a record of all variables that are `transmitter` attributes in the system, but this record is incomplete in the sense that the Variable Registry does not know whether any such attribute is initialised or not (i.e., whether it has an underlying entity) and whether the attribute even exists (The Variable Registry assumes that if node `n` is in the scope of `transmitter t`, then the attribute `t[n]` exists). Some of the metadata for these `transmitter` attribute variables is consequently placeholder data, describing the

`transmitter` rather than its individual attributes. For example, the `scope` of a `transmitter` attribute is listed as the scope of the `transmitter`, not the scope in which this attribute was actually initialised.

- The Variable Registry keeps track of metadata for entities referred to by `transmitter` attributes, but, again, these entities may or may not actually exist in the Variable Stores. The semantics of FTIL are such that all Variable Registry metadata for `transmitter` attributes can be derived from the properties of the `transmitter` itself, so these placeholders describe what the information would be if the attribute is initialised, without keeping track of whether it really is initialised.

- Every variable in every Variable Store must have at least one qualified name that refers to it (and potentially an infinite number of them). The same qualified name that leads to any referential entity r on node n leads to $m_n(r)$ in the Variable Registry, and vice versa.

The few items in the Variable Registry that are not information shared among all nodes in scope, and must therefore be stored in the Variable Registry on a per-node basis, are properties of entities and of variables that are mirrored from the Variable Stores to the Variable Registry through the same mapping as other properties, except that their values may be different from node to node. The reason they are kept in the Variable Registry is that they are integral to effective prerequisite checking on the coordinator node.

In this section, we will go in detail over what information the Variable Registry holds.

3.8.1 Entity metadata properties

Figure 3.2 summarises which types of information exist as entity metadata in the system for entities that are neither pseudotypes nor strings, and is accessible programmatically. The table details how to access it.

Figure 3.2: Entity metadata (excluding pseudotypes and strings)

For entities of type...	attribute...	of type...	means...
All	`rm`	`rm`	Entity type
Elementary	`rm`	`rm`	Is it shared?
All collections	`rm`	`rm`	Unique collection ID
Labelled collections	`rm`	shared string	Collection label
Module proxy	`rm`	shared string	Signature of the module

Additionally, for labelled collections, the Variable Registry also includes the information of whether the collection's label is **suspended** (See Section 3.10.3). This flag is maintained separately per node, and is not accessible to the FTIL user.

Entities that are pseudotypes or string types do not have a `metadata` attribute and generally don't export any metadata. What metadata on them exists in the system is implementation dependent. Exceptions are methods, modules and `type`s, which do provide the information about themselves given in Figure 3.3 through "dot" syntax, although they do so directly, not through a `metadata` attribute.[18]

Additionally, if introspection is supported (which is a Priority 2 ask), entities of type `type` have all of their other attributes, detailed in Section 3.5.5, in the Variable Registry as well, and these are accessible by the Compute Manager programmatically. In this case, as convenience features, if `v` is an `array` then "`v.size()`" is syntactic sugaring for "`v.metadata.type.size`".

[18] Whether one wants to consider this data or metadata is a matter of definition, but the use of shared strings indicates metadata.

Figure 3.3: Metadata for pseudotypes

For entities of type...	attribute...	of type...	means...
Method	`rm`	shared string	Name of script
	`rm`	variable	reference to module
Module	`rm`	shared string	Signature of the module
Type	`rm`	shared string	name of the type

Lastly, every entity managed in the Variable Store, including entities that are pseudotypes, has a **Variable Registry reference count**. This is not accessible from FTIL and is distinct from all reference counts managed in the Variable Stores (regarding which see Section 3.9.1). The idea is that just like each Variable Store separately must keep track of the number of references to each entity so as to deallocate it once it is no longer referenced, so does the Variable Registry need to keep track of references to each entity within itself, so as to deallocate entities once they are no longer needed. The Variable Registry reference count is an upper bound for all other reference counts, so the Variable Registry is guaranteed to be the last to deallocate any entity.

3.8.2 Entity Variable Registry data

Figure 3.4 lists data (not metadata) that is available in the Variable Registry and can be used for flow-control decisions, scoping decisions, indirect addressing into catalogues and prerequisite checking, even when it pertains to an entity whose scope does not include the coordinator node.

Figure 3.4: Variable Registry data

For entities of type...	The Variable Registry includes...
Collection	Their attributes (including methods)
`rm`	Their attributes (See note)
Module	The module file text (possibly also some parsed versions thereof).
Script	The identity of the user who defined the script. The text of the script.

Note 3.8.4. The information of the identity of the user who defined the script is only required for the long-term-persistent version of the Variable Registry. At any point in time, only the scripts belonging to the currently active session's user are needed.

For `catalogues`, the Compute Manager can also use their `.keys()` and `.size()` methods (e.g., for flow-control decisions). As a reminder: these are shared values, which are already available at the nodes at the scope of the entity, so using these to manipulate the entity does not require privileged execution.

Note 3.8.5. For entities `t` of type `transmitter`, the Variable Registry includes attribute information for all `t[n]` where `n` is a node in the scope of `t`. However, the actual variable information in each of these attributes does not include which entity `t[n]` refers to, whether it is initialised, or whether it even exists. All it includes is information that can be derived from the `transmitter`'s own metadata.

The Variable Registry also holds the following other data, to be used by the Compute Manager for general housekeeping.

- The state of the current execution stack;

- As part of the above, at each execution stack level: the current program counter position;
- At each execution stack level: the value of any `for`-loop variable;
- The scoping stack (to allow proper scope-unnesting when finishing any execution block); and
- Any other information necessary to properly handle end-of-block activities. For example, whether the block is a `modifying` block, or, if the block is a `for` block, what variable the `for`-loop variable is.

All of the above is reflected also in the relevant Variable Stores, as most Variable Registry data is. However, there are also cases of information that is needed simply for the basic operation of the Compute Manager, and are among the cases of information that exists in the Variable Registry but does not need to be reflected in any Variable Store.

Examples of these are:

- Full information for non-module scripts, stored on a per-user basis, and
- The state of the execution stack and program counters as at the last save point.

Note 3.8.6. Regarding the latter item: while all Segment Managers create a save point synchronously (and if necessary all Segment Managers roll back to save points synchronously) there is no need for any Segment Manager to store the state of the execution stack as at the time of saving. The only reason this is stored in the Variable Registry is for reporting to the user if execution aborts and a roll-back is necessary.

The above is not intended as an exhaustive list. Mentioned here are the parts of the Variable Registry entity data with largest direct impact on the FTIL user. Any additional data required for maintaining the basic functionality of the Compute Manager would also be saved here.

3.8.3 Variable properties

There is only a single variable property that is available to the FTIL user. This is the variable's **scope**. It is available as `.metadata.scope` for variables that are initialised and have an entity type that supports the `.metadata` property. The "`.metadata.scope`" attribute is of type `nodeset` and reports the variable's scope. In the case of attributes of a `transmitter`, it reports the scope of the `transmitter`. (See Section 3.5.4 for an explanation of this special behaviour of `transmitter`s.)

Using the Compute Manager, FTIL variables keep track of several other properties that are not accessible to FTIL programmatically in any direct way, but do influence prerequisite checking. These are:

Entity: Each variable knows what entity it references. It may not reference any entity, in which case it knows that, too. The exception is `transmitter` attributes, for which this information only exists in the Variable Store. (Whether or not a variable references an entity *is* available to the FTIL user, namely through the "`initialised()`" function, but with some caveats. See Section 3.11.20.)

Attribute: Every variable retains whether it is an attribute of another variable (which may be a collection or a `transmitter` type, or the variable may be the "`.module`" attribute of a method).

Dependent: If it is an attribute, the variable retains whether it is a dependent variable. A dependent variable is a variable that is an attribute of a `transmitter` or a `catalogue`, or is the module reference of a method.

Type: If the variable is dependent, its parent variable determines what type entities it can hold, but the variable holds this information. This is needed if the variable is uninitialised (without an underlying entity), as may be the case with `transmitter` attributes, or as may be the

case when `catalogue` attributes are sent as uninitialised parameters to scripts. If the parent entity is of type `transmitter<T>`, the variable's type is `T`. If the parent is of type `catalogue`, the variable's type is `object`. If the parent is of pseudotype "method", the variable's type is the pseudotype "module".

Protection counter: All variables maintain this integer. It is separately maintained per node. (See below for a discussion.) A variable is considered *protected* if its protection counter is nonzero, in which case it cannot be deallocated or re-referenced.

Immutability: For variables referring either to distributed elementary-type entities or to `transmitter`s, this Boolean flag, separately maintained per node, determines whether the variable is immutable. (See below for a discussion.) If it is "true", the variable cannot be used to modify its underlying entity.

Variable Registry reference count: All variables maintain an integer counting how many identifiers reference them. (The Variable Stores keep their own reference count for each variable, too, but these may differ from the Variable Registry reference count.) When the reference count drops to zero for a variable that is not an attribute, that variable is deleted. Because the Variable Registry reference count is an upper bound for the corresponding reference counts in the Variable Stores, the Variable Registry will only delete an unreferenced variable after it had already been removed from all Variable Stores.[19]

Note 3.8.7. The "attribute", "dependent" and "type" properties of the variable are all determined at the variable's creation, and are unaffected by any later assignments. The semantics of the language ensure that when a referential entity gets deallocated, all of its attribute variables are deallocated as well.[20] For this reason, none of these attributes can change throughout a variable's life-span.

Note 3.8.8. The immutability flag is not maintained for collections because for collections, whether or not an entity can be modified is a function of the entity, not of the variable referring to it, and is determined by whether or not the collection has a label[21].

Note 3.8.9. Despite the fact that both the Variable Registry and the Variable Stores keep reference counts, and despite the fact that these reference counts are maintained for both variables and entities, it is still the case that a garbage collection mechanism will be needed in order to remove circular reference chains that are not connected to an identifier. Such a mechanism should exist both for the Variable Stores and for the Variable Registry.

When an elementary variable `u` is created (or re-referenced) using

```
u = v
```

or, e.g.,

```
u = v.attr
```

i.e., when the assignment is dynamic and executes successfully, immutability and protection do not get inherited. Immutability of `u` (if the property exists) starts at "False", its protection counter starts at zero and its scope becomes the scope of the command. (The fact that assignment was successful indicates that `u`'s original protection counter was at zero.)

Static assignment does not change a variable's protection or scope (and leaves the variable non-immutable).

The reason for the difference in the handling between protection status and immutability status is that there are different reasons why a given variable can be protected but only one reason why it may be immutable. Specifically, if `x` has an immutable flag and that flag is set to "True" then

[19] Ignoring garbage collection.

[20] Advanced note: if an identifier exists that refers to one of the variables, the parent entity must appear on some Protected List, as explained in Section 3.9.3, so is protected and cannot be deallocated.

[21] Or is inside a "`rm`" clause that modifies it, regarding which see Section 3.10.3.

- If `x` is a top-level variable then it is a system constant,
- If `x` is an attribute of a collection then the collection is labelled and its label is not suspended, and
- If `x` is an attribute of a `transmitter` which is an attribute of a collection then that collection is labelled and its label is not suspended.

Because whether a label is or is not suspended may differ from node to node, this flag must be kept separately for each node.

Note 3.8.10. If a variable has an immutability flag and that flag is set to False, it is always possible to modify the underlying entity (e.g., via sub-assignment, as described in Section 3.2.7).

For variables that do not have an immutability flag, however, there are still a variety of reasons why the system may protect their underlying entity from modification. The underlying entity may be "shared". It may be of type `string` or of a pseudotype that does not allow modification. Or it may be a collection that is immutable because of its label status.

Keeping track of a variable's protection status is a little more involved, because there are multiple reasons that may make a variable protected:

- It might be a system constant,
- It might be a `for`-loop variable,
- It might be an attribute of a labelled (non-suspended) collection,
- It might be an attribute of a transmitter that is an attribute of a labelled (non-suspended) collection,
- It might be an antecedent to a parameter in a script call on the stack,
- It might be the calling `object` in a method call on the stack.

Instead of keeping a single Boolean "protected" status, we keep a counter that tells us "for how many reasons is this variable kept protected". As reasons come and go, we increment and decrement this counter. Only when the counter is at zero can the variable be deleted or reassigned.

Specifically, we require such a counter per node, because whether or not a label is suspended may be different from node to node, and only if all of the variable's protection counters are at zero can it be deleted or re-referenced to a new entity.

3.8.4 Identifier properties

Identifiers are the top-level names in the system.

Each level on the FTIL execution stack maintains a **mapping from top-level names to variables**. The identifiers are this mapping. They carry no additional information. The mapping is separate for each level on the execution stack. Within each level on the execution stack (including at the top, command-line level), the names in the mapping are all distinct.

Note 3.8.11. To make sure the names always remain distinct, as required, FTIL uses the same mechanism as the one described in Section 3.5.3 in the context of keeping `object` attribute names distinct.

Note 3.8.12. Scripts on the execution stack may not be distinct: if the same script is called twice recursively, this counts as two separate levels on the execution stack, and each level maintains its own index.[22]

Recall that the mapping from names to variables may not be one-to-one even within a single level on the execution stack. This happens when the same variable is passed more than once as a parameter to a script.

[22] Rule number one of designing a modern computer language: 'Don't be Fortran'.

3.9 Variable Store metadata

As discussed, most of the information in the Variable Registry is also mirrored, in one way or another, in the individual Variable Stores.

However, the system also keeps track of various bits of information about its entities/variables/etc. that is unique to the Variable Stores and does not mirror any Variable Registry information. This is additional data that enables individual Segment Managers on the individual nodes to verify the legality of instructions and to properly interpret the semantics of commands. This data, stored in the Variable Stores, may be different from node to node.

None of it is directly accessible programmatically to the FTIL user.

The following types of information are stored in the Variable Store.

- Entity data and metadata,
- Variable properties, and
- Qualified name lists.

This section discusses these.

Note that all variable/entity properties that exist in the Variable Registry *also* exist in the Variable Store, on those nodes where the variable/entity is in scope. These properties have already been discussed in Section 3.8 and will not be repeated here. For data items that are stored on the Variable Registry "per node", each node only sees its portion of the data in its own Variable Store. The only exceptions are the two Variable Registry reference counts (one for entities, one for variables), which are not mirrored in the Variable Stores at all. Variable Stores store their own reference counts, for their own internal bookkeeping.

The key differences between the Variable Store and the Variable Registry are

- The Variable Store only keeps information that is in scope for the node; the Variable Registry keeps information about all variables/entities in the system.
- The Variable Store contains both data and metadata; the Variable Registry does not include the main data in the system (this being the value and number of elements in elementary-type entities).

3.9.1 Entity data and metadata

Figure 3.5 summarises which types of information exists as entity metadata in the Variable Store for entities that are neither pseudotypes nor string, in addition to those already discussed in Section 3.8.1.

Figure 3.5: Entity Variable Store metadata (excluding pseudotypes and strings)

Entities of type...	retain property of type...	to mean...
All	integer	Reference count to the entity
Labelled collection	Boolean	Is label strong or weak?

The Variable Store also keeps for elementary-type entities their **elements**, i.e. their payload. This is the system's main data. Everything else in the system is there just to manipulate it.

3.9.2 Variable properties

FTIL variables keep track of several properties that are not accessible to FTIL programmatically in any direct way, but do influence prerequisite checking. Of these, the properties stored only in the Variable Store (which are all potentially different from node to node) are:

Entity: For variables that are attributes of `transmitters`, each variable knows what entity it references. It may not reference any entity, in which case it knows that, too. (For other types of variables, this information is part of the Variable Registry.)

Reference count: Each variable knows how many identifiers reference it.

All other variable properties that are stored in the Variable Store are also mirrored in the Variable Registry. These have been discussed in Section 3.8.3.

3.9.3 Qualified name lists

Recall from Section 3.2.6 that a *qualified name* is for our purposes always one that refers to a variable and an underlying entity (never an element), and always one where any indirect addressing has been fully resolved. In other words, it can never be "`a.b.c[i].d`" or "`a.b.c[i - 17].d`" or "`a.b.c[3 + 4].d`". All of these must first be resolved to something like "`a.b.c[7].d`" before they are considered qualified names.

Here are the qualified names lists maintained by FTIL per execution stack level.

Confirmed Name List: This is a mapping from qualified names to labels (strings) that determines which qualified names have been label-confirmed, whether explicitly or implicitly, during the course of the script's execution so far, and to which label they have been confirmed. See Section 3.10.2 for a full description of how this list is used and how it is maintained.

Modified Name List: We keep track of all qualified names that are being used as arguments in currently-running "`modifying()`" blocks. See Section 3.10.3 for a full description. This is a list of qualified names in which the qualified names are added or removed in a stack-like fashion. Read access to the list is, however, in random order.

Protected List: When a script is called, the qualified name of each sent parameter is stored. If this is a method call, the qualified name of the calling object is stored as well. Items in this list are indexed by the name of the script parameter they are mapped into, so they can be matched against their respective identifiers on the next level of the execution stack. This list is used and emptied once the called script returns. (See Section 3.13.2 for details.)

Note 3.9.1. It is tempting to want to store these three lists not as qualified names but rather in terms of the qualified names' underlying variables. That, however, does not work for any of the three lists. The Confirmed Name List allows its underlying variables to change. The Modified Name List describes a per-name behaviour: the same variable may behave differently if referenced in two distinct ways. Lastly, the Protected List takes part in updating the Confirmed Name List, so must be maintained as a list of qualified names, too.

3.10 Labelling

For entities of collection type that do not already have a "`.metadata.label`" attribute (and, in particular, that are not already module proxies), FTIL supports the command "`rm`".[23] It uses the following syntax:

```
label(v, "Label Name{Signature}")
```

Labelling is one of the most important and most unique features of FTIL.

The idea of assigning a label to an `object` instance is to ascertain that it syntactically follows a particular interface and semantically represents a particular type of information (e.g., due to how it was created) that makes it suited to a particular usage. Any potential user of the `object` instance can verify the label of the `object` instance before making use of it, by use of the built-in command

[23] All module proxies are labelled `rm`s.

```
confirm_label(v, "Label Name{Signature}")
```

The label itself is merely a string (stored in the entity's "`.metadata.label`" attribute as a shared string). However, in order to make sure that anyone labelling or using a label refers to exactly the same semantics, we utilise the internal structure `"Label Name{Signature}"` within the label text. This structure does not impact FTIL at all, but the idea is that there will be a register (somewhere, off the Purgles Platform) that will document for each "`Label Name`" its exact semantic and syntactic definition. This will be available in electronic form and will be circulated among all Purgles Platform participants.

The "`Signature`" is a hash, such as a SHA3, of the label name and the full definition describing it. In this way, any two parties using "`Label Name{Signature}`" can be sure that they agree not just on the name of the label but also on its exact definition. By convention, signatures are represented as lowercase, zero-padded, hexadecimal strings.

Note 3.10.1. Whether or not the Alerting Project provides a convenient tool to compute and verify this hash is beyond the scope of this document.

We do not require any particular algorithm for the hash that is used to compute this signature. All that is required is that it is a *cryptographically strong hash*, meaning that given an object o with a signature $h(o)$, it is cryptographically difficult to create a different object, o', that will satisfy $h(o) = h(o')$. Such a hash algorithm will be required for all other types of signatures used in the FinTracer system, too.

Labels in FTIL are either "weak" or "strong", depending on whether they were created in a privileged context or not. (This is a property of the entity, and may be different from node to node.)

Labelled collections also have a Boolean status maintaining whether their label is "suspended" or not. FTIL supports a directive, "`modifying()`" that allows one to alter a collection even after it has been labelled. A collection's label is "suspended" if and only if FTIL execution is currently inside a "`modifying()`" block that allows the collection to be modified. The label is considered suspended in such a case because during modification the collection may not satisfy the fitness-for-purpose conditions that are normally guaranteed by the label.

Like label strength, the value of the "suspended" flag may also be different from node to node.

Labelling in FTIL is composed of

- The `label` command, that initially creates the label,
- The `confirm_label` command, that inspects the label, and
- The `modifying` directive, that allows one to alter a collection after it has already been labelled.

We describe the semantics of each in turn.

3.10.1 Using "label"

Labelling a collection `v` is done using the following command.

```
label(v, "Label Name{Signature}")
```

The second parameter to `label` must necessarily be an explicit string literal. Variables cannot be used here.

This command has to be run in exactly the scope of `v`. Informally, what it does is assign the second parameter to "`v.metadata.label`", signifying by this that `v` is fit for the particular purpose that the label purports it to, and then "freeze" `v`, disallowing any alteration in it that may damage its fit-for-purpose. The only subsequent allowed modifications are inside a "`modifying()`" clause, which is FTIL's way of stating that the modification will not harm the entity's fit-for-purpose.

A labelled entity, `v`, has the following information in its metadata:

Its label: This is a shared string, available as `v.metadata.label`.

Whether the label is suspended: This is a Boolean that is not available through the `metadata` attribute, and may be different from node to node. It is initialised as `False` upon labelling, and only becomes `True` inside a `modifying()` clause where `v` is the argument given to the `modifying()` directive.

Whether the label is strong: This is a Boolean that is not available through the `metadata` attribute, and may be different from node to node. An entity's label is considered "strong" at a given node if at the time of labelling the node was in a privileged execution mode (See Section 3.15). Because each node may be in a different mode during execution, the strength of a label may vary from node to node.

Unlabelled collections have none of these `metadata` attributes.

Formally, executing "`label(v, "Label Name{Signature}")`", assuming that `v` is a collection and `"Label Name{Signature}"` is an explicit literal string, is performed through the following steps.

1. Verify that the variable `v` is not "protected" (or the command fails).

2. Verify that the underlying entity `v` does not have a label. (One cannot label a collection twice, not even if the current label is suspended, and one cannot label a module-proxy—these are always labelled before becoming module proxies.) The command fails if `v` is already pre-labelled.

3. For every attribute of `v` that is of a collection type, verify that the attribute has a label. (There is no requirement that the label be confirmed in the sense of Section 3.10.2, merely that the label exists.) These labels must not be suspended. Also, if the command is run in privileged mode, these labels must all be strong. The command fails if any of these prerequisite checks fail.

4. If the reference counter to the entity `v` is greater than 1, the entity is copied away at this time. (A new mini-registry is created; the variable `v` is redirected to the new mini-registry; all attributes of the original collection are shallow-copied to the new mini-registry, with the exception of `transmitter`-type attributes, which undergo semi-deep copying: the `transmitter` itself is duplicated, but all the `transmitter`'s attributes are shallow-copied. This process may cause the collection to change its "`.metadata.id`".)

5. If the entity had been copied, as above, the new value of the attribute `v.metadata.id` is determined at the coordinator node, and this value is updated in the Variable Registry and is communicated to all nodes in the scope of `v`, for all Segment Managers in scope to update it also in their respective Variable Stores. (This communication step does not require any special privileges.)

6. Attribute `v.metadata.label` accepts the label string. It stores it as a shared string.

7. The label is marked as not-suspended. If execution is in privileged mode it is marked as "strong". Otherwise: "weak".

8. The collection is now "frozen". *Freezing* a collection amounts to the following actions: (a) All of the collection's attributes are marked as protected by incrementing their protection counter; this is also performed on the attributes of any `transmitter` that is an attribute of the collection, and (b) All of the collection's attributes that are of distributed elementary types or are `transmitters`, and all the attributes of any such `transmitter` are all marked as immutable. (We refer to the reverse process, where all these are marked non-immutable and all the protection counters are decremented, as *unfreezing*.)

9. The qualified name for `v` which was used in the `label` command is now added into the Confirmed Name List, exactly as if a "`confirm_label(v, "Label Name{Signature}")` was run immediately after the `label` command. (See Section 3.10.2 for details.)

A collection that is labelled with a non-suspended label cannot be altered: no attributes can be added to it, removed from it or modified. The behaviour of collections with suspended labels is more complex, and is explained in detail in Section 3.10.3.

Note 3.10.2. The guiding principle at work here is that label guarantees are compartmentalised: the label of a collection guarantees that it is fit for purpose insofar as its non-collection attributes go, given that the collection-type attributes are all labelled, and their fit-for-purpose is guaranteed by *their* labels. For this reason, neither when labelling nor when modifying a labelled collection do we need to descend recursively into its sub-collections.

3.10.2 Using "confirm_label"

The command

```
confirm_label(v, "Label Name{Signature}")
```

follows a similar structure to the `label` command: its first parameter must be a collection, and its second parameter must be an explicit literal string.

Note 3.10.3. If one wants to confirm that a collection's label is either "`A`" or "`B`", one cannot do this just using "`confirm_label`". The way to implement such a check is

```
assert("label" in v.metadata)
if v.metadata.label == "A{Signature}":
  confirm_label(v, "A{Signature}")
else:
  confirm_label(v, "B{Signature}")
```

Note that this "`if`" statement will be resolved properly even if the scope of `v` does not include the coordinator node.

The purpose of label confirmation is that it allows a script certifier to know at certification-time exactly what code FTIL will execute in privileged mode at run-time, at least conceptually, even when that code involves, for example, calling the methods of an `object` that was received as input.

The way FTIL enforces this is by requiring the label of a collection `v` to be confirmed prior to the following operations:

- Running a method of `v`, and
- Running "`modifying(v)`".

The label confirmation requires the FTIL script-writer to actively specify what label they are expecting `v` to have, and by this what guarantees they are expecting it to hold.

The language does not enforce the use of `confirm_label` in other contexts, such as when accessing `v`'s attributes, but writers of scripts that perform potentially-sensitive operations should nevertheless use it to verify the nature of all inputs of unknown origins passed to their scripts before relying in any way on the semantics of the information these inputs are supposed to contain.

In principle, this sounds easy to do: before running either one of the operations requiring label confirmation, the user can state regarding the particular affected variable what their expected label should be; the executor should then confirm that the label exists, is correct, and is strong if confirmation is done in privileged mode (otherwise, any strength will do); and then the code can run.

Unfortunately, this leads to very cumbersome code, because everything must be confirmed and re-confirmed all the time. FTIL was designed to maintain the safety stipulated above while minimising as much as possible the need for explicit label confirmation.

To solve this problem, FTIL introduced the *Confirmed Name List*. This is a mapping, maintained separately for each position in FTIL's execution stack, from qualified names to labels. Conceptually, this is the list of qualified names regarding which the FTIL script-writer has stipulated what labels are to be expected in them.

The existence of the Confirmed Name List now splits the problem of label confirmation to two separate processes:

Adding a label to the confirmed list: Adding qualified names to the list, matched with their expected labels, and

Verifying a label: Verifying that an entity has a label, that the label is not suspended, that if the script is in privileged mode the label is strong, and that the label has its expected value.

While the `confirm_label` command is an explicit way to perform *both* activities, FTIL at various times invokes either process implicitly. For example, when one calls an `object` method, one implicitly first verifies the `object`'s label, whereas by running the command `label`, one implicitly adds the label to the confirmed list.

Importantly, label verification is done in FTIL solely against the Confirmed Name List, so the user never needs to stipulate the expected label solely for verification. It only needs to be stated when adding a label to the confirmed list.

Note 3.10.4. Notably, the Confirmed Name List lists qualified names, not variables or entities. FTIL allows the variables with these names to be modified and replaced between addition to the list and verification. Such modifications are not necessarily straightforward, because only labelled collections (which are necessarily "frozen") can be added to the list, but in any case where the meaning of a name changes, the list keeps track of the name, not the meaning.

This ensures that only what was explicitly added to the list is considered confirmed.

Informally, here are the processes that add a label to the confirmed list:

- An explicit `confirm_label()`,
- An explicit `label()`,
- The adding of any collection, `v`, to the list implicitly also adds any collection for which `v` is an antecedent, and
- If a script parameter is on the Confirmed Name List at the successful termination of its script, the corresponding qualified name in the calling script is automatically added to the list. (Note: It is not enough that the script parameter is an antecedent to a label-confirmed collection.)

Also, here are the processes that verify a label:

- An explicit `confirm_label()`,
- Calling an `object`'s method, and
- Modifying a collection.

Formally, the process is as follows.

- Whenever a script is executed, all of its parameters (including the calling `object`, in the case of a method call) have their qualified names added to the Protected List. (This is a mapping from formal—i.e., called—script parameter names to invocation—i.e., calling—script parameter names. It is explained in detail in Section 3.13.2.) Note that qualified

names, by definition, have fully-resolved indirect addressing values. (Here, this indirect addressing may involve both `catalogues` and `transmitters`.) It is important to resolve any indirect address because these qualified names will be used at the end of the script call, where it is important that the addresses will be to the same values as when the script was called.

- The FTIL executor maintains at each execution stack position the Confirmed Name List, a mapping from qualified names to labels, which starts off as empty at the start of script execution and is discarded at the end of script execution.
- When running "`confirm_label(v, "Label Name{Signature}")`", the executor verifies that `v` is a collection, that it has the correct label, that the label is not suspended, and if the executor is in privileged mode also that the label is strong. If any of these fail, the command fails and execution is aborted. Otherwise, the qualified name is added to the Confirmed Name List, with the label. Note that the qualified name is added with all indirect addressing resolved. If the name already exists on the List, its label is updated.
- When running "`label(v, "Label Name{Signature}")`", if this is successful, the name is also added to the Confirmed Name List with its label.
- When a script terminates successfully, if any of its qualified names on the Confirmed Name List is the name of one of the script's formal parameters (this necessarily meaning that the name is atomic), and if this identifier is associated with a variable of type collection that has the expected label and the label is not suspended and is strong if the execution mode of the calling script is privileged, and the variable that was sent to the script for that parameter, as it appears on the Protected List, is not already on the Confirmed Name List for the calling script, that variable, using its qualified name on the Protected List, is added to the Confirmed Name List for the calling script with the label which it now has.
- Whenever either a method of an `object`, `v`, is invoked or "`modifying(v)`" is executed on a collection, `v`, the FTIL executor runs the following label-confirmation checks, prior to proceeding with executing the actual command:
  1. We find the longest extended prefix of `v` that either exists on the Confirmed Name List or has an underlying entity whose label is marked as "suspended".
  2. If no such prefix exists, or if the found prefix has an entity marked as "suspended", confirmation fails and execution aborts.
  3. Otherwise, we implicitly run `confirm_label` on the found prefix, with the label associated with it on the Confirmed Name List.
  4. Only if this confirmation is successful (i.e., if the label exists, is correct, and is strong enough), do we go on to process the main command. Otherwise, confirmation fails and execution aborts.

A "`modifying`" directive also alters the Confirmed Name List in other ways, explained in Section 3.10.3, but these are meant simply to uphold the invariant that the collection-type attributes of a collection that is label-confirmed are also considered label-confirmed.

3.10.3 Using "modifying"

FTIL supports the directive "`modifying(v)`".

Like "`on()`", "`modifying()`" may be applied to a single command or may create its own command block, via

```
modifying(v):
  command
  command
```

It accepts exactly one parameter, of a collection type, which must not be a module proxy.

The parameter, `v`, must be a labelled collection with a non-suspended label. Its label is confirmed, as described in Section 3.10.2, as part of prerequisite checking. If label-confirmation fails, the directive fails and execution aborts.

Furthermore, as described in Section 2.11.4, "`modifying(v)`" is a privileged command that requires the following special conditions, all of which are also verified as part of prerequisite checking:

- If the scope of the command is the same as the scope of its parameter, `v`, every node within that scope must be executing in privileged mode.
- If the scope of the command is narrower than the scope of `v`, every node within the scope of `v` must have individually certified the presently-running script. (Note that this is an unusually strict condition: it is not enough for these nodes to have certified one of the calling scripts.)

The purpose of "`modifying`" is to allow the value of `v` to change, while retaining its fit-for-purpose declaration. At the end of the `modifying` block, `v` will return to its original label and will still be considered label-confirmed.

While such alterations are sometimes inevitable, it is imperative to not allow any accidental, unintended changes to `v` while inside a `modifying` block.

Ensuring that no such unintended alterations can occur is, perhaps, the source for most complications involving "`modifying()`". If another variable refers to the same entity, or even if the same variable as the `modifying` parameter has another identifier (name) associated with it, these others are not allowed to modify the entity.

To implement this, FTIL keeps a Modified Name List, which is the list of the qualified names that were used as arguments for any of the currently-running "`modifying()`" blocks. As `modifying` blocks can be nested, the size of the list at all times equals the depth of this nesting. Adding or removing an item from the Modified Name List is done in a stack-like fashion. However, access to this list, as will be demonstrated shortly, can be in arbitrary order.

Equally important is the fact that FTIL `modifying` blocks are atomic in the sense that if any command fails during a `modifying` block, the state of the system is rolled back to what it was prior to entering the outermost `modifying` block (or as late as practicable prior to it), before commencing the abort process (as described in Section 3.13.6). The state during execution of a `modifying` block is considered incoherent.[24]

From inside a `modifying` block, one can invoke scripts and methods, as usual, and this does not change the atomic nature of the block. However, one cannot `broadcast`, `transmit` or use a privileged form of scoping or flow control, because such actions, involving communication between nodes, cannot be rolled back. (Recall that privileged forms of scoping and flow control require an implicit `broadcast`.)

Technically, the way `modifying` blocks work is as follows.

At the beginning of a "`modifying(v)`" block, the FTIL executor performs the following.

1. It verifies that `v` is a collection and that it is not a module proxy.
2. It verifies the privileged execution conditions for `modifying`, as described above.
3. The qualified name `v` is label-confirmed as described in Section 3.10.2. (If successful, necessarily its label will be strong.)
4. The executor asserts that no identifier in the system, at any level of the current execution stack, references any attribute of `v`, nor any attribute of any `transmitter` that is an attribute of `v`. This can be done simply by checking that the reference counts for these variables are all zero.

[24] One way to ensure this atomicity property is to not have save-points during a `rm` block. Saving right before entering an outermost modifying block and right after exiting it is recommended.

5. For each attribute of `v` that is of a collection type, the executor now concatenates the name of the attribute to the qualified name of `v` to create a longer qualified name. Each of these qualified names is added to the Confirmed Names List with their present labels.

6. The label of the entity `v` is marked as suspended.

7. The qualified name `v` is added to the Modified Name List

8. The protection counter of all underlying variables on the extended prefix of `v` is incremented.

9. The entity `v` is "unfrozen", as described in Section 3.10.1.

The fact that an entity's label is marked as "suspended" (something that can only occur inside a `modifying` block), impacts how access to this entity's attributes is handled. Specifically, FTIL uses the following rule-set.

Whenever any variable is accessed in FTIL, it is accessed through its antecedents. One can consider this as serialised access: in order to get to "`a.b[7].c`" one must first go to "`a`", then "`a.b`", then "`a.b[7]`", and only then to "`a.b[7].c`". If in accessing any of these antecedents, in the process of reaching a variable, one goes through an antecedent whose underlying entity is label-suspended, the FTIL executor checks whether that particular prefix exists in the Modified Name List.

If the name is *not* on the Modified Name List:

- Only `metadata` properties can be accessed from this point. No data property can be used, nor can any underlying method be accessed (whether for setting, deleting or execution).

- The variable's behaviour is as though it was immutable: one cannot create any new attribute for the suspended entity.

Note 3.10.5. Elsewhere in FTIL, if a variable with an underlying entity of a collection type has an attribute that is a method or a collection, it will have that attribute throughout its entire scope. A `modifying` block is a unique situation where that invariant may break. This is not a fundamental structural break, however: from the point of view of the executor on that node, these attributes still exists. They are just inaccessible to that particular variable at that particular time. One can think of this as a form of "permission management".

If the name *is* on the Modified Name List:

- The variable ignores the "suspended" attribute and the entity's label, treating the entity as unlabelled (and therefore modifiable). It can freely make modifications on any items that aren't still immutable (like labelled sub-collections or attributes that are shared) and can delete or redirect any attribute that isn't still protected (like sub-collections whose attributes are also being used as arguments to a script earlier in the execution stack). The collection cannot be re-labelled while label-suspended, however.

- The variable can be used to traverse to attributes of the entity and beyond, thus reaching other variables. Importantly, once these variables are reached the test that no antecedent is suspended is not repeated: these downstream variables can, for example, be sent as parameters to a script where they will subsequently be modified under another name.

At the end of a "`modifying(v)`" block, the following occurs.

1. The executor marks the label of entity `v` as not suspended.

2. The qualified name of `v`, as stored at the top of the Modified Name List, is used to decrement the protection counter for all underlying variables along its extended prefix.

3. Subsequently, `v` is removed from the top of the Modified Name List.

4. For each attribute of `v`, `v.attr`, that is of a collection type, the executor confirms that `v.attr` has a label and that the label is strong (recalling that the current execution mode must be "privileged"). If any of these checks fail, execution aborts. Note that unlike elsewhere, we do not verify here that the label of `v.attr` is not suspended.

5. The executor re-"freezes" `v` (as described in Section 3.10.1).

Note 3.10.6. Because we do not verify at the end of a `modifying` block that all attributes' labels are not suspended, it is possible within a `modifying` block to create circular dependencies in labelled collections. (When a collection is initially labelled, it cannot have circular dependencies.)

The above description regarding how `modifying` operates may seem unnecessarily convoluted. However, its purpose is to ensure the main goal of allowing the `modifying` argument to alter its collection while disallowing any other way for this collection to be modified in the process.

To prove that this goal is, indeed, attained let us consider the situation case-by-case, according to the various potential `u` which might interfere with the value of `v` inside a "`modifying(v)`" block. For the sake of simplicity, we will assume here that this `modifying` block is not nested. However, readers are welcome to review also the case of nested `modifying` blocks to see that the same arguments work also there.

Case 1: Here, `u` is not an attribute of `v` but an attribute of one of its collection-type attribute (or farther removed)—In this case, `u` is part of a labelled collection, so is immutable under our assumption of no nesting of `modifying` blocks.

Case 2: Here, `u` is an attribute of `v`, or an attribute of a `transmitter` attribute of `v`—As part of our prerequisite checking process at the beginning of the `modifying` block, we have ascertained that `u` is not itself an identifier (because all such attributes have a zero reference count, and therefore no identifier identifies them directly), and that if `u` has a parent entity that is a `transmitter` that this `transmitter` is also not directly associated with an identifier. Therefore, we conclude that all access to `u` must be through the suspended collection. Only `v` can see the attribute `u`, however, because the entity of `v` is label-suspended.

Case 3: Here, `u` is another variable that references the same entity as `v`, or is another identifier that references `v` itself—In both cases, the name of `u` will not be on the Modified Name List, so it will not be able to see or modify any attributes of `v` other than viewing its `metadata`. It may dereference itself, but if it is not the same variable as `v`, that would not matter to `v`, whereas if it is another identifier that references the same variable as `v` the dereferencing will fail because `v` is protected.

Case 4: Here, there exists some "`v.t`" that is of `transmitter` type, and `u` is another variable referencing the same entity—This case is not possible, because `transmitter`s are not copyable (and where they do get copied they get semi-deep copied). There is never a situation where two variables can reference the same `transmitter`.

As a last point of order, now would be a good opportunity to explain the unusual privileged-execution conditions required for "`modifying(v)`", which were first introduced in Section 2.11.4.

Consider first the situation in which the scope of `v` equals the scope of the command. At each node, `v`'s label might be either strong or weak, and the execution mode might be privileged or not privileged.

Of the four possible combinations of label strength and execution privilege, two must clearly not be allowed. If a non-privileged script may be allowed to modify a strongly-labelled collection, this allows anyone to strongly-label any collection they choose. If a privileged script, on the other hand, attempts to modify a weakly-labelled collection, that label confers no fitness guarantees, and the script may end up making incorrect assumptions, with possible unintended consequences.

Therefore, the level of execution and the strength of labelling must clearly match.

The reason FTIL does not support non-privileged modification of weakly-labelled collections is that this creates a situation where if a node operator certifies the script generating the collection,

and by this *increases* the fitness guarantees provided by the collection, this may cause code to break. To avoid such a situation, only privileged execution of strongly-labelled collections is allowed.

Next, consider a situation where the "`modifying(v)`" command runs at a narrower scope than the scope of `v`. This is definitely something that FTIL wishes to allow, because in many cases it is necessary for individual nodes to alter their labelled collections (e.g., when a node replenishes its stock of encrypted zeroes, to use an example from Chapter 4).

Sometimes, however, the fitness guarantees provided by a label are global, rather than node-local, and when node n_1 makes a change in `v` this changes `v`'s fitness for its purpose from the perspective of another node, n_2.

In FTIL, certifying a script that contains a `modifying` block is a node's way of ascertaining that it is willing to have other nodes use the script in order to modify `v`, and this will not change `v`'s fit for purpose from its perspective, even when that node itself does not participate in the running of the script.

If a node wants to allow a particular script to modify a collection, but only when that modification is done globally, at the scope of the collection, the FTIL way of conveying this is to certify the scripts calling the modifying script, rather than the modifying script itself. It is good practice to wrap modifying scripts by a wrapper dummy script specifically to allow node certifiers this freedom.

3.10.4 Structure of a label

As described above, labels are merely textual strings. We describe them as having an internal structure, "`Label Name{Signature}`", but there is nothing in FTIL that enforces that convention or gives it any technical interpretation. The signature values reference information that is off the Purgles Platform.

A Priority 3 feature, meant strictly in order to reduce the amount of typing a command-line user needs to do in order to reference labels and the amount of checking of hexadecimal values a command-line user needs to do in order to verify labels (because signatures can be quite lengthy), is the following.

1. When labelling collections, the system will retain a directory of the signatures associated with each base "`Label Name`". If a label name already exists in the directory but with a different signature, labelling will fail. This directory will exist in the System Data.

2. If periodic pruning of this directory is needed, it will be done by a coordinator-side administrator.

3. Command-line use of `confirm_label` (including in non-module scripts and within "`exec()`" commands) may omit the signature from the label name. If the name exists in the directory, the system will complete the signature for the user automatically, before the command is executed.

3.11 FTIL commands

A good portion of the FTIL commands was already discussed, mainly in Section 3.5, and this is no surprise: the language is built around the manipulation of entities, and these manipulations are type-dependent.

In this section, we discuss classes of commands that were either not mentioned at all previously in this chapter, or deserve special highlighting.

We remind that in FTIL each command (with the exception of some locally-run commands like "`dir()`") is associated with a scope. All Segment Managers receiving the command must have in their scope all variables required to run the command. Also, if a command deletes (or reassigns)

a variable, the command's scope must exactly equal the original scope of the variable (with some exceptions, mentioned in our description of the `del` command. See Section 3.11.18).

By design, the scope of variables upon their creation is the scope of the command that performs the creation (with some added visibility to the coordinator node and some special handling of pseudotypes being exceptions).

In general, one can divide FTIL's commands to four categories:

1. Commands that are only available from the command line.
2. Commands that are available both from the command line and from scripts, but can only be used as *top-level commands*: if they do not return a value, they form an entire command; if they return a value, that value can only be directly assigned into a variable or element. We refer to commands that return a value as *functions*.
3. Functions whose return values can be used as input to other commands (which we refer to as *proper* functions) but which cannot be used inside mass operations (See Section 3.12).
4. Proper functions that are allowed inside mass operations. This is a subset of the proper functions. All proper functions allowed inside mass operations have fixed-sized inputs and fixed-sized outputs. We call proper functions that have fixed-sized inputs and outputs *simple* functions.

The majority of commands described in this section falls under the second of these categories. Regarding each command listed, if it falls under any of the other categories, we mention this explicitly. If no category is mentioned explicitly, the second category is the default.

Most FTIL commands are also non-privileged commands. We highlight specifically the privileged commands. Non-privilege should be considered the default. (We occasionally mention specifically regarding a command that it is non-privileged, simply to highlight this fact.) A summary of all privileged commands can be found in Section 2.11.4 and in Section 3.15.

3.11.1 Human input

There are instances in which an operator, either AUSTRAC-side or RE-side, needs to input data into their respective Segment Manager. This can be facilitated by

`rm`: This (non-privileged) command returns a distributed variable of type `T`, in response to a prompt to the operators of the nodes that are in the command's scope, with "`s`" being the string prompting the user to provide a specific input. Each operator's input populates the part of the distributed variable that resides in the operator's own node. The type `T` must be serialisable, but not all serialisable types can be constructed in this way. For example, it is not possible to create any of FTIL's special cryptographic types in this way, whether they are ciphertexts or key types (public or private). This is because the existence of special types indicates that these types hold specific guarantees (e.g., they represent elements on an elliptic curve, or their `Cipher<T>` type really came from encrypting a valid `T`.) Because when reading values from an external source FTIL has no means of ascertaining whether the object was generated appropriately and therefore meets its guarantees, any type that provides some guarantees for the user regarding a fit for purpose cannot be used here. We refer to variable types that can be read from an external input as *constructible types*.

The exact process for prompting the RE-side operator and for receiving the input involves many complexities that are not described here. They are detailed in Section 6.1.

There is no special command for reading a shared variable, nor is there a special command for sending a separate prompt string to each operator in reading a distributed variable, although both these effects can be worked around programmatically. (Precisely how to do this is left as an exercise for the reader.)

Examples of places where this command can be used:

- In reading an explicit list of accounts from the RE users.
- In reading an explicit list of accounts from the AUSTRAC user.

For the purposes of the definition of FTIL we do not see any difference between reading a variable from an RE-side operator and reading a variable from an AUSTRAC-side operator. In practice, the mechanisms for such reads may be different.

3.11.2 Transaction Store input

A different type of input is when each Segment Manager is requested to retrieve data from its respective Transaction Store. This is done by

`rm:` The parameter "`conditional`" is a conditional that can be evaluated against the list of all transactions visible to the RE (So, all transactions in which an account managed by the RE is a party) that occurred within the date range [*start_ date*, *end_ date*]. These are transactions stored in the Transaction Store on the RE's peer node[25]. The Transaction Store table columns are arranged by "payer" and "payee", so accounts managed by the RE can be at either column. For any pair of accounts `(x, y)`, the conditional evaluates to either "True" or "False" as a function of all transactions from `x` to `y` and all transactions from `y` to `x`. It cannot use any other information. The function returns, in a variable of type "`set<pair<AccountAddr, AccountAddr>>`", those `(x, y)` pairs for which the conditional evaluated as True. Note that the pairs are directed: the existence of `(x, y)` does not exclude the existence of `(y, x)` in the returned set. It can be safely assumed that if no transactions occurred between `x` and `y` in either direction, the conditional evaluates to False. TransactionsRead is a non-privileged command.

The idea behind this somewhat complicated definition of TransactionsRead is that if both the RE managing `x` and the RE managing `y` see the same set of transactions between the two, both will agree on whether `(x, y)` should be in the list.

Note 3.11.1. The parameters `start_date` and `end_date` are mainly there to support a situation where the Transaction Store is populated via a "pull" mechanism, in which case only the transactions within the relevant date range need to be pulled. If we are confident that FinTracer will only ever be a "push" system, a single-parameter version of `TransactionsRead` will suffice. We do not recommend this, however, as it reduces the future-proofing of the system.

Regarding the three-parameter version, note that FTIL does not have special types for managing dates. At present, we envision `start_date` and `end_date` to be given as strings. This reduces, perhaps, the programmatic flexibility of the date range, but makes FTIL's type system simpler.

Note, however, that in a one-parameter version of `TransactionsRead`, the date range information will have to be communicated as a string in any case, even if FTIL supports a date type.

3.11.3 Auxiliary DB input

Reading from the Auxiliary DB is similar to reading from the Transaction Store, with the main difference being that the Transaction Store follows a well-understood schema and FTIL access to it is very controlled. By contrast, the Auxiliary DB is mostly taken to be a generic SQL database.

FTIL supports the following functions for reading from the Auxiliary DB.

`rm:` The parameter, `query`, is a generic SQL query that gets executed against the Auxiliary DB. The result produced is of type `list<T>`, created by converting every returned element to a `T` type. The type `T` must be a constructible type, as defined in Section 3.11.1. Note that the Transaction Store table is part of the Auxiliary DB, so can be queried (but not altered) as part of the query.

[25] Or are ones that are pulled into the Transaction Store when the command is executed, if the Purgles Platform were to use a "pull" model for transactions.

rm: Same as `AuxDBReadList`, except that `T` must be a hash type, and each result from the query is first hashed before it is returned as a `list<T>`. The reason this requires a special command is that it is not always possible to simply return the value and then hash it in FTIL, because sometimes the returned value does not have a fixed length. For example, it may be a string. An alternative potential workaround would have been to compute the hash directly in the database, but even when this is feasible, FTIL has no way of telling whether the result is properly hashed. Here, the result returned from the database is not hashed, but FTIL hashes it itself, and can by this guarantee its nature. Importantly, the hash function used should be (a) documented, (b) cryptographically strong [See Section 3.10 for a definition], (c) available also from SQL as a database function, and (d) where input types allow this, available also from FTIL. The last two conditions are important because they allow `HashRead` results to be compared against other hashes, which have been calculated by other means.

rm: Same as `AuxDBHashReadList`, except that in this case the hash used is a keyed hash. The extra parameter `key` provides the key to be used. Note that this function is not templatised: the returned type is fully determined by the type of `key`.

rm: Same as `AuxDBReadList`, but returns `set<T>` instead of `list<T>`. Even though lists are closer to the native SQL return format, we expect that mostly the return types of interest to the FTIL user will be `set`s. By providing this as a built-in command, we are avoiding data copying as part of the conversion process from `list` to `set`.

rm: Same as `AuxDBHashReadList<T>(query)` but returning a `set` instead of a `list`, for the same reasons as explained above.

rm: Same as `AuxDBKeyedHashReadList(query, key)` but returning a `set` instead of a `list`, for the same reasons as explained above.

3.11.4 Output

At the end of computation, an output functionality should allow retrieval, by the AUSTRAC-side operator, of variable values visible to the coordinator node. Several commands support this. What these commands look like from the perspective of the operator is discussed in Chapter 6. Here, we consider their interface to the FTIL user. The output commands are not privileged operations.

rm: Displays the local-node value of `v` to the human operator of the coordinator node. The type of `v` is assumed to be either a fixed-sized, serialisable type or a list thereof. If the result is a long list, it is truncated to include only the first few (say, 20) results, The size of the full list is also displayed. The variable `v` may also be a string, script, signed script or module. In the latter case, the module text is displayed.

rm: Serialises the local-node value of `v` and sends it out to an external coordinator-node-side system. The variable `v` can be of any serialisable type, as well as a string, script, signed script or module type (for the module file's text). Unlike "`display`", "`export`" performs no truncation.

rm: The `inspect` command accepts, as a string, the signature of a module. It retrieves from each node in scope which of the module's scripts are presently certified, and displays this information to the coordinator-side user.[26] The `inspect` function can only be run from the command line.

The commands "`display()`" and "`export()`" are both run locally on the coordinator node. They may be executed in any scope that includes the coordinator node, but the command is executed as though its scope was only the coordinator node. It is an error to try to execute these commands in a scope that does not include the coordinator node.

[26] This is really a convenience function, because the retrieved information is all in the Auxiliary DB, and can therefore be queried directly.

The command "`inspect()`" can be executed in any scope, and returns results from each node where it was executed. The module being queried does not have to be known to the node or presently loaded into the system. If it is not known, the node simply does not return any results. (A node may hold certificates for a module that was never loaded on it. In this case, the certificates are still part of the Auxiliary DB and should be returned by `inspect`.)

The output of "`inspect()`" is displayed solely to the coordinator node's FTIL user, regardless of the scope in which the command was executed, even if that scope does not include the coordinator node at all.

Note 3.11.2. FTIL was designed from the ground up to provide privacy preservation guarantees in everything it does. In actual intel usage, however, such privacy preservation is often not wanted. An intel analyst may want to use the FinTracer system in order to pinpoint certain accounts of interest, but the analyst's immediate follow-up steps will be to request from the relevant REs additional information, e.g. about account owners, which the FinTracer system has no business knowing: such information has no use on the coordinator node as part of any privacy-preserving protocol, and preventing the coordinator node from being able to receive it is part of FTIL's design.

Presently, including for the Purgles Platform MVP, we expect such follow-up steps to be taken off the Purgles Platform, by use of the traditional notice mechanism: the intel analyst should use "`export()`" to export from the FTIL system their accounts of interest, and then use this information to fashion notices for the relevant REs.

In future, beyond the MVP, the Purgles Platform may host a second tenant, besides the FinTracer system, which will be a (non-privacy-preserving) system that serves purely for automation and speed-up of the traditional notice mechanism.

The definition of any such system is beyond the scope of this document. We do note however that

1. The definition of such a system will probably require it to be able to receive outputs from an "`export()`" or even to automatically initiate an "`export()`" operation (and perhaps other FTIL functionality, as well).

2. Such a system will probably need to run SQL queries over data that exists in the Auxiliary DBs at the peer nodes. In the present architecture of the FinTracer system, these data repositories are completely decoupled from the FTIL engine, and there is no reason why other clients should not be able to query them, if all proper cybersecurity mechanisms and governance structures are in place.

Note 3.11.3. In past conversations with the REs, they have raised the issue that they, too, want visibility into anything that the Purgles Platform finds. All our privacy-preserving algorithms were designed with this in mind. So, for example, any result from the system is first exposed to the relevant REs, and then communicated by them to AUSTRAC. However, REs have voiced the concern that this may not be enough, because just the fact that certain information is on their local node does not mean it is reasonably accessible to them.

This is a fair point, and it can be addressed by providing versions of the three output functions presented in this section also to (non-FTIL) users of the peer nodes. Such versions may have the following properties.

1. They will only work on data that is on the node local to where the command was invoked from.

2. They will work in what would be equivalent to "command-line" operation, e.g. in the sense that the qualified name `v` will have to be a name at the top level of the execution stack, as would have been accessible to an FTIL user at the command line.

3. They will work against the persistently stored data, so will only be able to query the system while in a coherent state.

4. Potentially, such querying may also need to be restricted to times at which the FTIL engine is idle.

Providing any such programmatic interface to the RE-side user (which would present significant complications over the existing architecture) is presently beyond the scope of the Purgles Platform MVP and describing the details of it is beyond the scope of this document.

We will note, however, the following.

1. By design, any information that exists on a node is in-principle "exposed" to the owner of that node. If each node owner examines every bit that exists on their node, that should not have any privacy implications whatsoever.

2. Despite the above, we recommend for any such additional output functionality to exclude private keys (as was also done in the AUSTRAC-side output functions), because once private keys are exported from the system, FTIL can no longer guarantee that they will not be exposed to *other* parties, an action that may compromise the security of the protocols in ways that impact *all* parties. While a similar argument can be made regarding leakage of any data in the system, we believe that with private keys this extra layer of security is merited.

3.11.5 Preempting execution

FTIL includes functions that allow one to sanity-check execution at various points, which abort execution if their conditions are not met.

`rm:` Verify a condition or fail. Useful in programmatic prerequisite checking. The condition is evaluated separately on each node in scope and is considered to have failed verification if there is any node where the condition was evaluated as "False".

`rm:` Verify a condition or exit. The command exits rather than fails, causing later on aborting to include variable clean-up. See Section 3.13.6 for details. This command is used for automatically verifying policy parameters. The condition is evaluated separately in each node in scope. If there is any node where the condition is not met, execution aborts and `message` is part of the return message displayed to the coordinator-side user. If display of the message is required, the message is evaluated entirely on the coordinator node. At no point does any node other than the coordinator receive the value of the evaluated `message`, if it is not an explicit literal string. The `message` parameter is optional.

`rm:` Request a confirmation from the AUSTRAC-side operator before continuing. Like "`limit`", the command exits (rather than fails) if the confirmation is not given. This command is useful in scripts. For example, a script may display to the AUSTRAC-side user the number of results returned by a query and request a confirmation before the results are actually retrieved and output. This command runs only on the coordinator node. It can be executed in any scope that includes the coordinator node, but will execute as though only the coordinator node was in its scope. It is an error to run it in a scope that does not include the coordinator node.

3.11.6 Inter-node communication

There are two FTIL commands that explicitly transport information between nodes.

`rm:` This command operates on a variable `t` of type `transmitter` and must be run exactly at the scope of the `transmitter`. Any variable that was originally stored in `t[n1]` on node `n2` is at the end of the process stored in `t[n2]` on node `n1`. This is considered a "write" operation, in the sense that any attribute of `t` that has a reference count greater than one gets copied-on-write in the process. Running `transmit` requires all nodes in scope to be running in privileged mode.

`rm`: The `broadcast` command is a somewhat unusual command, in that its operand, `b`, is not necessarily defined across the scope of the command. This command operates on any serialisable variable `b` whose scope includes the coordinator. Only the value of `b` at the coordinator is used. The command creates a new entity of the same type as `b` but shared, that receives the same value as `b` but is defined in the scope of the `broadcast` command. Running `broadcast` requires the coordinator node to be executing in privileged mode, but does not require this of any other node. The scope of the `broadcast` does not have to include the coordinator node. (This means that while the `broadcast` command might not have the coordinator node in its scope, the script from which it was executed necessarily does.) Note that the nodes receiving the value of `b` may not be aware of the existence of such a variable prior to the `broadcast`, and are just receiving the original variable's value, not any other properties of it. FTIL's communication protocol will, however, communicate `b`'s type to the receiving nodes. The result of `broadcast` should be immediately saved to a variable. As usual, this variable will not be protected.

Neither `transmit` nor `broadcast` can be used to communicate any non-serialisable type, including any type that is a private key or a container of private keys.

For a review of all commands that require any form of privilege from the FTIL executor, see Section 2.11.4 and Section 3.15.

3.11.7 Random values

Typically, the variable types described in Section 3.5 are initialised from nothing, from rearranging existing data, or from computing functions over existing data. Another case, which we highlight here, is the creation of objects from randomness.

`rm`: This proper function initialises a distributed integer as a uniformly-distributed random variable in the range $[min, max]$. The integer is generated separately for each node in the command's scope. Can also be used inside a mass operation, in which case a different random integer is generated per iteration. (See Section 3.12.)

`rm`: Creates a new random integer uniformly distributed in the range $[\min, \max]$, shared among all Segment Managers in the command's scope. This is implemented via an algorithm that ensures all Segment Managers can verify that no party and no collusion of parties can control the result. Cannot be used inside a mass operation.

`rm`: This proper function initialises a distributed `rm` that is a uniformly-distributed random variable over all `rm` if `rm` is false, or just over the nonzero values if `rm` is true. The parameter `rm` is a Boolean. The value is generated separately for each node in the command's scope. Can also be used inside a mass operation, in which case a different random value is generated per iteration. This is a Priority 2 operation.

`rm`: Creates a new random `rm`, distributed as in `rm`, but shared among all Segment Managers in the command's scope. This is implemented via an algorithm that ensures all Segment Managers can verify that no party and no collusion of parties can control the result. Cannot be used inside a mass operation. This is a Priority 3 operation.

`rm`: Same as `rm`, but using a floating point variable. This is a proper function and can be used inside mass operations.

`rm`: Same as `rm`, but using a floating point variable.

`rm`: Accepts a dictionary, "`rm`", and an integer, `n`, such that `n` is no smaller than the number of keys in the dictionary. Creates a new dictionary with the same keys (and key type), but replacing the value associated with each key by a distinct random integer in the range `[0, n - 1]`. A set can also be used instead of a dictionary, with the result still being a dictionary. Can

also be invoked with a single integer, `n`, in which case the result is a list of the integers in `[0, n - 1]`, randomly permuted. The output is in all cases a distributed variable. Its value is independently distributed between nodes in the command's scope (even if the command is invoked with parameters that are shared variables).

The functions `Random` and `SharedRandom` presently work with the only integer type supported, and `RandomFloat` and `SharedRandomFloat` work with the only floating point type supported. In future we may have more than one integer type and more than one floating point type. In this case, the above functions should generate random numbers according to the type with the widest range. The FTIL programmer will then convert them to any desired type. FTIL's conversion rules are discussed in Section 3.11.9.

The random value generating functions are not privileged, and are distinct from the (also non-privileged) functions that exist for the generation of all types of cryptographic keys in the system. These latter take the form `rm`<*type*>`rm` and `rm`<*type*>`rm`, as well as *type*`rm`. In Section 3.5.1, we have already listed in this category explicitly the functions `rm`, `rm` and `rm`.

3.11.8 Shared construction

At the moment, FTIL only supports the following types of methods for the creation of shared variables, all of which have been previously mentioned.

1. Broadcasting a local variable,
2. Generation of a random shared variable,
3. Generation of a random shared key,
4. System constants, which are a form of predefined shared variables,
5. Pseudotypes like `module` and `method` that are inherently shared, and are created by specialised constructors, and
6. Generation of shared variables from a computation that only has shared inputs.

In future, we may want to also support the creation of shared variables by a privacy-preserving computation from distributed inputs. For example, the sum, logical AND and logical OR of a distributed variable across a set of nodes can in-principle be computed as a shared variable. However, such operations are not needed for the algorithms currently on the table, and we rate support of these a Priority 3 requirement.

3.11.9 Type conversions

We have already discussed the ability to convert between types in the context of constructing one type from another type. For example, a dictionary can be converted to and from a list of pairs.

In general, if `s` is a variable of type `S` and we wish to construct a variable `t` of type `T` this can be done by

```
t = T(s)
```

where the appropriate constructor for the type `T` is supported. Such construction is always treated as a proper function, so can be used conveniently for type conversion.

Table 3.1 lists every pair (S, T) of types that support this kind of conversion among FTIL's atomic, fixed-sized types. Excluded from the table are only types that do not support such conversions at all.

The general spirit of FTIL type-conversion rules was meant to

1. Be easy to implement,

From \ To	rm	rm	rm	rm	rm	rm	rm	rm
rm	•	✓		✓	✓	✓	✓	✓
rm		•						
rm	✓		•					
rm	✓			•				
rm	✓				•			
rm	✓					•		
rm	✓						•	✓
rm								•

Table 3.1: The allowed type conversions.

2. Generate code that is easy to read,
3. Will not open up privacy problems or create undue potential for bugs, and
4. Provide the full functionality desired by the FTIL programmer.

For the MVP version of FTIL, this general spirit has resulted in the following rules.

1. MVP FTIL will support essentially no implicit conversions. The only exceptions are (1) as documented in Section 3.12, when using the mass operation reduction syntax, if the RHS is Boolean, it is implicitly converted to a $\{0, 1\}$ `int` value, and (2) implicit type conversion between the types "string" and "shared string".
2. MVP FTIL supports explicit type conversions where the types in question support this, but differentiates between information-conserving type conversions and information-lossy type conversions. Examples of information loss can be a loss of range or a loss of precision. Information-lossy conversions throw an error if the original variable cannot be exactly represented in the new variable. In mass operations, only information-conserving type conversions are allowed, except in the context of explicitly stating the type of the mass operation's LHS. As usual, any error within a mass operation causes the entire operation to fail.

As of the MVP version, there are only two types of information-lossy conversions:

1. When converting an `Ed25519Int` to an `int`, a too-large input may cause the result to be out of range. This results in an error.
2. When converting an `int` to a `float`, a too-large input may cause the result to lose precision. This results in an error. However, the language provides the command "`nearest()`", which returns for any `int` input the closest `float` to it, and never throws an error.

To be clear, not every conversion from `Ed25519Int` to `int` and from `int` to `float` is considered lossy. Only when the input is such that its specific value cannot be properly represented in the output is the conversion considered lossy, not when its type in general allows for such values. As a result, type conversions between `Ed25519Int` and `int` and between `int` and `float` are, in principle, allowed in mass operations. Only if the inputs are inappropriate will the command fail due to information loss.

Note 3.11.4. There are no conversions in FTIL from `int` to `bool` and from `float` to `int`. To convert your `int`, `x`, to a `bool`, use "`x != 0`". To convert your `float`, `v`, to an `int`, use a command like "`floor(v)`", "`ceil(v)`" or "`round(v)`". It is allowed to convert a `bool` to an `int`, and this is an information-conserving conversion.

Future versions of FTIL may allow other types of conversions as well. For example, when using the "`[] =`" sub-assignment syntax or in similar contexts where the target type is unambiguous. But any such extension is currently out-of-scope for the MVP (and is Priority 3 in general).

The functions `ceil`, `floor`, `round` and `nearest` are all proper functions, and can be used inside mass operations.

Note 3.11.5. The list of conversion options documented in Table 3.1 presents the type `int` as the de facto *lingua franca* of FTIL: conversions between fixed-sized, atomic types typically needs to go through `int` as a mediator. This works because the list of atomic, fixed-sized types introduced in this document only includes very few types that cannot always be converted to an `int` without loss.

This, however, is only the case because ElGamal semi-homomorphic encryption is the only cryptographic algorithm fully detailed in this document. Once all required cryptographic primitives mentioned in Section 3.5.1 are implemented, there will be a need for the language to support, in addition to any direct conversions between types where appropriate, also use of the widest available integer (currently `Ed25519Int`) as a mediator. The intent here is to allow the FTIL programmer not just to convert freely between types without loss whenever the destination type is able to contain the information, but also to perform arithmetic manipulation on any stored data.

Such arithmetic manipulation is critical in implementing many privacy-preserving protocols.

3.11.10 Flavour conversions

If `x` is an elementary, shared variable, then "`distribute(x)`" returns a distributed variable of the same type and value. It is created in the scope of the command, which must be a subset of the scope of `x`.

To change a value from "distributed" to "shared", use `broadcast`, described in Section 3.11.6.

3.11.11 Scope conversions

Let us assume, for simplicity, that the identifier `y` is not defined. Under this assumption, if `x` is an elementary variable, the command

```
y = x
```

makes `y` into a new reference to `x`. The entity referred to by `y` therefore is now the same as that referred to by `x`. However, the command was not necessarily run in the same scope as the scope in which `x` is defined. It may be executed in a subset of this scope. In this case, while `y` and `x` still refer to the same entity within the scope of the command, The identifier `y` remains undefined elsewhere. This is an example of how a variable's scope can be narrowed in FTIL.

Such narrowing can be done with any elementary variable, whether distributed or shared.

Expanding a variable's definition requires inter-node communication, and requires the tools of Section 3.11.6.

Referential variables cannot change their scope in this way. If `x` had been a referential variable, the assignment could only have been done at the scope of `x`.

3.11.12 Conditional execution

Usually, when FTIL allows conditional execution, the determination of which nodes perform which commands takes place at the coordinator node.

This is done in choosing the scope of commands using the "`on()`" modifier, and also in the flow control operations `if/else`, `for` and `while`, which are are only allowed in scripts (See Section 3.13).

However, there is one way in which code can be executed conditionally in nodes, based on conditions evaluated by the nodes themselves. This is done through the ternary conditional operator.

The FTIL ternary conditional operator has the following syntax:

conditional `?` *type*`(`*iftrue* `:` *iffalse*`)`

The semantic interpretation is that if *conditional* is true, the result equals *type*`(`*iftrue*`)`, whereas if *conditional* is false, the result equals *type*`(`*iffalse*`)`.

In both cases, *type* is the return type of the result, and type conversion takes place if necessary.

The ternary conditional operator has the following characteristics:

1. It is not privileged.
2. It is a proper function (meaning its result can be used as input for other functions).
3. Its parameters (as in general functions) can be literal constants, variables, or themselves expressions that are proper functions.
4. The return type, *type*, must be an elementary type.
5. It is allowed as part of mass operations, if its return value is a fixed-sized type. (See Section 3.12.)
6. The value of its conditional at any node remains private, and is not shared with other nodes or with the coordinator.

As in all FTIL proper functions, the return value is a value, not a reference. In this case, this means that the value returned is a new (temporary) entity, rather than a reference to either *iftrue* or *iffalse*. Whether the conditional is true or false, usage of the ternary conditional operator always involves data copying.

Note 3.11.6. Evaluation of the ternary conditional operator, like all other Boolean arithmetic in FTIL, does not use any short-circuit Boolean evaluation, as this has the potential to leak information by non-data means.

In the ternary conditional operator and in Boolean binary operator computation, all operands will need to be evaluated, even though in the ternary operator always (and in binary operators sometimes) the evaluation of some parameters is superfluous for arriving at the result of the computation.

This means, in particular, that regardless of the value of the conditional, the expressions for *iftrue* and for *iffalse* must always be valid to evaluate, or this will result in an aborted execution.

3.11.13 Refreshing

Mathematically, to *refresh* an ElGamal ciphertext is to add to it a fresh (i.e., never before used) encryption of a zero. In the FinTracer algorithm, we often need to refresh millions of elements of value type ElGamal ciphertext.

The problem is that the naive way of implementing a refresh does not take advantage of the fact that summation of two ElGamal ciphertexts is roughly 100 times faster than encryption.

A better way is to prepare in advance a sufficient stockpile of encrypted zeroes, which can all be calculated offline, so that when a `Refresh` operation is performed, this merely sums a value from the existing stockpile.

To explain why `Refresh` merits special treatment, consider how one may try to use the zeroes of the stockpile in order to refresh a variable.

Readers familiar with the mass operations syntax introduced in Section 3.12 may be tempted to program a "refresh" command as follows, where "`stockpile`" is the list storing the stockpiled encrypted zeroes.

```
v[@] = u[@].Refresh(stockpile)  # This is an FTIL error.
```

This code is an error because each refresh operation changes the stockpile. The command is not a data-parallel operation and therefore not valid for use with the mass operation syntax.

Even had that not been a concern, consider that each refresh, for each ciphertext element, needs to verify the stockpile hasn't been depleted (or fail). We definitely don't want millions of such prerequisite checks. Rather, we would like an implementation that simply checks once, at the start, that enough zeroes exist in the stockpile for refreshing the entire target object. This can only be performed by refreshing an entire object as a single unit.

To support such refreshing, FTIL provides the built-in, in-place command

```
Refresh(u, stockpile)
```

that works, in the same syntax, whether `u` is an `ElGamalCipher` or an `array`, `list`, `memblock` or `dict` thereof, as well as the not in-place built-in command

```
Refresh(u, stockpile, v)
```

that places in `v` the refreshed version of `u`, and works on all the data types above, plus also `set`.

Refresh as a built-in command is only really effective for `sets` and `dicts`, where in-FTIL solutions will struggle to perform the operation truly in-place (in the case of `dicts`) and with no unnecessary overheads.

Performing the same for `lists` and `memblocks` (and `arrays`, if these are fully supported), is relatively straightforward. An example of an in-FTIL, in-place implementation for these data types is provided in Section 4.1.2.

One important question in implementing such a `Refresh()` is how integrated it needs to be with the rest of the FinTracer environment. We believe it should be fully integrated. Alternative solutions, taking different approaches, may do, for example, any of the following in refreshing ciphertexts:

- They may generate the encrypted zeroes by a process external to the FinTracer box.
- They may transmit the encrypted zeroes to the peer nodes by a process external to the FinTracer box, such as via dedicated harddisks, uploaded by an RE-side operator.
- They may not keep the stockpile of zeroes as an FTIL variable. It may be in a form entirely hidden from the FTIL programmer, and not be used as an extra input to the `Refresh()` command.

The problems with such non-integrated solutions are as follows.

1. When removing the ElGamal private key from the coordinator node, the FinTracer system can no longer verify that the key was not shared with any unauthorised party,
2. When uploading RE-side data into the system externally, the FinTracer system can no longer guarantee that these zeroes satisfy the cryptographic properties they should satisfy, and can therefore no longer guarantee the safe usage of the system, and
3. Technically, the FTIL programmer will need to specify which stockpile to use in refreshing, because algorithms may use ciphertexts calculated over multiple cryptographic keys, and the zeroes used must be ones that were generated with the correct key.

Whether the solution is in basic FTIL or using a special command, we believe that only an in-box solution allows complete preservation of the full "chain of evidence" of cryptographic guarantees, and should therefore be our implementation of choice.

3.11.14 Labelling

FTIL supports three functions for labelling:

- `label`,
- `confirm_label`, and
- `modifying`.

See Section 3.10 for details, including for details regarding how these functions interact with privileged execution.

3.11.15 Arithmetic and Boolean operations

Integers support the four arithmetic operations, as well as modulo-taking ("`%`") and the unary minus, all of which should also be available in mass operations. Also supported (but not relevant for mass operations) are the in-place operations "`+=`", "`-=`", "`*=`" and "`/=`". Overflows and underflows lead to errors, as does division by 0.

Integer division is handled the Python way: the result of division is always rounded towards negative infinity, and `a % b` is defined as $a - b\lfloor a/b \rfloor$.

Support of bitwise logical operations and bit-shifting is encouraged. (Priority 2.)

For `floats`, we require the same but only for the four arithmetic operations and unary minus. Modulo is Priority 2. Other numerical types (like `Ed25519Int`) should support the subset of these applicable to them. This goes also for homomorphic and semi-homomorphic cipher types.

For Booleans, we support the operations "`and`", "`or`" and "`not`" and their in-place counterparts "`&=`" and "`rm`". We also support the Boolean-generating comparison operators "`==`", "`!=`", "`>`", "`<`", "`>=`" and "`<=`", for applicable types.

Except for the in-place operations, all these should be available also in mass operations.

FTIL does not use short-circuit Boolean evaluation. All parts of an expression are always evaluated, including in the ternary operator.

For `floats`, power taking ("`**`"), conversions to `int` through `ceil`, `floor` and `round`, and the functions `max`, `min`, `exp`, `log`, `sin`, `cos` and the sign function ("`sgn`") should all also be supported. For `ints`, of these only `max` and `min` are required (but see Section 3.11.9 for "`nearest`").

For a fully-operational system, the above functions should all be available in mass operations, too. However, presently none of our algorithms utilise them in a mass-operation context, for which reason we view it as acceptable to consider the mass-operation support for these as Priority 2 for the Purgles Platform's MVP. Supporting them in other contexts is still Priority 1, however.

3.11.16 String operations

FTIL is extremely sparse in its support for string operations, because strings are only meant for interactions with human operators or to convey SQL queries to databases. If pretty-printing is necessary, it should be handled either before or after the FTIL layer. FTIL's handling of string operations is different to other types. See Section 3.7.3 for details. In particular, string commands may be handled by the coordinator node instead of by the nodes in scope, may trigger implicit conversions between strings and shared strings (in either direction), and may require privilege at the coordinator node.

The following are FTIL's string operations.

String concatenation: Strings can be concatenated using the "`+`" operator.

`rm`: Where `v` is either a fixed-sized entity or an element within an elementary entity, "`str(v)`" returns a human-readable string representation of `v`. The value of `v` is evaluated solely at the coordinator node, regardless of the scope of the rest of the command.

These are proper functions.

Additionally, FTIL can perform some date arithmetic, using date strings recognisable by SQL. (Indeed, these date operations may optionally be implemented by offloading the calculation to the underlying Auxiliary DB on the coordinator node.) An example input would be a string in "`YYYY-MM-DD`" format.

The FTIL date functions are:

`rm`: Where both `start` and `finish` are strings containing legal dates, this function returns an `int` containing the number of days separating `start` from `finish`, where "`datediff("1970-01-01", "1970-01-01")`" equals zero. This is a proper function that conceptually runs on the nodes in the scope of the command, as per the standard execution pathway, but for technical reasons may need to be offloaded for execution on the coordinator node. There are no privacy implications for this offloading. See Section 3.7.3 for details. Because the function runs on the nodes

(at least conceptually), any input that is a non-shared string will need to be converted to a shared string, which requires the coordinator node to be executing in privileged mode. The output `int` is a shared integer.

`rm`: Where `start` is a string containing a legal date and `offset` is an `int`, the function returns a legal date-formatted string that represents a date that is `offset` days from `start`. The functions `datediff` and `dateoffset` are complementary in that "`datediff("1970-01-01", dateoffset("1970-01-`always returns the integer `offset`. Like "`str()`", this is a proper function that is evaluated solely at the coordinator node, regardless of the scope of the rest of the command. The value of `offset` used is therefore solely its value at the coordinator node. If `start` is a shared string (such as if it is a literal constant) it must first be converted to a string in order to be evaluated at the coordinator node, but such an action is not privileged. (If the resulting string requires broadcasting to nodes other than the coordinator, that may, however, be privileged at the coordinator, as per the standard rules explained in Section 3.7.3.)

These functions fail if their string inputs are not legal dates, or otherwise not valid dates for the operation.

The idea of these date commands is that one should be able to convert dates to integers, perform any necessary arithmetic on the integers, and then convert back to date format in order to plug the resulting dates into SQL commands.

3.11.17 Membership

The construct "`x in y`" can be used in FTIL to determine whether `x` is a member of a "set" (in the general meaning of the term) represented by `y`.[27] The result type is `bool`. This is a proper function. It can be used both inside mass operations and not. However, its use in mass operations is restricted to situations where `y` is of an elementary type.

Its negation, usable in exactly the same contexts, is "`x not in y`".

Figure 3.6 lists the membership queries that can be resolved using Variable Registry data. Where the result flavour is listed as "shared", it is considered to be shared within the scope of `y`.

Figure 3.6: Cases of "x in y" resolvable using Variable Registry data

Type of `rm`	Type of `rm`	Result flavour	Comments
shared string	`rm`	local	Is `rm` the name of an attribute of `rm`?
shared string	metadata	shared	Is `rm` the name of an attribute of `rm`?
shared string	`rm`	shared	Is `rm` the name of an attribute of `rm`?
`rm`	`rm`	shared	Is `rm` a key of `rm`?

Figure 3.7 lists the membership queries that require Variable Store data. In this case, all results are distributed.

Figure 3.7: Cases of "x in y" requiring Variable Store data

Type of `rm`	Type of `rm`	Comments
`rm`	`rm`	Is `rm` an initialised key of `rm`?
`rm`	`rm`	Is `rm` a key of `rm`?
`rm`	`rm`	Is `rm` a key of `rm`?
`rm`	`rm`	Is `rm` a member of `rm`?

[27] Despite the visual similarity, this is different to the construct "`rm`", regarding which see Section 3.13.4.

Both types of information are accessible both to Segment Managers (i.e., as part of standard execution) and to the Compute Manager (i.e., in contexts such as scoping and flow control). The general rule is that when Segment Managers evaluate the conditions, they evaluate them based on data that exists in their local Variable Stores, whereas when the Compute Manager evaluates the conditions it does so based on its own Variable Registry where it can, or else based on data that exists in the coordinator node Variable store.

Where the result is based on data that only exists in Variable Stores, if computation is in a Segment Manager context, the individual Segment Managers on the nodes evaluate the condition individually, and the result is a Boolean distributed among them, whereas if computation is in a Compute Manager context, the result is evaluated based on coordinator node Variable Store data alone, and is a temporary, local Boolean, only usable by the Compute Manager (barring implicit broadcasts).

This is, so far, the standard process for evaluating a command, as detailed in Section 3.7.

The only unusual complication regarding membership queries is when `x` is a shared string and `y` is an `object`. Here, membership can be evaluated both using Variable Registry data and using Variable Store data, but each will return a different value. Variable Registry data is used in Compute Manager contexts. It returns whether `x` is an attribute of `y` anywhere. The return value is in this case a temporary Boolean value, local to the coordinator node, that can only be used by the Compute Manager (barring implicit broadcasts).

On the other hand, Variable Store data is used in contexts handled by the Segment Managers. In this case, the return value indicates whether `x` is an attribute of `y` on the node that performed the querying. The result is a Boolean value distributed among all nodes in the command's scope.

The rest of the process is the standard command evaluation process, as described in Section 3.7. A full end-to-end example was given already in the discussion of strings in Section 3.5.4.

3.11.18 Deleting entities, variables and identifiers

The command "`rm`", for a variable `x`, when used at the command line, is used to delete the variable, removing its reference to the underlying entity. If this is the last reference to said entity, the entity is deallocated. If "`x`" is an identifier, the identifier is also deleted.

For security reasons, entities whose last reference is deleted, prior to freeing their own memory, should preferably overwrite their memory by random bits. (This overwrite is Priority 2.)

When used inside scripts, the `del` command needs to handle some complications to do with the fact that the same variable may have many names: it may be, simultaneously, an attribute, a script formal parameter, an identifier of a calling script, or any combination thereof. In some situations, we don't want to delete the variable entirely, but rather to only deallocate the link to the underlying entity, leaving the variable as an uninitialised variable. The reason for this is to allow the same variable to be redirected to another entity later on, still keeping all of its original names (and particularly the connection between them).

For this reason, when used inside scripts, `del x`, on an identifier `x`, does not always delete the identifier.

The full rule-set governing `del x` on a variable `x` inside scripts is as follows.

If the variable `x` is an attribute and has no identifiers, `del` deletes the variable. If the variable `x` is not an attribute and has exactly one identifier (It cannot have zero), then `x` is that identifier; `del` deletes both the identifier and the variable. Otherwise, the command unlinks the variable from its underlying entity, making it uninitialised.

In this latter case, if `x` is an identifier, it is necessarily a script formal parameter, so does not get removed.

See Section 3.13.3 for a full description of how parameter passing in scripts works, including in the context of uninitialised variables.

In command-line use, `del` can be used in several ways that are not permitted inside scripts.

First, `del x` can be used on a non-module script. As usual, in this case the name of the script must be preceded by a "`::`".

Second, when at the command line, "`del`" may also accept a wildcard-expression as a parameter. This allows the user, for example, to delete all non-constant variables using "`del *`". This is useful if, say, a script clutters the name-space by an inordinate amount of junk variables. Deleting using a wild-card does not impact scripts. Attempting to delete a constant variable has no effect.

If a "`del`" command does not delete any variable (e.g., the only variables matching the parameter are constant), the command fails.

Attempts to delete variables by a command whose scope does not include every node the variable is defined in cause the `del` command to fail. Attempts to delete variables by a command whose scope extends beyond the scope of the variable are only allowed when deleting via a wildcard-expression.

Outside of deleting via a wildcard-expression, variables can only be deallocated at the exact scope in which they are defined. Variables can also only be re-referenced to a new entity at the exact scope in which they are initially defined.

To remove all members from a `catalogue`, `c`, use "`c.empty()`" instead of any `del` syntax.

As a Priority 2 feature, it is also allowed, both from the command line and from scripts, to run `del` on a method of an unlabelled—or a label-suspended—`object`. There is no use of "`::`" in this case.

3.11.19 Deleting elements

The syntax "`rm`", when `x` is a `dict`, deletes the key `key` from the dictionary, along with its associated value. If the dictionary does not have this key, the command fails.

If `key` is an iterable of the dictionary's key type, all keys in `key` are deleted. When deleting through an iterable, any value in `key` that is not in the domain of `x` is ignored, without this causing the command to fail.

Remark for the advanced reader 3.11.1. Eagle-eyed readers will note that if `x` is a `dict<X,Y>` and `keys` is a `set<X>`, the final result of "`del x[keys]`" is the same as that of the mass operation "`x.filter[@ not in keys]`".

While this is true, the two operations are not equivalent. They have different complexities: the former iterates on the variable `keys`, so presumably works at a complexity that is proportional to the size of the `keys` set, whereas the latter iterates on the keys of `x`, so presumably takes time proportional to the original size of the `dict`.

The syntax "`rm`" is also used to remove position `i` from a list `x`.

For lists, FTIL also supports removing entire slices via Python-like semantics, using "`del x[a : b]`", "`del x[a :]`", "`del x[: b]`" and "`del x[:]`", where the last deletes every element of the list.

To remove all elements from a `set` or a `dict`, use "`x.empty()`", instead of any `del` syntax.

To remove some subset of elements, `y`, from a set `x`, use "`x.setminus(y)`". The "`setminus`" method works in place. There is also a function by the same name that does not work in place, and, instead, returns the set difference.

3.11.20 Miscellaneous commands

The following are a few additional FTIL commands, used in general housekeeping.

`rm`: Returns the names of the top-level variables (i.e., the identifiers) in the Variable Registry to the AUSTRAC-side user. This is a locally-run command, and can only be executed from the command line. It does not accept a scoping directive. The result is output to the user but not stored in any variable. The function may also receive a parameter. This should be a variable of type `object` or of one of the pseudotypes `metadata` or `type`. If a parameter is given, the result returned relates not to the top-level variables but rather to the attributes (both methods and members) of the parameter. As a Priority 2 feature, the returned output from "`dir()`" (with or without a parameter) should include also the variable types and,

for variables of a collection type, also their labels if such exist. The "`dir()`" command is conceptually run by the Compute Manager, and uses Variable Registry data, meaning that it reports even on variables whose scope does not include the coordinator node. However, it does not report on non-module scripts that do not belong to the current user.

`rm:` Scripts (both module scripts and non-module scripts) are allowed to start with an explicit literal string If it exists, this is the script's help string. If it does not exist, the help string is taken to be the string describing the script's parameter signature, e.g. "`f(a, b = 7)`". The command `help(script)`, available only on the command line, outputs to the FTIL user the help string of the script. Note that the script name must be preceded by a "`::`". The command can also be used on methods (in which case, no "`::`" is used). When applied on a method, the command outputs the help string of the method's underlying script. This command is a locally-run command and accepts no scoping directive. It is executed using Variable Registry data, by the Compute Manager. The command only returns information on non-module scripts if the scripts belong to the current user.

`rm:` The command reads the `URI` target as text input and executes it as though these were commands typed at the FTIL command-line, but with command echo. The `URI` parameter must be an explicit string literal. This command can only be invoked at the command line and does not accept a scoping directive. If any of the commands in `URI` fail, the `exec` fails and is aborted.

`rm:` Takes an integer, `n`, and returns a list with the values 0 through $n-1$. This is a proper function.

`rm:` This is a proper function, returning the current execution scope. As one would expect from a function that takes no arguments (so, in particular, all of its arguments are shared), the returned value is a shared value. It can also be computed by the Compute Manager.

`rm:` A proper function, not eligible for mass operations, returning a `bool`, that determines whether a variable `x` is initialised (is associated with an entity). This is resolved either by Variable Registry data or by Variable Store data, depending on whether it is executed in a Compute Manager or in a Segment Manager context. (See Section 3.7.) As per the standard of Section 3.13.4, use of this function in flow control decisions can make flow control privileged if `x` is a variable visible to the Compute Manager at the coordinator node but not to Segment Managers at peer nodes that are in scope, but otherwise the function can be used without special privileges. If executed on a `transmitter` attribute in a Compute Manager context, the function returns `True` always. (This is relevant when querying, for example, `t[n]`, where `t` is a `transmitter` and `n` is a `nodeid` within the scope of the `transmitter`. If `n` is not in the scope of the `transmitter`, the command will not return `True` or `False` but will, rather, fail, because in that case `t[n]` does not exist at all.)

`rm:` Anything from a hash character (outside a string literal) to the end of the line is a comment. (We may also want to support multi-line comments, but that's Priority 2 at best.)

Additionally, the standard library includes the proper function `BranchToBank`. This accepts a `BSBNum` as input and returns the `nodeid` corresponding to the peer node of the RE managing the account. This function can be used in mass operations. It can fail if used on an input that is not a real BSB value.

Note 3.11.7. previous versions of this document also supported a "`scope(v)`" function to determine the scope of a variable `v`. This is no longer supported. Use "`v.metadata.scope`", instead.

Note also that FTIL no longer supports the system constant `AllNodes`. Use, instead, the result of "`scope()`" when run on the command line. FTIL scripts have no visibility into what nodes may exist beyond their own execution scope and the scope of the variables sent to them. If a script really needs to discover all nodes in the system, it can do so by checking the scope of a system constant, e.g. "`CoordinatorID.metadata.scope`".

3.12 Mass operations

FTIL was designed to be quite flexible. However, at its core it must perform some heavy lifting—massive amounts of data need to be processed, and this processing must be performed as efficiently as possible. As a result, we do not rely on FTIL to perform any long loops over billions of entries. (In fact, outside scripts, FTIL has no loops, nor any other means of flow control.)

To accommodate for this, FTIL supports some mass operations—single operations that manipulate mass amounts of data. A good choice of an underlying implementation paradigm for FTIL is one that allows these mass operations (whether in general, or at least in their commonly-used forms) to be implemented efficiently. Advocating for any particular choice in this is beyond the scope of this document.

We do note, however, that in general these mass operations take the form of data-parallel actions (presented as list comprehensions) that are performed independently (or, at least, essentially-independently) over all elements in an entire *iterable* variable. Iterable variables (or *iterables*) in FTIL are lists, memblocks, dictionaries, sets and arrays. These are all templatised variables over a fixed-sized type. In other words, when iterating over them we iterate over a homogeneous population of fixed-sized elements. Note, in particular, that any validity checks that need to be performed can be performed only once, at the container level. We believe this structure makes it as easy as possible to run such code efficiently in practice.

The FTIL list comprehension syntax relies on *iterators* that can either be named or unnamed. We begin by describing examples with unnamed iterators. These are presented in the code as "at" symbols, "`@`", and are referred to as *wildcards*. In general, mass operations can be worded in a very succinct syntax, or detail more elaborately all elements. We first describe the most succinct syntax, where many elements are inferred, and describe how they are to be inferred, and then go back to describe ways to make such details explicit, in case one wishes to override an inference.

Note 3.12.1. We may decide to add a few more commands for facilitating very specific mass operations, with the intent of being able to implement them in a way that minimises data movement and duplication. The examples implemented so far do not require those, though, so no such extra commands were included in this document.

We remind the reader once again that any specific syntax described in this document is conceptual only. The material points are the capabilities, not the syntax.

Note 3.12.2. In all examples below, mass operations are demonstrated that involve operators working on fixed-sized inputs and returning fixed-sized outputs. Mass operations can also be used with simple functions, i.e. functions whose inputs are all fixed-sized types and whose output is a fixed-sized type. (See Section 3.11.) Commands that are not simple functions cannot be used inside mass operations.

Some operations have the potential to throw an error. For example, division may throw an error if the divisor is zero. Any such error, occurring in any part of a mass operation, causes the entire operation to fail.

For reasons of computational efficiency, programmers should probably avoid using such functions inside mass operations whenever possible, but it is not illegal to do so.

Note 3.12.3. Results of mass operations are always distributed, never shared.

3.12.1 Implicit mass operations

FTIL supports the following types of mass operations:

RHS-only wildcard operations: These are operations with a syntax such as

"`rm`".

They can be recognised by the fact that the wildcard operator "`@`" appears in them, but only on the right hand side (RHS). The wildcard may appear any number of times. The

semantic interpretation of this is the following. First, we seek the right-most wildcard. In this case, it is used as an index for "`w`". The full RHS computation, "`v[@].first + w[@][4]`", is now taken to yield an entity of the same basic type (`array`, `memblock`, `list`, `set` or `dict`) as `w` and with the same set of keys. The value associated with each key, `k`, is the value of "`v[k].first + w[k][4]`", and the value type of the result is set accordingly. This result is computed, and then assigned to the left-hand-side (LHS) variable, in this case "`u.xyz`", as usual. If for any `k` it is not possible to evaluate the RHS, e.g., in the present example, if for some `k`, `v[k]` is not a key of `v`, the operation fails. As demonstrated in the example, the wildcard may not be the only index being used. All of the following are legal: "`w[@][4]`", "`w[4][@]`", "`w[@].first`", "`Encrypt(w[@], key)`". The type of the results can be overwritten by surrounding the RHS by brackets preceded by the target type, as in "`u.xyz = list{v[@].first + w[@][4]}`". See Section 3.12.2 for the exact interpretation of the target type syntax.

LHS wildcard operations: These are operations with a syntax such as

"`rm`".

They are distinct from the RHS-only wildcard operations in that the wildcard appears in the LHS. It may or may not appear in the RHS. The semantic interpretation of this command is as follows. First, we seek the right-most wildcard and find which variable it indexes. In this case, it is an index of `w`. (In other circumstances, it may be an index into the LHS variable.) The line is now interpreted as a "`for`" loop over all keys of said variable. For each such key, `k`, we perform, in this case, "`u[k].xyz = v[k].first + w[k][4]`". Again, as in the example, all manner of sub-indexing is allowed. This includes even sub-indexing of the compositional form "`w[t[@]]`". However, this last is only allowed on the RHS. On the LHS it has a different semantic meaning. (See below.) Also, as before, if for any `k` the expression is invalid, the operation fails. Note that in this case the command has to be a "`=`" assignment (although what is being assigned here are the elements and not the variable).

LHS wildcard modifiers: These are operations with a syntax such as

"`rm`".

They differ from the previous case only in that instead of direct assignment to the LHS, they modify it. This requires the underlying element type to support the modification operator used. Modification operators supported in the mass operation syntax are "`+=`", "`-=`", "`*=`" and "`/=`". Clearly, this means that numerical types are supported, but, for example, the "`+=`" syntax is also valid for adding together two `ElGamalCipher`s, whereas "`*=`" works for multiplying an `ElGamalCipher` by an `int` or an `Ed25519Int`. If the LHS is a `list`, `array` or `memblock`, the operation has the same basic semantic interpretation as in the previous case, i.e. as a loop over keys `k` of

"`rm`".

The range of `k` must be a legal range for the keys of the LHS. The operation cannot extend a `list` (and, of course, `array`s and `memblock`s can never be extended). If the LHS is a dictionary type, the interpretation is different, because here it may be the case that `k` is not an existing key of the LHS. For dictionaries, where a key, `k`, is encountered that does not exist in the LHS dictionary, the operation is treated as though the value was initially a zero. In this case, the result of "`u[k] += a`" becomes identical to "`u[k] = a`", "`u[k] -= a`" becomes "`u[k] = -a`", and "`*=`" and "`/=`" both leave `u` without the key `k`. This logic is used also for non-numerical types, as long as they support the relevant modification operator. This syntax does not support LHS values that are of type `set`.

Filtering: The syntax "`v[t[@] == 7]`" can be used as well. The wildcard is used here as part of an expression that evaluates as a Boolean, and this Boolean is then used as an index.

This can occur either in the LHS or in the RHS, but can only occur once in an expression. Elsewhere, the wildcard can be used in the expression as usual, and even within the Boolean expression, the wildcard can be used more than once. Semantically, this operator changes the range over which the wildcard ranges. The range becomes all the keys `k` of `v` for which the Boolean expression (in this case "`t[k] == 7`") holds. It overrides the usual interpretation for the range of the wildcard. Otherwise, the semantic interpretation is exactly as in the previous cases, except in the RHS-only case if the expected result type is `list` or `memblock`. In this case, the expression evaluates as a `list` (or `memblock`) with only the values in the places where the conditional holds. The rest of the items are removed, and the remaining items are shifted so as to keep the `list`/`memblock` contiguous. It is not allowed to perform filtering when the target type is `array`.

Some examples:

- The expression "`u = v[t[@] == 7]`", creates a new variable, `u`, of the same type as `v`. If `v` is a dictionary, it is a copy of `v` that filters out any key `k` of `v` for which "`t[k] == 7`" does not hold. If `v` is a list, the new list is a filtered down version of `v`, keeping only those values at indices `i` for which "`t[i] == 7`" holds. If some values do not satisfy this criterion, the resulting list will be shorter than the original list. The variable order in the resulting list is implementation dependent, but must be deterministically replicable if same-valued inputs are filtered in the same way, including on other nodes.
- The expression "`u[@] = v[t[@] == 7]`" takes an existing variable `u` and for any `k` that is a key of `v` for which "`t[k] == 7`" is satisfied, assigns `v[k]` into `u[k]`. If `u` and `v` are lists, the highest `k` value for which "`t[k] == 7`" is satisfied within the range of the length of `v` must not be larger than the maximum index into `u`, or the command will fail.
- The expression "`u[@] += v[t[@] == 7]`" takes an existing variable `u` and for any `k` that is a key of `v` for which "`t[k] == 7`" is satisfied, increases `u[k]` by `v[k]`. If `u` and `v` are lists, the highest `k` value for which "`t[k] == 7`" is satisfied within the range of the length of `v` must not be larger than the maximum index into `u`, or the command will fail. If `u` is a dictionary and any such `k` is not originally a key of `u`, this key will be added as though the original value was zero.
- The expression "`u[t[@] == 7] = v[@]`" has a similar meaning to the meaning of "`u[@] = v[t[@] == 7]`", but not identical. Here, instead of `k` ranging over the range of `v` and filtering that range by the Boolean expression, `k` ranges over the range of `u` and the filtering is applied over that range. If `u` and `v` are both lists, here the highest `k` value for which "`t[k] == 7`" is satisfied within the range of the length of `u` must not be larger than the maximum index into `v`, or the command will fail.

In-place filtering: Consider a dictionary or a set from which we want to remove some keys based on a conditional. The filtering syntax introduced above can be used here, but the result will be an entirely new entity, with potentially massive amounts of data duplication as part of the process. To avoid this, we support a syntax for in-place filtering. The command "`d.filter[@ < 1000]`" retains in the variable `d` only those keys, `k`, satisfying the conditional `k < 1000`. The variable `d` can be of any iterable type except `array`, but the syntax is only really useful where such operations can be performed efficiently in-place, such as when filtering `dict`s or `set`s.

Reduction: If the expression on the RHS is surrounded by braces, but these braces are not preceded by a type definition, this is interpreted as a reduction syntax. In the implicit syntax a reduction must be an expression of one of the following types, both types involving the wildcard being used as part of a nested look-up on the LHS: either "`u[t[@]] = {`*expr*`}`" or "`u[t[@]] += {`*expr*`}`". (The nested look-up itself might be more complicated. For example, it may be "`u[t[@].first]`", but the only two operations supported are "`=`" and "`+=`".) In both cases, the expression in the RHS may or may not also use wildcards. Semantically, in

both cases the command is evaluated as a `for` loop that ranges over all `k'` in the domain of `t`, partitioned by the value of `t[k']`. Within each part in this partition, the values of *expr* are summed together, i.e. we compute the sum of *expr* over all `k'` for which `t[k'] == k`, for each such `k`, and the final sum is then assigned (in the first variant) or added (in the second variant) to `u[k]`. This syntax assumes that the underlying variable types support the "`+=`" operator. They may be numerical or `ElGamalCipher`. If the values are Boolean, they are converted to integers in $\{0, 1\}$ and summed as such.[28] Keys of `u` for which no such `k'` exists remain unchanged. The LHS must be a pre-existing variable for which such element assignments/modifications are individually legal. If the "`+=`" operator can be executed in-place, the whole expression is executed in-place (up to copy-on-writes). Note that the wildcard range is determined here by its usage in the LHS, not (as in standard wildcard evaluation) based on its right-most usage. Reduction should also support "`u[t[@]] = max{`*expr*`}`" and "`u[t[@]] = min{`*expr*`}`", to compute the maximum and the minimum, respectively, of the underlying values, for `int` and `float` types.

Note 3.12.4. Above, we discuss iterables like `lists` and `dicts`. The `transmitter` type is not iterable. If `t` is of type `transmitter<list<int>>`, it would not be valid to use "`t[@]`". However, it would still be valid to write "`t[n][@]`", because "`t[n]`" is iterable.

Also, note that `sets` are iterable, so if `s` is a set then "`s[@]`" is a valid syntax. Moreover, in a line such as "`u[s[@]] += {v[s[@]]}`", the interpretation is, as usual, that for every element `k` in the set `s`, we perform "`u[k] += v[k]`". We single out sets in particular, because in sets it is less clear what the iterator "`@`" means as a standalone. Sets have no "keys" as such. For this reason, when iterating over sets every usage of "`@`" must be inside a reference to the same set.

3.12.2 Explicit mass operations

The implicit mass operations described above are very convenient to use in writing, because of the highly compressed code that they create, but ultimately they are all syntactic sugaring. They do not provide any more functionality than what FTIL affords in explicit mass operations. Explicit mass operations also have the advantage that they allow the user to determine explicitly all aspects of the operation. In implicit operations these are inferred, and sometimes one needs to override the "default" behaviour. Below are the same examples used above for implicit operations, translated to the explicit syntax.

RHS-only operations: "`u.xyz = list{v[i].first + w[i][4] over i = w.iter}`".

LHS operations: "`u[i].xyz = v[i].first + w[i][4] over i = w.iter`".

LHS modifiers: "`u[i].xyz += v[i].first + w[i][4] over i = w.iter`".

Filtering:

- "`u = list{v[i] over i = v.iter with t[i] == 7}`",
- "`u[i] = v[i] over i = v.iter with t[i] == 7`".

Reduction:

- "`u[t[i]] = {`*expr*`} over i = t.iter`",
- "`u[t[i]] += {`*expr*`} over i = t.iter`",
- "`u[t[i]] = max{`*expr*`} over i = t.iter`",
- "`u[t[i]] = min{`*expr*`} over i = t.iter`".

Note 3.12.5. An exception to the idea that implicit mass operations are merely syntactic sugaring is in-place filtering. There is no separate explicit format for in-place filtering.

[28] Which is convenient for implementing a "`rm`".

The explicit syntax also provides an additional way to specify iterators.

FTIL allows the syntax "`arrayiter(k)`", to iterate over the range 0 to $k-1$ for any k.

If `c` is of type `array<array<T, size2>, size1>`, the keys of `c` range from 0 to `size1`-1 and the keys of `c[x]`, for any `x` in this range, range from 0 to `size2`-1. To iterate over the former, one can use "`arrayiter(c.metadata.type.size)`". To iterate over the latter, one can use "`arrayiter(c.metadata.type.element.size)`".

One can also still use "`c.iter`", as usual, which in the case that `c` is an `array` is equivalent to "`arrayiter(c.metadata.type.size)`".

We refer to all of the above as *array iterators*.

Array iterators can be used to iterate over any variable that has the relevant integer keys, and are not restricted to only work on `array`s or to only work on `array`s of the correct size. One can also iterate on an `array` with a non-array iterator.

Note 3.12.6. Even though we have demoted the support for `array`s to Priority 2, support of `arrayiter` is still Priority 1.

The explicit mass operations syntax may be used with any number of normal iterators and any number of array iterators, where all iterators are comma separated. However, with the exception of the reduction syntax, on the LHS it should be the case that either no iterator is used (for mass operations that create a new entity) or all iterators are used (for operations that manipulate an existing entity). In the latter case, all iterators should be used directly as indices into the target variable. For these operations, it is necessarily the case that at most one non-array iterator can be used.

In RHS-only operations, the dimensions of the target variable are determined by the iterators used. Iterators should be defined in the same order that they are to be indexed in the output. For example, the single non-array iterator, if it exists, should be defined first. Elsewhere, there is no strict restriction regarding the order in which the iterators are defined other than that we recommend non-array iterators should come before array iterators.

Here is an example of an RHS-only operation with all bells and whistles together:

```
u = set<array<int, 7>>{v[i][j] * v[i][j] over i = v.iter,
                                              j = arrayiter(7) with w[i] > 3}
```

Note that in the explicit syntax we do not differentiate between an RHS-only wildcard operation and a filtering operation. Both can be done together, and the difference is merely in the appearance of the `with` clause.

The "`with`" clause, if it exists, should appear only at the end.

In operations that generate new entities, the "`with` clause, if it exists, must not refer to any array iterators. If the "`with`" clause in the example had been "`w[j] > 3`", this would have been an error.

Note that explicit mass operations that create a new entity require the programmer to explicitly list the basic type (`array`, `memblock`, `list`, `set` or `dict`) of the output type. In the implicit syntax, this was optional. Other than this declaration of the basic type, the details of the exact type are worked out automatically: the element type of containers is worked out by the type returned in the RHS, while the size of `array`s is determined by the respective array iterators. It is possible for a user to choose a different basic type to the default one, but if the default one is not `array` (i.e., if the first iterator is not an array iterator) one cannot override the default in order to make the output an `array`, if the default is `set` one cannot override it at all, and if the default is `dict` one can only set it to either `dict` or `set`.

In operations other than RHS-only ones, the order in which iterators are defined does not change the command's semantics, and if the command is such that the order does make a difference, results are undefined. (The format is only meant for data-parallel operations. This is with the exception of the many-to-one write relationship of the reduction operations, where we assume that addition is commutative and associative.)

Reduction in the explicit syntax is far more powerful than in the implicit syntax. It follows the syntax exemplified here:

LHS = {*expr*} over i = a.iter, j = arrayiter(*size*)

or

LHS += {*expr*} over i = a.iter, j = arrayiter(*size*)

and can also accept a "with" clause, with the same restrictions as above.

The reduction syntax supports any number of normal iterators and any number of array iterators, comma separated, followed by an optional "with" clause. The LHS may be any pre-existing variable of an appropriate container or array type, indexed in any way that would make the individual assignments/modifications of the mass operation be legal, had they been applied one at a time.

If the type of the variable assigned to is a set of arrays or a dict whose value type is an array type, the reduction syntax also requires a default value, like so:

LHS = {*expr*}(*default*) over i = a.iter, j = arrayiter(*size*)

This default value is not allowed if the result type is not a set of arrays or a map whose value type is an array.

The default value is optional if the underlying scalar type (possibly under several dimensions of arrays) is one that has a neutral element in addition. Where such an element exists (e.g., zero for ints and floats), it can be used as a default "default value" (and this is interpreted as initialising all cells in the potentially-multidimensional arrays that the operation needs to initialise).

To understand how the default value is used, consider the following example:

```
# Set 'a' to be a map from integers to 20-by-10 integer matrices.
a = map<int, array<array<int, 10>, 20>>()

# Set 'b' to be a 10-by-30 matrix, initialised as all zeroes.
b = array<array<int, 30>, 10>(0)

# Set 'c' to be a map from integers to 20-by-30 integer matrices.
c = map<int, array<array<int, 30>, 20>>()

... # Insert here some code that populates 'a' and 'b' with actual data.

# And now we want to set each c[i] value to c[i] = a[i] * b,
# where '*' signifies matrix multiplication.

# We do this using the following mass operation syntax.
# (Note that no default value is explicitly given. Because the underlying
#  scalar type of the result is 'int', the default value is taken to be 0.)
c[i][j][k] = {a[i][j][m] * b[m][k]} over i = a.iter,
                j = arrayiter(20), k = arrayiter(30), m = arrayiter(10)
```

The way this mass operation command is interpreted is as though the following loop was executed:

```
# IMPORTANT: This "effective implementation" is not quite FTIL.
#            See note below for further details.

for i in a.keys():
  c[i] = array<array<int, 30>, 20>>(0)

for i in a.keys():
  for j in range(20):
    for k in range(30):
```

```
    for m in range(10):
      c[i][j][k] += a[i][j][m] * b[m][k]
```

The default value is used to initialise new indices of `c` in the line:

```
c[i] = array<array<int, 30>, 20>>(0)
```

In general, reduction loops over the iterators as usual, but must first loop only over those iterators that are used as indices into the top-level `set` or `dict` (i.e., ignoring indexing into the underlying `array`s). This initial loop is used to initialise any `array`s that need to be created, using the default value. The reason we must do this in a separate loop is that, in the general case, indexing in a reduction syntax may involve repeating values. For example, instead of assigning to `c[i][j][k]`, we could have chosen to assign to `c[sgn(i)][j][k]`.

A similar situation occurs also in the reduction syntax with "+=". If the mass operation had been

```
c[i][j][k] += {a[i][j][m] * b[m][k]} over i = a.iter,
                j = arrayiter(20), k = arrayiter(30), m = arrayiter(10)
```

the only difference in the "effective implementation" would have been that the initial loop

```
for i in a.keys():
  c[i] = array<array<int, 30>, 20>>(0)
```

would have been replaced by

```
for i in a.keys():
  if i not in c:
    c[i] = array<array<int, 30>, 20>>(0)
```

i.e., only those `array`s needed that are not already initialised in `c` get initialised. The rest start from their old values.

Note 3.12.7. The "effective implementation" lines presented here are, as mentioned, not really legal FTIL commands. This is unless the scope of running this command is exactly the coordinator node. In all other cases, FTIL would have resolved the flow control instructions based on coordinator node information, and would have then performed the assignments on the in-scope nodes using the nodes' individual Variable Store data. This disparity was not the intended interpretation of the code in these "effective implementation" code snippets.

Another example of how array iterators can be used in the reduction syntax is to perform `array` reshaping, as in the following example:

```
u = array<array<int, 20>, 300>(0)
u[i*30+k][j] = {v[i][j][k]} over i = arrayiter(10), j = arrayiter(20),
                                  k = arrayiter(30)
```

Some places where this syntax affords more power than the implicit one:

1. It is explicitly determined which container the iterator iterates over, possibly overriding the default rules.

2. The syntax supports iterating over multiple indices, including reductions involving multiple non-array indices.

3. The "`arrayiter`" format allows one to iterate over a chosen integer range, even when no variable with the range of keys is currently defined.

3.12.3 Memory layout

Low-level languages, such as C, typically stipulate the memory layout of their objects. High-level languages typically do not. This puts FTIL in a tight spot. On its front end, FTIL aims to provide a high-level interface for its users. On the back end, that interface needs to power some high-powered computing whose efficiency will greatly depend on a conducive memory layout of the objects it manipulates.

To get out of this dilemma, FTIL recommends the memory arrangement of *some* entities, but not of others. Specifically, the intended implementation of FTIL is one where the following are contiguous in memory, and stored in a consistent endianity among all computation nodes (or otherwise stored in a way that allows efficient iteration over items and efficient communication of entities):

1. All fixed-sized types, and
2. All `memblock`s.

This allows the back-end implementer the necessary rigour to implement the mass operations discussed in this section most efficiently, especially when manipulating `memblock`s and `array`s, while at the same time allowing the front-end programmer flexibility, as well as the vocabulary, to specify as part of the FTIL program which parts need to be most optimised. The programmer's choice between flexibility and power is as simple as the choice between using a `list` and a `memblock`.

In general, however, the FTIL elementary types were all designed with a view that they will be easy to store efficiently in memory (even if not necessarily contiguously, or not necessarily in a consistent manner across computation nodes). The general rule is that the more efficient mass operations on these types will be, and the more efficient their serialisation and deserialisation will be, the better.

3.13 FTIL scripts

A special type of FTIL command is an FTIL *script*. Scripts are FTIL's version of a "user-defined function". They can be thought of as canned sequences of FTIL instructions, which may accept variables as parameters, but FTIL scripts also do more in that they allow flow-control and other instructions that are not allowed on the FTIL command-line.

There are two types of scripts in FTIL: *module scripts* and *non-module scripts*. In terms of the FTIL-user's view of scripts, these have only minor differences (which we delve into in Section 3.13.5). The main differences are, however, in how these are defined and how they enable the execution of privileged commands (regarding which, see Section 3.14 and Section 3.15).

Non-module scripts also form their own pseudotype (regarding which, see Section 3.5.5), whereas module scripts are simply parts of their respective modules.

FTIL scripts use a Python-like syntax. They are defined by

```
def ScriptName(param1, param2,...):
  command1
  command2
  ...
```

and the corresponding syntax for invoking them is

```
ScriptName(param1_value, param2_value,...)
```

To disambiguate, we refer to "`param1`", etc., in the script definition, as the script's *formal parameters*, whereas the "`param1_value`", etc., when invoking the script will be called the *invocation parameters*.

Script formal parameters follow the naming rules described in Section 3.4.

Still copying the Python format, FTIL script definitions may begin with an explicit literal string, like so:

```
def ScriptName(param1, param2,...):
"""
  Explicit literal string here.
"""

  command1
  command2
  ...
```

In this case, the literal string is the script's *help string*. It can be retrieved by the FTIL `help()` function, as described in Section 3.11.20.

Script formal parameters may also have default values. The syntax for this is

```
def ScriptName(param1 = default1, param2 = default2,...):
  command1
  command2
  ...
```

Default values can be literals of one of the types `int`, `float`, `bool` or `string`, or may be system constants.

3.13.1 Script parameters

FTIL is quite restrictive in terms of what can be sent to scripts as invocation parameters. Script parameters must be one of the following:

- Explicit literal constants of one of the types `int`, `float`, `bool` or `string`,
- Variables that exist in the system (including system constants). These can be of any type, but not pseudotypes,
- Yet-to-be-defined top-level variables, or
- Variables that are yet-to-be-defined attributes of a referential entity that exists in the system. (By this we mean a collection or a `transmitter`; not the module reference of a method.)

In the last case, the attribute name must be legal for the entity, and the entity must be able to accept new attributes, i.e. it cannot be immutable. In the case of collections, this means that the collection is not labelled[29]. Note that both dependent and independent variables are allowed.

The following are examples of what *cannot* be sent as invocation parameters:

- Elements,
- Pseudotype entities,
- Computations, such as, for example, nested functions, and
- Qualified names whose longest prefix is not a variable that exists in the system.

Notably, by "existing in the system" we mean "existing in the Variable Registry". It is legal (and common) in FTIL to invoke scripts with parameters whose definition's scope (and the scope of definition of their antecedents) may be narrower than the scope in which the script is invoked.

Though actually trying to use such values in subsequent computations on a node where they are not defined would yield an error, merely sending them as script parameters is allowed.

[29] Or its label is suspended and the longest prefix of the invocation parameter's qualified name is on the Modified Name List. (See Section 3.10.3 for the full details.)

3.13.2 The script environment

The execution environment of a script includes several components, some of which have already been discussed elsewhere. These are components that are managed separately for each script on the execution stack. The same script can appear multiple times on the execution stack if it is called recursively, in which case each invocation is managed separately.

The following are these per-stack-position components.

Identifiers: Each stack position maintains a mapping from identifiers (top-level names) to the variables they identify. This was discussed in Section 3.8.4.

The Confirmed Name List: Described in Section 3.10.2.

The Modified Name List: Described in Section 3.10.3.

The Protected List: Described below.

The Protected List has been mentioned previously in this document, and described as a mapping from formal parameter names to invocation parameter qualified names. In fact, it is slightly more involved than this, in that in addition to the qualified name the mapping is also to an integer number, d, as described below.

The Protected List is part of FTIL's handling of parameter passing. When a script

```
def ScriptName(param1 = default1, param2 = default2,...):
  command1
  command2
  ...
```

is invoked using

```
ScriptName(param1_value, param2_value,...)
```

FTIL must set up an execution context for the new invocation.

As a first step, each Segment Manager in the scope of the command sets up a Protected List. Each formal parameter "`paramX`" is mapped to the qualified name of its respective invocation parameter, "`paramX_value`", in the meaning of "qualified name" defined in Section 3.2.6.

Recall that for each individual Segment Manager, not only may "`paramX_value`" not correspond to an in-scope variable, it may even be the case that some or all of the antecedents of "`paramX_value`" are not defined in the Segment Manager's scope.

Each Segment Manager now proceeds to increment the protection counter (See Section 3.8.3) of every antecedent that is in scope for it, of the qualified name of each invocation parameter. The value of d for each formal parameter, as stored in the Protected List, is the number of protection counters thus incremented for each parameter. (If the qualified name sent to the script is within the scope of the Segment Manager, the value of d will be the number of antecedents, but any portion of the qualified name that is out of scope for the Segment Manager will reduce d.)

Remark for the advanced reader 3.13.1. A variable sent to a script may be an attribute of a `transmitter`. For example, if `t` is a transmitter, a possible invocation parameter for a script may be `t[n]`. On the Protected List, `n` must appear as a resolved value. Importantly, however, note that it may have a different value at each node, so each Segment Manager may actually store a different value in its Protected List.

No such discrepancy can occur with any of the values for which the protection counter is now incremented, however. Both in the case of antecedents of parameters and in the case of method-calling `objects`, all items that receive protection cannot have indirect addressing into a `transmitter` in their qualified names. For this reason, these qualified names can all be resolved by the Compute Manager, which is how the Compute Manager can maintain a mirror of all protection counters also in the Variable Registry.

This protection of each script's variables is kept throughout the runtime of a script. When the script exits (whether successfully or not), the information in the Protected List is used to decrement the protection counters again, restoring all protections to their former levels (barring any unrelated protection changes).

Note 3.13.1. This is where the value of d on the Protected List is used: in order to make sure we decrement the protection counter on exactly those variables that initially received an increment to their counters. The variables whose protection counters are decremented are the underlying variables of the d shortest prefixes of each qualified name.

The reason for the added protection that script parameters get is that while FTIL scripts may delete or redirect variables, including their own parameters, the essential meaning of each variable, what it signifies, is encoded in who its antecedents are, and this is information that needs to be protected throughout a script's execution.

Another action taken by the Segment Managers to set up the new execution context is the creation of a new identifier namespace. This is the mapping from top-level names (identifiers) to the variables they identify that is described in Section 3.8.4.

Initially, the identifier namespace includes only the script's formal parameters. Later on, each time a new local variable is defined, its name is added to the namespace, and each time a variable is deleted, its name is removed from the namespace. We refer to the names in the script's namespace as the script's *local identifiers*.

It is allowed, in FTIL, to remove any variable in the namespace that is not protected. (Some variables may be protected, however, for example `for`-loop variables).

It is even allowed to delete a variable that is identified by one of the script's formal parameters. Doing so, however, will only deallocate the underlying entity. The variable will become uninitialised but will not be deleted entirely, and therefore the identifier itself will not be deleted, either, and will not be removed from the identifier map.

It is, in general, allowed for script parameter variables to be uninitialised. No variable in FTIL that is not a script parameter is allowed to be uninitialised, however (in the sense of not being associated with an entity).[30]

The above demonstrates a distinction between script parameters and script local variables in how they are handled by the language, and it is therefore important for the FTIL executor to know which is which. One way to do this is to consult the Protected List: any item on the Protected List is a parameter, any other item in the identifier namespace is a local variable.

This is a second use for the Protected List. (A third use of the Protected List is in the implicit label-confirmation of variables returned from scripts. This is explained in Section 3.10.2.)

At the termination of a script's execution, the script's identifier namespace is deleted, so all local variables and all script parameters cease to exist. Any local identifier that is the sole identifier for a non-attribute variable triggers by this the deallocation of its underlying variable (which, in turn, may trigger the deallocation of its underlying entity, which, in turn, may trigger even more deallocations).

See Section 3.13.6 for some special handling of this termination in the case of script failure.

A major change in FTIL compared with earlier versions of this document is that now within each script the identifier namespace determines what variables can be accessed. Specifically, commands can now only refer to the following top-level names as non-script variables.

- Local variables defined in the script,
- The script's own parameters, and
- System constants.

[30] The fine print here is that script parameters are identifiers, not variables, and many identifiers may reference the same variable. A "variable that is not a script parameter" is a variable such that none of the identifiers referring to it is the formal parameter of a script.

(Namespacing of scripts is described in Section 3.13.5.)

While a script is running, the identifier namespaces of earlier scripts in the execution stack are disabled and cannot be referenced.

3.13.3 Parameter passing

When a script is invoked, the formal parameters need to be initialised from the invocation parameters. When a script terminates successfully, the invocation parameters must receive their new values.

Under normal circumstances, this is quite easy to do: when a script is called, the formal parameters become new identifiers (in the newly-created namespace) that are new names to the same variables as the invocation parameters are identifying, using the same mapping as given in the Protected List.

Once a script terminates successfully, the script's identifier namespace is erased, and as part of this the formal parameters cease to be identifiers in the system. As the calling script resumes, its own identifier namespace, in which the invocation parameters are defined, is re-activated, and they can once again access their variables, whether or not the entities these variables refer to, or the values of these entities, has changed in the interim.

Some complications do occur, however:

- The invocation parameter may be a literal constant,
- The formal parameter may have been initialised from a default value,
- The invocation parameter and some of its antecedents may not have been defined in the scope of the specific Segment Manager,
- The invocation parameter may not be defined in the scope of the specific Segment Manager, but all of its antecedents are defined, and
- The formal parameter may identify an uninitialised variable at script termination.

In this section, we describe how each of these eventualities is handled.

The general procedure when calling a script is as follows: for each parameter, if the parameter exists as a variable, the script's formal parameter becomes a new name for this variable; if it does not yet exist as a variable, but its invocation parameter is a non-atomic name (making it an attribute), the attribute is created as an uninitialised variable, and the formal parameter becomes a name for it; if the invocation parameter is an atomic name, the name is created as a new identifier to an uninitialised variable, and the script's formal parameter is another new name for it.

The general procedure when returning from a script is as follows.

First, all local identifiers are deleted as though by a "`del`" command. Necessarily, the underlying variables of such local identifiers are all non-attribute variables that have exactly one identifier, so `del` deletes both the identifier and the variable.

Next, the following is performed for each parameter, `p`.

If the parameter's underlying variable is initialised, only the identifier, `p`, is deleted. If the variable is (a) not an attribute, and (b) with no other identifiers, this triggers the variable's deletion, too, as per the usual process.

Otherwise, if the parameter's underlying variable is an attribute, we perform the following: if the variable `p` is not associated by any other identifiers, both the variable and the parameter are deleted. Otherwise (i.e., if there are other identifiers associated with the same variable), only the identifier is deleted.

In the final case (where the parameter's underlying variable is not an attribute), if this variable has no other identifier, both the identifier and the underlying variable are deleted. Alternatively, if this variable has exactly one other identifier, then the other identifier is necessarily the qualified name of the formal parameter's invocation parameter (as recorded on the Protected List). In this

case, we delete both identifiers as well as the variable. If, on the other hand, the parameter's underlying variable has more than one other identifier, we merely delete the identifier `p`.

Let us now return to the list of potential complications given above, to show how each of these is handled within the framework.

Parameter is a literal constant: This constant is made into a variable, and the formal parameter is its name. Note that the formal parameter is not considered protected. Its value is in this case shared, however, so cannot be modified directly.

Initialisation from default value: As above. Note that in this case the default value may also be a system constant, in which case no new variable is required. The system constant itself is the required variable. If the system constant is distributed, the variable is not immutable.

Parameter plus antecedents undefined: A Segment Manager seeing a parameter passed to a script that does not exist in its scope and is also missing some antecedents in its scope knows that this parameter can never be initialised by it. The Segment Manager can freely ignore the parameter. It does not exist for this Segment Manager, and any attempt to access it should fail.

Parameter undefined, antecedents defined: A Segment Manager seeing a parameter passed to a script that is not itself defined at its scope but has all of its antecedents defined in the scope cannot know whether the variable truly is undefined (and may potentially be defined within the script itself) or is simply defined in a scope that does not include the Segment Manager's node. It uses the standard procedure, described above, of handling an uninitialised parameter value, as though the parameter may still be initialised during the course of the script. If, in truth, the parameter already is initialised, but in another scope, the semantics of the language will make attempts to assign to this parameter at the scope of the Segment Manager's node fail during prerequisite checking.

Uninitialised at script termination: This is handled as per the standard process for script termination, as discussed above.

In total, the semantics of FTIL allow every variable to be used as an input variable, an output variable, or an input-output (modified) variable, without this requiring any form of pre-declaration in the script syntax. Undefined variables that are sent to scripts (e.g., to serve as output variables) may even remain unused, in which case they simply cease to exist at script termination.

3.13.4 Flow control

Flow control, touched on briefly in Section 3.7, is unique to the script environment. For all Compute Manager/Segment Manager jurisdictional details regarding flow control, see the discussion in Section 3.7 and Section 2.11. We do not repeat it here. (In brief: flow control parameters are resolved by the Compute Manager. If they are not shared, they are implicitly broadcast to the Segment Managers, making the commands privileged on the coordinator node.) Here, only the FTIL syntax and its semantics are discussed.

FTIL scripts have three mechanisms for script flow control:

- "`if`" statements,
- "`while`" loops, and
- "`for`" loops.

Each defines its own command block:

```
if condition:
  command1
  command2
```

```
  ...

while condition:
  command1
  command2
  ...

for i in loop_range:
  command1
  command2
  ...
```

An "`if`" block first evaluates its condition. If the condition is true, the command block is executed. Otherwise, it is skipped. The condition is not rechecked once block execution begins.

An "`if`" block may also have an optional "`else`" portion, as

```
if condition:
  command1
  command2
  ...
else:
  alternate1
  alternate2
  ...
```

Here, if the "`if`" condition is true, the "`else`" portion is skipped, but if it is false, the main block is skipped but the "`else`" block is executed.

A "`while`" block first evaluates its condition. If the condition is true, the command block executed. Otherwise, it is skipped. The condition is not rechecked during the command block's execution. However, where "`while`" differs from "`if`" is that at the end of the command block's execution, execution returns to the "`while`" statement, and everything repeats: the condition is re-evaluated, etc..

A "`for`" block first evaluates its "`loop_range`". This must be evaluated as a `list` or as a `set` (including a `nodeset`). However, if it is given as a `set` it is implicitly converted to a `list`, sorted as in standard `set`-to-`list` conversions.

Note 3.13.2. Support of `array`s or `memblock`s for loop variables is Priority 3. These can both be converted to `list`s if need be, recalling that the FTIL `for`-loop construct is not meant for massive loops.

The list that forms the range of the `for`-loop is evaluated only once, when the `for`-loop block is initially entered.

Once the range has been ascertained, FTIL creates a new variable, named as the "`i`" parameter given to the command (See example above), which must be a new top-level name, not corresponding to any existing, active identifier, and assigns to it the first item in the list.

The type of `i` is the type of the list element, and it is "shared". The variable is marked as protected. Its scope is the scope in which the `for` command was executed, but its value also exists in the Variable Registry.

This wide scope allows `i` to be used in every execution context. For example, it can be used in indirect addressing into a `catalogue`.

Once `i` has been set, the commands of the loop block are executed as usual. At the end of the block, execution returns to the beginning of the block and `i` receives the value of the next item in the list.

This goes on until the list is exhausted, at which time `i` is deallocated and execution continues to the next instruction.

Note 3.13.3. Note that while conditionals and loop ranges must be evaluated at the coordinator node, the use of Variable Registry data allows one, in some cases, to loop effectively over lists even when they pertain to variables that are not defined on the coordinator node.

For example, if `c` is a `catalogue` that is define in a scope that does not include the coordinator node, one can still loop over `c.keys()`, because these are available in the Variable Registry. Similarly, if `t` is a `transmitter`, one can loop over `t.metadata.scope`.

3.13.5 Module vs non-module scripts

FTIL supports two different types of scripts: *module scripts* and *non-module scripts*. These are also sometimes known as *signed scripts* and *unsigned scripts*, respectively.

A non-module script is a script that is typed in at the command line (or, completely equivalently, created through an `exec` command).

A module script is one that is created as part of the loading of an entire module via an `import` command (See Section 3.14).

From an execution perspective, the two types of scripts are quite distinct: module scripts are shared with the nodes that execute them, for them to verify their proper overall execution, globally in the system (as detailed in Section 2.11.3). They can be *certified*, a process described in Section 6.3 that then allows them to run in privileged mode (See Section 2.11.4) and execute sensitive commands (See Section 3.15 for details).

Non-module scripts, by contrast, are little more than canned instruction sequences, convenience features for the local FTIL user. Their details are not shared with the peer nodes, they are executed in a "looser" execution model (as detailed in Section 2.11.2), cannot be certified, and allow no special execution privileges.

From the perspective of the FTIL programmer, the distinctions between module scripts and non-module scripts are twofold.

First, non-module scripts behave far more like variables. They are created individually and can be deleted using a `del` command. As variables, they are unique in the FinTracer system in that they are associated with a specific user of the system. All other variables are equally visible and accessible to any coordinator-node user and the system, but non-module scripts are created by a specific user and are only visible and available within the FTIL sessions of that specific user.

Note that when deleting a script using a `del` command, just like in any other reference to the script other than when invoking it, one precedes the script name by "`::`". For example:

```
del ::NonModuleScriptName
```

This ensures that a script name can never be confused with any non-script variable.

By contrast, module scripts have no variable-like behaviour of their own, but the module they belong to, as a single unit, is a variable in the system, following general variable rules in terms of its availability. It cannot be explicitly deleted, but gets automatically deleted when it is no longer referenced (as explained in Section 3.14).

Importantly, each module defines its own module-script namespace, whereas the non-module scripts are part of a (per-user) non-module-script namespace. Although both types of scripts can call other scripts, including recursively, module scripts can only call scripts that are part of the same module, whereas non-module scripts can only call non-module scripts that belong to the same user. Names in each namespace have no impact on any other namespace: namespaces do not clutter each other.

The second distinction between module scripts and non-module scripts from an FTIL-user's perspective is in the availability of *methods*.

Code inside a module script—and only inside module scripts—can include the following special command:

```
obj.attr = ::ModuleScriptName
```

This looks like a simple assignment, but the "`::`", revealing that the right-hand side is a script name, signals that it is not.

In the above line, "obj" must be an unlabelled object[31], ModuleScriptName must be the name of a script within the same module as the currently-executing command and expecting at least one parameter, and attr must be an attribute name that would be legal to be assigned into. (It must have a legal attribute name, and must either not be previously defined, or is previously defined but is not protected and has the same scope of the command, so can be deallocated and redirected.)

The command must be run at the same scope as the definition scope of "obj".

What the command does is create a new entity of pseudotype "method" in the variable obj.attr. Such entities are non-copyable (although if an object needs to be copied due to labelling or by becoming a module proxy, the entity might end up being referred to by more than one variable).

Such attributes can be deleted by del commands, like all other object attributes (and this does not use the "::" prefix). As usual, deleting only works if the object is not labelled (or if its label is suspended, with the usual conditions regarding the Modified Name List) and when the del command is executed at the exact scope of the method.

Once a method has been created, it can be invoked using method-calling syntax, like so:

```
obj.attr(param1, param2,...)
```

This invocation command, in turn, calls the underlying function with parameters

```
ModuleScriptName(obj, param1, param2,...)
```

which is to say that the calling object's name is inserted as a first parameter to the underlying function call.

Method invocation differs from direct function invocation in two important ways, however:

1. In standard function invocation, the antecedents of all parameters become "protected". In method invocation, this protection is extended not just to the antecedents of the calling object, but also to the calling object itself: a calling object's variable cannot be deleted or redirected within its own method call.

2. One can invoke "obj.attr()" whenever obj is a label-confirmed object (in the sense of Section 3.10.2) within the command's entire scope. More specifically, it is allowed for code to invoke a script that was defined in a different module (and, more generally, outside of the code's own script namespace), as long as this call is done via a method. Thus, the restriction of being in the same namespace (used for script calling) is replaced for method calling by the restriction of being label-confirmed.[32]

In addition to the above, module scripts also support the modifier "NOCERT". This appears in the following syntax:

```
def MyScript(argument1, argument2, ...) NOCERT:
  commands
  ...
```

A script labelled as NOCERT cannot be certified. Any certification given to it is ignored.

The modifier NOCERT is useful for writing helper scripts that, in and of themselves, may provide general functionality that, under uncontrolled conditions, may be unsafe, but are written in order to be used in particular, safe contexts. By labelling the scripts with this modifier, the FTIL programmer can be sure that the helper scripts will never be accidentally certified, potentially opening the door to unsafe use by other FTIL programmers later on.

[31] Or otherwise, as usual, its label should be suspended and rm should be a qualified name on the Modified Name List. (See Section 3.10.3 for the full details.)

[32] However, again, we stress that we mean this in the technical sense of Section 3.10.2. The name must appear on the Confirmed Name List. Specifically, if the calling script is running in privileged mode, it will not be able to execute a method of an rm whose label is weak.

3.13.6 Aborting

Recall from Section 2.9 that FTIL commands may *abort*, and that this abort may either be a case of command *exit* (i.e., a designed exit point) or a case of command *fail* (i.e., an unexpected abort).

Many reasons may cause a command to fail. Examples are:

- The command is an `assert` and the assertion condition is false,
- The command is a `confirm_label` and was invoked on an unlabelled collection or on a collection whose label does not match the expected value,
- The command did not pass its prerequisite checks,
- The command passed its prerequisite checks, but failed for other reasons. For example, its parameters had invalid values for the operation, or
- The user manually aborted execution, by pressing "`^C`".

Reasons for a command exit, on the other hand, are limited to failing a "`limit`" or "`confirm`" test, which are commands that allow the FTIL programmer to test for expected, planned situations where an abort point should be designed.

For example, if a user query returns too many results, this may void any privacy preservation guarantees provided by the system. Instead, the programmer can test whether the number of results will be too large and abort preemptively.

Both types of aborts use the following process.

First, the FTIL executor outputs (to the FTIL user, but presumably also to any back-end logging mechanism) the state of the entire execution stack and all its program counters at the time of the abort, to give the user the best possible information to understand the abort and potentially debug any issues. The aborting command will also have some informative message regarding the cause of the abort (from each executing node separately), and this is also output.

Next, the state of the system is rolled back to its last save point, in order to ensure its coherency. This roll-back may be only to the beginning of the present command, or it may roll the system back substantially, e.g. to the beginning of a `modifying` block.

If the state to which the system is rolled back is not the beginning of the present command, the system will also output the state of the execution stack and program counters at the time of the last save point, to ensure the user knows exactly what the new state of the system currently is.

To simplify flow control and allow better algorithm verifiability, FTIL offers no form of exception handling. Any abort terminates not only the command but also the script calling it, as well as all other scripts up the execution stack. The user is therefore at this point immediately returned to the command prompt.

The difference between a "fail" and an "exit" is in their clean-up procedures.

When a command aborts over an "exit", this causes all calling scripts to terminate immediately, but the local identifiers in each script are cleaned up exactly as usual, i.e. as described in Section 3.13.2 and Section 3.13.3. Local variables, for example, are deleted with their identifiers, and their underlying entities are unlinked (and potentially deleted as well).

When a command "fail"s, on the other hand, the clean-up procedure for local identifiers is very different. The idea for it is that FTIL wants to create a variable "dump", which may potentially allow the user to interrogate the cause of the failure.

Such a dump does not create or delete any variables nor any entities. Instead, the way it works is that FTIL takes all identifiers from the entire execution stack, name-mangles them, and makes them into identifiers in the root of the now-empty stack (where they remain accessible even once at the command prompt).

If an identifier was originally "`idname`", it will now be renamed, at the root of the stack, to

```
dumpXXXX:scriptname1:scriptname2:... :idname
```

where "`dump`" is a string literal indicating that this is a dump object (Users should preferably avoid using this name in command-line-level identifier prefixes), "`XXXX`" indicates a time-stamp (which should ensure uniqueness of name prefix for each dump event), "`scriptname1:scriptname2:...`" are the names of the scripts that were on the calling stack below the target identifier at the time of the abort, the colons are literal colons, and "`idname`" is the original identifier name.

If any script, "`scriptname`", is called via a method invocation syntax, its name will appear in the name mangling as "`_signature::scriptname`", where "`scriptname`" is the name of the script (not the method), "`signature`" is the (zero-padded, lowercase, hexadecimal) signature of the script's module, and the rest are literals.

Note 3.13.4. This naming convention makes it relatively straightforward to clean up the command-line namespace once the dump objects are no longer needed.

To delete all dump objects in a single command, the user can run "`del dump*:*`". (The colon disambiguates from user variable names, even if these begin with the string "`dump`".)

To delete all dump objects from a specific dump event, one can use "`del dumpXXXX:*`".

Because of the use of timestamps in the target identifiers' names, it is unlikely that when renaming an identifier as part of a dump the new name will clash with an already existing identifier. However, if such an occurrence does happen, the existing variable is left untouched and clean-up reports an error.

Note 3.13.5. Note that colon is an allowed character in a variable name, though its use is reserved for this context, so typically no clash should occur between dump objects and other identifiers. Colons are only forbidden as the first or last characters in a top-level variable name.

The following are not "dumped" in this way:

- Loop variables,
- System constants (because they are not on the stack to begin with),
- Command-line variables (because they are at the root of the stack to begin with), and
- Pseudotypes (because they cannot be associated with local identifiers, in any case).

Note 3.13.6. This renaming of the identifiers happens not just at the coordinator node but at every node. It is important, therefore, to ensure that all nodes agree for every identifier in their scope on the same mangled name that it should be mapped into. For this reason, it is important that every node in scope is aware of every script on the calling stack, including even non-module scripts.

This is why, despite the fact that peer nodes are generally oblivious to non-module scripts and execute their individual commands rather than the scripts as a single unit, it is still important that all nodes in scope are made aware when a script is called and when it terminates, even for non-module scripts.

3.13.7 Scripts vs commands

Though this is probably understood by now, after the full discussion of FTIL scripts, it is worth pointing out explicitly that the semantics of FTIL user scripts are not the same semantics as those of built-in commands.

Here are some of the ways in which commands and scripts differ:

1. Commands accept parameters in whichever way makes the most sense for that command, be it by value or by reference, whereas scripts can only use variables as arguments (never elements), and these are passed always using the process described in Section 3.13.3.
2. Commands can return values, in which case we call them *functions* (See Section 3.11). Scripts cannot, and simply use some of their parameters as output parameters.
3. Commands can accept as inputs the results of operations and of functions, allowing function and operator nesting. Scripts do not accept operations to be evaluated as parameters.

3.14 Modules

FTIL uses its own system of code libraries, called *modules*. These can be loaded from URIs, and can even reference each other in order to enable modular library programming.

In addition to this convenience that is offered by modules, they are also critical to FTIL's functionality. All inter-node communication in FTIL is covered by privileged commands and has to be executed in privileged mode (See Section 3.15), but one can only enter privileged mode from inside module scripts.

Module scripts and the module and method pseudotypes have all already been introduced in Section 3.5.5. Module proxy objects have similarly been introduced in Section 3.5.3, and FTIL's method calling syntax and semantics have been expanded on in Section 3.13.5. We will not repeat this material here.

Additionally, each script defined in any given module may or may not be *certified* in any given node. The operators of each node can decide individually whether or not they wish to certify any such script, so whether a module script is certified or not may be different for each node. The technical process of how to certify a script is discussed in Section 6.3, and it, too, will not be reviewed in this present section. Important for our present purposes is only that this certification process does not involve the FTIL user and the module files at all. It happens (on a technical level) solely at the individual nodes.

In this section, we discuss how one defines a new module, how one loads it, how one uses it, and how it is ultimately unloaded.

In order to create a module, one begins by creating a *module file*. This is not necessarily a file-system file, but is a resource whose contents can be read as a string from some URI accessible to the Compute Manager, which is at the coordinator node.

Presumably, the resource will actually be found in the Auxiliary DB at the coordinator node.

A module file has the following structure. It begins with the line

```
FTIL1.0
```

which identifies the FTIL version used. This line will be verified on loading, to make sure the code is in a compatible FTIL version to the running version. This is a form of future-proofing, as it allows the language to disregard legacy signed scripts in the future, if new language features are introduced that change the semantics of old scripts.

Ignoring any whitespace lines, the second line in a module file is

> `module` *modulename*

This defines "*modulename*" to be the name of the module. Module names follow the same restrictions as `object` attribute names.

Following this, the module file includes zero or more lines in the format

> `using` x

where "x" is the name of a module used by *modulename*.

A single "`using`" command may specify more than a single used module, by listing their names as a comma-separated list like so:

> `using` x`,` y`,` z

It does not matter in what order the dependent modules are specified, nor whether any are listed in a single `using` instruction or not. No argument to the `using` directive may be the same as the *modulename*, however.

The remainder of the module file can only include script definitions. No command outside of script definitions is allowed.

Each script must have a unique name, and exactly one must be called "`__init`", which is a special name. (Other than in this context, script names cannot begin with underscores.)

The order of the scripts in the file does not matter, including where the "`__init`" script is defined in the file.

If the module file does not use the "`using`" directive, the `__init` script must accept exactly one parameter, like so:

```
def __init(self):
```

If the "`using`" directive is used, `__init` accepts exactly two parameters:

```
def __init(self, prerequisites):
```

Module files are loaded into modules using the "`import`" command. An invocation of `import` looks like this:

```
import "file://path/filename.ftil" to modules
```

where the string in quotes is the URI of the module to be loaded. Its name must be given as an explicit, literal string. The "`to modules`" part is optional, but if it appears then `modules` should be an unlabelled `object` to which it is legal to now assign, at the scope in which the command is run, the attribute *modulename* (i.e., the name of the module as defined inside the module file).

Otherwise, *modulename* will be created as a top-level name, and that, too, must be legal to do.

The `import` command can only be executed from the command line, cannot be used as a building block inside larger commands, and does not return a value. However, it does accept an "`on()`" modifier that allows it to be run in a given scope. Effectively, this will be the scope at which the module is defined and can from this point on be used. Its prerequisite modules (those listed under "`using`" clauses) must have been previously `import`ed in exactly this scope.

The process followed by the FTIL executor when running `import` is as follows.

1. The module file is read, the first line is verified and the FTIL version number is checked for compatibility. (Presently, only "`FTIL1.0`" is accepted.) The other conditions for a valid module file are also verified.

2. The module file is sent to all Segment Managers in scope, where it is used to initialise a variable of type `module`. This variable manages the contents of the module file, computes its signature, ascertains which of its scripts are certified, and later on uses this knowledge to verify that the execution of the module's scripts by the Compute Manager is done correctly, as detailed in Section 2.11.3.

3. A copy of the script file is retained by the Compute Manager, for actual script execution later on.

4. If the `object` attribute "`modules.modulename`" exists (when using the optional "`to modules`" directive), or, alternatively, if the top-level name "`modulename`" exists (when not using this directive), it is deallocated. (If it cannot be deallocated, e.g. because it was defined in a scope different to the scope of the command, that is an error and the script is not loaded.)

5. If the two-parameter version of "`__init`" is used in the file, the FTIL executor now prepares a new, unlabelled `object` at the same scope as the `import` command, and assigns to it attributes according to the `using` directives used. If, for example, the file contains the directive "`using x`", an attribute "`x`" is now created to the new `object`, and it is assigned the value of "`modules.x`" (if "`to modules`" was used) or the top-level variable "`x`" (if not). Any attribute thus assigned must be a module proxy. If it is not, the command fails.

6. The `__init` script is now executed, at the same scope as the `import` command, with the new `object` (if it was created) used as the second script parameter and an uninitialised value given as its first ("`self`") parameter.

7. At the end of its execution, the `__init` script must return in the parameter "`self`" a labelled `object`, defined over the same scope as the command's execution scope, and containing only method attributes and member attributes that are, themselves, module proxy objects. If it returns anything else, the command fails.

8. The module's signature is made into the return value's "`.metadata.signature`" attribute, signalling to the system that this return value is now a module proxy.

9. The returned value is assigned into the top-level name "*modulename*" (if "`to modules`" was not used) or "`modules.`*modulename*" (otherwise). If this assignment fails, the `import` command as a whole fails.

Note 3.14.1. The idea here is that the FTIL user can define an (unlabelled) `object`, "`modules`" to be the place to which all module proxies are attached. Running all `import` commands with a "`to modules`" suffix then creates the imported modules as attributes of the `modules object`, and this becomes, also, the place where each loaded module finds its prerequisites.

Note that this `import` creates two entities in the system: a module and a module proxy object. The module is unnamed, but the module proxy object is named according to the name specified for it in the module file, and is placed according to a location specified by the user during the `import`.

The module is reference counted, so it erases itself if its reference count drops to zero. Initially, however, if the module proxy contains method attributes, these methods reference the module, preventing it from deleting itself. (The module proxy itself knows the module's signature, but does not contain an actual reference to it. If the proxy has no methods, the module will deallocate itself immediately.)

As an example of how this works, consider that if we run `import` and then immediately erase the module proxy, this will cause the underlying module to erase itself, because its reference count will by this have dropped to zero. However, if before erasing the module proxy one runs a module proxy method that creates a new `object` with a new method, belonging to the same module, then one can erase the proxy object and the module will remain. The module will only erase itself once it is no longer needed.[33]

Note 3.14.2. If this is your first introduction to FTIL modules, it may sound strange that one should call a method of the module proxy object in order to create another `object` with additional module scripts as method attributes. As Chapter 4 demonstrates, however, this is actually the standard way to use module proxy objects in FTIL. They are merely the gateway that provides an interface to the module, with the proxy's methods being this interface. To actually work with a module, one generally wants to create entities that are not module proxies, by use of the module proxy's methods.

As another example, suppose that we wish to inspect the module's signature. We can either do this from the module proxy object, using, e.g.,

```
modules.modulename.metadata.signature
```

or can get this information from any method created by the module. For example:

```
modules.modulename.MyMethod.module.signature
```

As a final point regarding modules, consider what happens when one repeats the same `import` command several times. Each command creates a new copy of the module and a new copy of the proxy object.

It is necessary to run the `__init` script every time `import` is called, and to generate a new proxy object every time `import` is called, but it's a good idea not to create unnecessarily a new copy in the system of an existing module.

[33] The module will also only erase itself from those nodes where it is no longer needed. It is possible for a module to be unloaded from some nodes but not from others.

Instead, the system can retain a record of the signatures of the modules that are presently loaded, and if a new module is `imported` that matches one of the signatures, instead of creating a new module, it is possible to direct the new proxy object to the existing module, instead.

We call this *lazy module loading* and rate it a Priority 2 feature.

Note 3.14.3. It is important to only perform lazy module loading based on a match in signatures. A match of names is not enough.

3.15 Privileged execution

Privileged execution has already been mentioned many times, both in this chapter and in Chapter 2.

To recap, FTIL uses *modules* (described in Section 3.14) that define *module scripts* (described in Section 3.13.5). These module scripts can be *certified* by each node, though whether or not to certify any given script of any given module is a decision taken independently by the operators of each FinTracer node. The technical process for doing so is described in Section 6.3.

Whether or not a script is certified impacts whether FTIL executes (at that particular node) in *privileged mode* (See Section 2.11.4) or not (See Section 2.11.3).

Specifically, some FTIL commands, known as *privileged commands* behave differently depending on whether they are executed in privileged mode or not, and, indeed, may simply fail if executed in non-privileged mode.

Privileged commands are all different, however, and each interacts differently with privileged mode execution. The specific ways in which each command requires privilege can be found in the sections describing the commands.

Specifically, one can divide privileged commands into the following categories:

Inter-node communication: See Section 3.11.6,

Flow control: See Section 3.7.1,

Scoping: See Section 3.7.2,

Labelling: See Section 3.10, and

Method invocation: For general method invocation, see Section 3.13.5. For a specific focus on privilege, see Section 3.10.2.

Flow control commands and scoping all share roughly the same behaviours with respect to privilege, but other than that each case is unique, with each of the 2 inter-node communication commands and each of the 3 labelling commands being a unique case.

The one piece of the puzzle that is still missing, and to which this present section is dedicated, is to explain the exact interaction between module script certification and FTIL's execution mode at each node.

When one invokes a module script in FTIL, the FTIL executor at each node checks whether the script is certified in that node. If the script is certified, this elevates execution to privileged mode on that individual node.

If the script is not certified, however, then if the script was run from the command line or from a non-module script, then execution continues in signed-script mode. If, on the other hand, the script was executed from another module script (including in the case of method invocation) then the execution mode remains as it was in the calling script, which may be either signed-script mode or privileged mode.

There is no adverse effect to having some nodes execute in privileged mode while others are in signed-script mode.

Once execution was elevated to privileged mode, it stays at privileged mode until the script finishes running, whether by successfully completing or by aborting. In particular, if additional scripts are called within the main script, they are executed in privileged mode whether or not they have been certified themselves.

Note 3.15.1. The rationale behind this system of privilege management is that different participants in the system may wish to certify scripts at different granularity levels. Consider, for example, a script `A` that runs scripts `B` and `C`, each of which executes a privileged command. It is possible that some RE will see `B` and `C` as safe operations and will be happy to certify them for running, while another RE will only be willing to certify script `A`, indicating by this that it only deems `B` and `C` safe to run in this one particular context.

This process occurs, in exactly the same way, whether a script is invoked through a script call or whether it is invoked through a method call.

Note, however, that for a method call to succeed at all, in a privileged execution mode, that method will have to belong to a label-confirmed `object`, which (in a privileged execution mode) in turn requires the `object` to have a strong label, meaning that the `label` command was executed for it in privileged execution mode.

Thus, while the script underlying the method call may not have been individually certified, the node's certifier would still have had to certify, directly or indirectly, that the `object`, with all its methods, is fit for the purpose that its label purports it to be fit.

The difference is that certifying the individual script signifies that the script is safe, and may use the privileged operations in *any* context, whereas certifying the label of the `object` merely means that it is safe to be used in *some* contexts, namely the contexts indicated by its label.

Note 3.15.2. The rationale here is that if certified script `A` calls non-certified script `B`, these two must belong to the same module, so whoever certified `A` would have been able to inspect `B` and determine that its use is safe specifically in the context of `A`.

If anyone attempts to change the module later on, this will alter the module's signature and invalidate the certification.

Consider, now, on the other hand, a situation where certified script `A` calls a method `obj.B`, this method may belong to a different module, which the certifier may not have access to. Even if the certifier has access to it, there is no guarantee this other module will not change in the future. (A change in the other module will not change the signature of `A`'s module, so `A`'s certificate will still be valid.)

The reason the certifier of `A` can nevertheless certify `A` is that the certifier of `obj`'s label ascertains what `obj.B` does. For example, if `obj` is labelled inside its own module (as would normally be the case), whoever certifies `obj`'s labelling does have access to the code of `obj.B`, and if that code were to change this would change the signature of that other module, and render the certificate of `obj`'s labelling code invalid.

The result is that `obj`'s label will become weak, it will fail label confirmation at `A`, and an attempt by `A` to run `obj.B` will fail.

Remark for the advanced reader 3.15.1. Continuing the previous note: One can also try to change `obj`'s module after `obj` had already been created and labelled. This will also not work, however. The methods of `obj` carry a reference to their original module, and this reference will remain even if a new module (with a different signature) will be loaded using a same-named module proxy object. The methods of `obj` do not go through the module proxy object in order to invoke themselves.

Certificates in each node accumulate independently of what code is run. There is no adverse effect for the system to encounter certificates that do not match any known module scripts.

Note, just as importantly, for the privileged commands, that just like in some contexts they fail unless executed in privileged mode, in other contexts they may fail unless executed in *non-privileged* mode. Examples of cases of such failures that may occur in privileged mode are:

1. In a privileged execution mode one cannot execute a method of an object that is not label confirmed as having a strong label, and

2. One cannot run "`modifying(obj)`" on an `object`, `obj`, that is not label confirmed as having a strong label.

Also, being in privileged (or non-privileged) mode might not even by itself be enough. For example, if "`modifying(obj)`" is run at a different scope than the scope of `obj`, the script containing the "`modifying()`" directive must be individually certified in every node within the scope of `obj`. Note that this is a highly restrictive condition, not just in the fact that certification is required for the entire scope of `obj` (rather than just at the scope of the command) but also in that certification is required for the specific script, and it is not enough for the current execution state to be privileged.

Failure of privileged commands, just like failure of any other command, leads to aborting the entire execution stack, with no way for the FTIL programmer to prevent this. It makes no difference if the failure was due to over-privilege or under-privilege.

Chapter 4

Example algorithms

The purpose of the framework described in Chapter 2 and Chapter 3 is to support and facilitate the implementation of secure, distributed, privacy-preserving algorithms for nation-wide financial crime investigation that we wish to run on the Purgles Platform, and in particular on the FinTracer system.

In this chapter, we return to the many FinTracer-related algorithms that have been discussed in previous documents, re-introduce them, and demonstrate how they can be implemented easily, securely and efficiently within the framework described.

This serves several purposes:

1. It provides a uniform, up-to-date and rigorous description of the algorithms in question,
2. It demonstrates that these algorithms, many of which we will want to have running on Day One of deployment, are feasible within the framework described,
3. It exemplifies the limitations of the framework by demonstrating those algorithms that are not as good a fit with it, and
4. By a combination of the above, it provides us with better intuition regarding the capabilities and limitations of expanding the system, as more algorithms are potentially introduced in the future.

The fit of our algorithms to the framework described will be demonstrated by providing actual FTIL code that implements these algorithms. The idea is for such code to be provided to the system as module scripts, upon its deployment, for REs to decide which of them they are willing to certify. Additional modules can be programmed as the capabilities of the system need to be expanded, but the code provided here covers all of what we expect the prototype to support and more.

Note 4.0.1. In previous releases of this document, some code segments documented algorithms that are no longer—and already then were not—advocated for use as part of the FinTracer algorithm suite. These originally appeared in order to demonstrate the versatility of FTIL. Given that this is no longer the purpose of this chapter, and given that these code segments actively exemplified algorithms that we do not recommend, these outdated algorithms have now all been removed.

We begin with a description of the plain-vanilla FinTracer tag propagation algorithm. This algorithm is based on the following idea. Suppose that we assign to each account in Australia a value. We can conceptualise these many values as a single vector, `v`, such that each position in the vector corresponds to a particular Australian bank account, and the value of `v` in the position is the value associated with the account. We will call such a `v` a *tag*.

The way to actually store `v` is as follows.

1. Each RE stores only the parts of `v` relevant to the accounts it manages;

2. The tag values themselves are stored individually encrypted using a semi-homomorphic encryption algorithm. (For reasons of computational efficiency, we are using additive ElGamal over the Ed25519 group.) The private key to this encryption algorithm is only known to AUSTRAC and not to the REs managing the accounts; and

3. Only accounts for which tag values need calculating are actually stored. The rest are assumed to be zeroes.

The idea is now for the REs to jointly compute a function over the tag values, utilising the fact that some operations (most notably addition) can be performed on the encrypted values without the party performing the operations learning any new information. The results can then be forwarded to AUSTRAC, which is the only party that can actually decrypt and read them. In this way, the REs gain no information about the final result while AUSTRAC gains no information regarding how it was computed.

Which functions can be computed in this way? If all we have are addition operations, we can essentially compute any *linear* function we want. This is to say that if we start with some v then what we can compute is any Mv, where M is a matrix of our choice.

FinTracer algorithms are largely about various ways to define M.

Let us begin by constructing the simplest of all possible Ms. Namely, let G be some directed graph over the set of accounts such that whether or not the edge (x, y) exists in the graph is a fact that each of $R(x)$ and $R(y)$ can determine independently, where $R(x)$ and $R(y)$ are the REs managing accounts x and y, respectively. As illustrative examples, (x, y) may exist in G if

1. Account x has transferred money to account y in a given timeframe and above a given amount threshold, or

2. Accounts x and y have had money transferred between them in *some* direction, also in a given timeframe and above a given amount threshold.

Let M_G be the adjacency matrix of G. If v is an $n \times 1$ column vector, with n being the total number of accounts, M_G is an $n \times n$ matrix, where the cell $M_G[\tilde{y}, \tilde{x}]$ equals 1 if $(x, y) \in G$ and 0 otherwise. Here, $\tilde{x}$ and $\tilde{y}$ represent the positions in vector v corresponding to accounts x and y, respectively.

Calculating $M_G v$ is in this case possible to do. We will discuss multiple methods for it, but let us begin with the most straightforward implementation: for each $(x, y) \in G$, let $R(x)$ send $v[\tilde{x}]$ to $R(y)$. Once all sent values are received, $R(y)$ sums all values up (which can be done under our semi-homomorphic encryption) and stores the result in position $\tilde{y}$ of a new vector, w. The resulting w equals $M_G v$.

In the following subsections we begin by implementing the algorithm described above in FTIL, and then show how it can be used as a building block and expanded further.

4.1 Key management

4.1.1 Setup

For starters, AUSTRAC needs to generate a new ElGamal encryption key, which will be used to encrypt all tag values. The following code sets up such a new key. It also generates two shared constants, "`one`" and "`zero`", which will be used in the algorithm (e.g., for setting up the initial value of v. Finally, the code creates a new variable called "`zeroes`", which will later act as a stockpile of encrypted zeroes.

The reason such a stockpile is beneficial is that the calculation of $M_G v$ requires a great many encrypted zeroes, and their generation may well take up 99% of the algorithm's entire execution time. Given that these zeroes do not depend on the choice of G or v, it is prudent to calculate

these in advance, offline, so as to speed up the online portion of computation by a factor of roughly 100.

Note that each node will generate its own stockpile of encrypted zeroes, for which reason the public key must be shared among all nodes.

```
def NewFinTracerKey(shared_pub_key, priv_key, one, zero, zeroes):
  """
    Usage:
      NewFinTracerKey generates a new private-public key pair and other
        preliminaries later used in the propagation of FinTracer tags.
    Inputs:
      None.
    Modified:
      None.
    Outputs:
      shared_pub_key -- The public key, shared by all nodes.
      priv_key -- The private key, known only to the Coordinator node
                    (i.e., to AUSTRAC).
      one -- An encrypted "1", shared among all nodes.
      zero -- An encrypted "0", shared among all nodes.
      zeroes -- An initially-empty stockpile of encrypted values, distributed
                  among all nodes, meant to provide a storage buffer for the
                  many encrypted zeroes required for tag propagation.
  """

  on(CoordinatorID):
    NewElGamalKey(_pub_key, priv_key)
    _temp1 = ElGamalEncrypt(Ed25519(1), _pub_key)
    _temp0 = ElGamalEncrypt(Ed25519(0), _pub_key)

  one = broadcast(_temp1)
  zero = broadcast(_temp0)
  shared_pub_key = broadcast(_pub_key)

  # We initialise 'zeroes' as an empty list on every node.
  zeroes = list<ElGamalCipher>()
```

Note 4.1.1. The code above creates a `rm` stockpile in all Segment Managers, including in the coordinator node. The stockpile at the coordinator node itself is the least used one, and in some cases may not be needed at all. It would have been trivial to avoid creating it in the coordinator node by an "`rm`" directive, after defining "`rm`", but it certainly doesn't hurt to have it there, so the code is written to create it. As we shall see, we can populate the stockpiles in each node independently, so this does not waste any computation resources.

Let us now fill up the zeroes stockpile. In the original plans for FinTracer, zeroes were created by the coordinator node and transported to all other nodes. The reason for this was that it was thought such a choice would spare all peer nodes from having to host GPUs. We have since decided that for reasons of cryptographic security it is preferable that each node generates its own encrypted zeroes. Below is the code to support this preferred method.

Notably, in FTIL functionality is typically exposed to the user as object methods rather than as standalone scripts. We will therefore assume that all the elements that were generated by `NewFinTracerKey` are now managed by a single `object`, which we shall name `keymgr`. Ultimately, it is necessary to have such an `object`, in order to ensure that the various generated elements are all kept coherent, e.g. that at all times the language can guarantee which pubic key corresponds to which private key.

```
def AddZeroes(keymgr, num = 10000000000):
  """
    Usage:
      AddZeroes populates the "keymgr.zeroes" stockpile with locally-encrypted
        zeroes, working in parallel on all nodes in its scope.
    Inputs:
      num [Default: 10000000000] -- How many zeroes to add to the stockpile.
    Modified:
      keymgr -- The key manager object into whose stockpile the newly-encrypted
                zeroes are appended.
    Outputs:
      None.
  """

  confirm_label(keymgr, "FinTracer Key{Signature}")

  # We wish to encrypt an entire list. We could have done this through a mass
  # operation, but are doing this here by invoking a form of "ElGamalEncrypt"
  # that encrypts an entire list. In actuality, the preferred choice of
  # implementation should be a matter of execution efficiency.
  _encrypted = ElGamalEncrypt(list<Ed25519>(num, Ed25519(0)), keymgr.pub_key)
  modifying(keymgr) keymgr.zeroes.append(_encrypted)
```

Note that there is no need to always add the same number of zeroes in every node. To create additional zeroes on only some nodes, merely invoke `AddZeroes` with an "`on()`" directive.

We've limited the use of "`modifying()`" to the smallest code section possible, essentially covering only an atomic command. This is not strictly necessary, because "`modifying()`" blocks preserve coherence of their variables by providing transactional guarantees. Any potential issues that the system might run into in trying to execute the block would merely cause the system to roll back to the start of the block (or to an earlier command). This makes "`modifying()`" blocks "atomic" from the perspective of the FTIL programmer.

4.1.2 ElGamal refresh

Before moving on, let us consider how we will use the key manager's stockpile of zeroes.

The general idea here is that we do not want any cipher values communicated out by any party to ever repeat themselves in a way that will make the repetition reveal any information. To do this, we *refresh* each cipher value before it is transmitted, which is to say that we take a never-before-used encrypted zero, and sum it to the cipher value.

As argued in Section 3.11.13, we will probably need a specialised "`Refresh()`" command in order to implement this addition.

Below, however, is a native-FTIL script that can perform a refresh on memblocks and lists of ciphertexts. We refer to it as "`ElGamalRefresh()`" so as to distinguish it from any potential FTIL built-in command. Unlike the built-in command, it, too, works using a key manager `object`.

In our examples, we will always use "`ElGamalRefresh()`" rather than "`Refresh()`".

```
def ElGamalRefresh(keymgr, cipher):
  """
    Usage:
      ElGamalRefresh is an FTIL-native as-a-method re-implementation of
        "Refresh", refreshing (i.e., adding a newly-encrypted zero) to lists or
        memblocks of encrypted ElGamal ciphertexts.
    Inputs:
      None.
```

```
    Modified:
      keymgr -- The key manager object whose stockpile of zeroes is being used
                for the refreshing.
      cipher -- The cipher value (a list or memblock) that is being refreshed.
    Outputs:
      None.
  """

  confirm_label(keymgr, "FinTracer Key{Signature}")
  assert(cipher.size() <= keymgr.zeroes.size())

  modifying(keymgr):
    cipher[i] += keymgr.zeroes[i] over i = cipher.iter
    del keymgr.zeroes[:cipher.size()]
```

The fact that the updates of `cipher` and of `keymgr.zeroes` are both done inside the same "`modifying()`" block ensures that these are treated as a single transaction: zeroes cannot be removed without cipher values having been updated (which would have wasted computation resources) and zeroes cannot remain in the stockpile after having been used in ciphertext refreshing (which would have introduced a security hole).

For this reason, we can avoid complicated additional verification commands, such as

```
assert(cipher.metadata.type == list<ElGamalCipher>
         or cipher.metadata.type == memblock<ElGamalCipher>)
```

knowing that the action of the block is intrinsically tolerant to faults. This allows the FTIL code itself not to have to worry about such things, so it, in turn, can be implemented (as was done here) to completely avoid data replication.

Note 4.1.2. If the `array` type is fully supported, the above code should also work if `cipher` is an `array` of `ElGamalCipher`. Checking that this is the case via an `assert` would have required the more complicated conditional

```
(cipher.metadata.type.basename == "array"
                    and cipher.metadata.type.element == ElGamalCipher)
```

This `assert` condition would have required introspection to be supported as well. (See the discussion of the pseudotype `type` in Section 3.5.5.)

4.1.3 The key manager object

We now turn to the creation of a manager `object` per key.

```
def NewFTKeyMgr(keymgr):
  """
    Usage:
      Sets up a new key manager object.
    Inputs:
      None.
    Modified:
      None.
    Outputs:
      keymgr -- The new key manager object generated, to be labelled
                  "FinTracer Key".
  """
```

```
keymgr = object()
NewFinTracerKey(keymgr.shared_pub_key, keymgr.priv_key, keymgr.one,
                keymgr.zero, keymgr.zeroes)
keymgr.AddZeroes = ::AddZeroes
keymgr.ElGamalRefresh = ::ElGamalRefresh
label(keymgr, "FinTracer Key{Signature}")
```

With this script, a new managed key can be generated simply by

```
NewFTKeyMgr(keymgr)
```

4.1.4 Notes on the use of "modifying()"

With the "`modifying()`" directive being one of the newest additions to FTIL, it's worth pausing here for a moment to examine some nuances of its use, particularly as they are reflected in the programming of the key manager `object`.

Necessity

The use of labels makes `object`s immutable, but immutability is not the essence of labelling. The essence of labelling is to declare the `object` as fit for a particular purpose. The problem is that the underlying realities may change, so what is fit for a particular purpose now may not be fit for the same purpose later on. The "`modifying()`" directive allows us to change labelled `object`s in a way that keeps them fit for their original purpose.

Remark for the advanced reader 4.1.1. Similar can be said about shared elementary entities. The essence of a shared entity is that all nodes who hold one can be sure the value they hold is the same as what is seen by all other nodes. They are immutable because that is the easiest way to uphold this invariant, not because immutability is part of the essence of being shared. Future FTIL features may allow changing of the values of shared objects, as long as these changes are done in lockstep at all nodes.

Importantly, one cannot avoid a directive like "`modifying()`" in the design of FTIL. Suppose, for example, that `obj` is a labelled `object`, and that we want to modify its value without something like "`modifying()`".

On the face of it, one may think that we can create a new `object`, `temp`, that will receive the target value of `obj`, and will finally assign this value into `obj`, labelling the result.

There are several problems with this, including the race conditions involved (we do not want to end up with either two or zero labelled `object`s if there is a fault) and the inability for `temp` to retain the original command scope within which `obj` was originally label-confirmed, or even its original labelling strength. These problems extend also to questions about the safety design of the language: if `object`s can be switched under the hood so easily (including, in this case, by their own methods), users will have to verify every label of every `object` before and after each command.

However, the problem that is the most obvious signal that no such workaround is really feasible is the problem of modifying an `object` on nodes that only form part of its definition scope. In the present example, an FTIL user may wish to run `AddZeroes` on only a single node. The remaining nodes do not participate in the calculation, and essentially are not even involved. If we force labelled `object` modifications to be done by swapping the underlying `object`, all such operations would have had to be done on the `object`'s entire definition scope, with `temp` duplicating `obj` and then replacing it in every node, even though there is no difference between the two in most nodes.

Local vs global change

Consider, again, a situation where "`modifying()`" is used in a scope that is smaller than the scope where the modified `object` is defined.

Labelling, as discussed, reflects that the `object` is fit for a particular purpose. Importantly, this is a global property. If one node changes the value of the `object`, this may disrupt the `object`'s fit for purpose for the other nodes, those that made no modification and may perhaps not even be aware of a modification.

How, therefore, does "`modifying()`" preserve the global notion of fitness, while allowing local modifications to take place?

The answer is that the slightly-complicated rules around certification of scripts employing the privileged "`modifying()`" directive ensure this.

Any node that certified a script using a "`modifying()`" declares by this that the modification done by the script is safe to be done locally, and that it does not require to be notified if another node wishes to use it on an `object` that is in scope for the certifying node.

By contrast, if said certifying node wants to convey that this particular use of "`modifying()`" is only considered safe by the node if performed globally, across the `object`'s entire definition scope, then this can be communicated by not certifying the script itself, but executing it only from a privileged context. If the idea is that such an action can be performed from any context whatsoever, the FTIL idiom that describes this is a wrapper script around the script performing the "`modifying()`". Here, the wrapper script can be certified without the script actually performing the "`modifying()`" block needing its own certification.

4.2 Modularisation

4.2.1 Key management in a module

To demonstrate how scripts can be packaged into modules, and how these module scripts can then be certified, let us look at the example of the key manager.

The scripts up until this point can all be packaged into a single module file.

To do this, let us add code that will create the module proxy object.

First, consider what interface we want this module proxy to provide: it needs to allow invocation of `NewFTKeyMgr` in order to create a key manager `object`, after which the FTIL user will simply use this `object` for all remaining actions. No other interfaces are required.

For this, we now package `NewFTKeyMgr`, via a wrapper script, as a method. The idea is for this to eventually become a method for the module proxy object.

Here is what such packaging looks like.

```
def NewFTKeyMgrMethod(ftkeys, keymgr):
  """
    Usage:
      Wrapper around NewFTKeyMgr for its use as a method.
    Inputs:
      ftkeys -- The module proxy. (Unused.)
    Modified:
      None.
    Outputs:
      keymgr -- The new key manager object generated, to be labelled
                "FinTracer Key".
  """

  NewFTKeyMgr(keymgr)
```

Now that this interface is in place, we can write the `__init` script, which will generate the `ftkeys` module proxy.

```
def __init(ftkeys):
  """
```

```
    Usage:
      __init script for the key management module.
    Inputs:
      None.
    Modified:
      None.
    Outputs:
      ftkeys -- The new module proxy object generated, to be labelled
                  "FinTracer Keys Module".
  """

  ftkeys = object()
  ftkeys.NewMgr = ::NewFTKeyMgrMethod
  label(ftkeys, "FinTracer Keys Module{Signature}")
```

The key line in the above script is the line creating `ftkeys.NewMgr`. Though it looks like an assignment, it is in fact a line that binds the script "`NewFTKeyMgrMethod`" as a method called "`NewMgr`" of the object "`ftkeys`".

To complete the full module, we now just need to create a text file that contains all the scripts provided thus far in this section, and to prepend this file by the initial lines

```
FTIL1.0
module ftkeys
```

Suppose that this file was stored in the URL "`file://ftkeys.ftil`".

Creating a new key manager would then have been doable by the command sequence

```
import "file://ftkeys.ftil"
ftkeys.NewMgr(keymgr)
```

Once the module proxy object `ftkeys` has been imported, additional key managers can be created simply by

```
ftkeys.NewMgr(keymgr2)
```

Note that the module specifies exactly what interfaces are being exposed to the user. The module first exposes the method

```
ftkeys.NewMgr()
```

and this, in turn, creates a new `object` that exposes the methods

```
keymgr.AddZeroes()
```

and

```
keymgr.ElGamalRefresh()
```

All other scripts in the module are not directly accessible to the user.

A user may further view but not modify each of `keymgr.shared_pub_key`, `keymgr.one`, `keymgr.zero` and `keymgr.zeroes` (but not `keymgr.priv_key`), but to an FTIL user (necessarily sitting at the coordinator node), the values exposed will only be those local to the coordinator node. For example, such a user will be able to view their own `zeroes`, but not the zeroes of any other node.

Note 4.2.1. If any script is aborted in mid-execution via a "fail", any variable that exists in the system at the time of the abort will continue to exist as before, but be accessible through new variable-dump names. Because these variables will exist outside any certified context, the user will not be able to continue using them for potentially-unsafe operations, but their values will be available for, e.g., debugging purposes.

Such variable dumps do not change the visibility of a variable: it will remain the case that only the values local to the coordinator node are visible to the FTIL user.

4.2.2 Script certification

A question that may arise at this point is which scripts need certifying and which not.

The answer is not straightforward. It is a question of what operations we wish to allow and which not.

If, for example, all our handling of key management issues will forever be done entirely from other module scripts, external to the key management module, which are, themselves certified, and therefore execute in privileged mode, then largely all of the functionality of the key management module will be available to these external scripts.

The main exception is that it would not be possible in this way to run `AddZeroes` and `ElGamalRefresh` on only a subset of the nodes over which the key manager `object` is defined. This is because both scripts utilise a `modifying` clause.

In the case of `AddZeroes`, it would still be possible to add a different number of zeroes, including none, to the stockpile of each node. This can be done by sending a distributed `int` variable as the parameter for `num` instead of the shared literal constant it has as a default. The script's functionality would therefore be the same. However, from a privacy perspective the two implementations are nevertheless different: even if one uses a distributed `num` parameter, without certifying `AddZeroes` individually each node must be informed every time *any* node wants to add any zeroes to its stockpile.

In the case of `ElGamalRefresh`, the difference is also in functionality: without certifying `ElGamalRefresh` individually, it would not be possible to utilise this script to refresh a variable `cipher` whose scope is smaller than the scope of the key manager `object`.

Because both individual management of the zeroes stockpile and individual refreshing of ciphertexts are common and expected uses of this module, we would probably want to certify both these scripts.

Whether or not we choose to certify other scripts in this module depends on the question of what functionality we want this module to provide from a *non-privileged* context. For example: what do we want to be able to run from the command-line?

Downstream users of any key manager `object` which operate in privileged mode will need the key manager itself to have a strong label for it to be trustworthy to them and for them to be able to invoke any of its methods. If we wish the command-line user to be able to create a new key manager `object` with a strong label themselves, by running `ftkeys.NewMgr` directly, this would require certifying either `NewFTKeyMgr` or `NewFTKeyMgr`. It doesn't matter which of the two is certified, as long as one is. Also, because this is the only way to invoke `NewFinTracerKey`, it is never necessary to certify `NewFinTracerKey` directly.

Because it is unclear what added context would make the creation of the key manager any safer than simply invoking `ftkeys.NewMgr`, certifying either `NewFTKeyMgr` or `NewFTKeyMgr` seems like a reasonable choice.

There is, however, one other script in this module whose certification makes a difference to the functionality exported by the module to other scripts that execute in privileged mode.

This is namely the `__init` script. The `__init` script cannot be invoked directly by other scripts. It can only run as part of an `import` instruction, which, in turn, can only occur on the command line.

This means that whether or not `__init` is a certified script determines whether or not the module proxy `ftkeys` itself has a strong label.

This is important because in some contexts we may want to give other scripts, working in privileged mode, the ability to generate their own key management `objects`, which, in turn, should also have strong labels. The way to do this is to send `ftkeys` as a parameter to such scripts, and to have these scripts execute `ftkeys.NewMgr` themselves. However, such method invocation will only work if `ftkeys` has a strong label, so requires `__init` to be certified.

Note 4.2.2. If the outside script actually wishes to invoke any of the key manager `object`'s methods, FTIL enforces the use of `confirm_label`, technically forcing the FTIL programmer to make sure that the `object`'s label is strong for such invocation to succeed. If a script only examines

the key manager's member attributes it does not strictly need to confirm the `object`'s label, but if it does not then it would not be advisable for anyone to certify either this script or any script that calls it, because the `object` whose attributes are being examined isn't, itself, trustworthy.

In short, by the choice of what gets certified and what not, FTIL allows us to convey a range of privacy attitudes towards the code. By choosing to certify "`AddZeroes()`" and "`ElGamalRefresh()`" we allow the zeroes stockpiles to be managed independently. By certifying "`__init()`" we allow outside scripts to create strongly-labelled key managers. And by certifying either "`NewFTKeyMgr()`" or "`NewFTKeyMgrMethod()`", we allow strongly-labelled key managers to be created on the command-line, or otherwise from a non-privileged context.

All variations are possible.

For example, if one was to certify "`AddZeroes()`", "`ElGamalRefresh()`" and "`__init()`", but neither "`NewFTKeyMgr()`" nor "`NewFTKeyMgrMethod()`", the module would have allowed creation and use of strongly-labelled key manager `objects`, but only from module scripts running in privileged mode, i.e. only from inside larger execution contexts that have been independently identified as safe. Any creation of key managers from other contexts, such as from the command line, would have failed during the set-up phase of `NewFinTracerKey` (because of its use of "`broadcast`"), and even if `NewFinTracerKey` itself had been individually certified to prevent this, the result would have been weakly-labelled key managers, not usable in practical protocols.

Note 4.2.3. It may be tempting to believe that no `object` such as "`ftkeys`" is necessary at all. The putative alternative solution would have been to simply rename the script "`NewFTKeyMgr()`" to "`__init()`". On the face of it, a key manager can in this way be generated directly by an `import` statement.

The problem with this approach is that any `object` created by an `import` cannot be used in a "`modifying()`" clause. It would not have been possible, even inside a "`modifying()`" clause, to modify the `zeroes` stockpile, e.g. in order to add to it more zeroes or to remove parts of it that have already been used to refresh a cipher value.

Additionally, it is only allowed to run `import` from the command line, so it would not have been possible, in such an implementation, to allow scripts in other modules to generate their own key managers.

The separation between "`NewFTKeyMgr()`" and "`NewFTKeyMgrMethod()`" has been introduced here simply for didactic reasons. In general, as we see, most functionality will need to be exposed to the user through methods, so even if a script does not need any method-specific functionality, we will henceforth largely add to them an initial "`self`" parameter just to enable them to be attached to `objects`, rather than have two versions, one without a "`self`" and one acting as a wrapper, adding the "`self`" parameter.

4.2.3 Notes on naming conventions and multi-tenancy

FTIL makes no statement regarding how to name your variables. This is a matter of taste. Importantly, however, some naming conventions are more conducive to support particular work types than others.

In Section 11.1, we consider a situation where multiple teams, each running its own investigation, need to work over overlapping time periods within the same FinTracer system. Questions that may arise are:

1. How can I protect my variables from being accidentally overwritten by another team?
2. How can I learn about new work that's been done by my own team?
3. How can I avoid duplicating work already done by others?

For all these questions and many others, naming can be an important component of the answer.

In Section 11.1, we advocated that each team will have its own top-level `object`-type variable that will identify the investigation, and all artefacts of an investigation should be attributes of this variable. When this is the case,

```
dir()
```

will return the list of investigations currently in progress, whereas

```
dir(investigation_a)
```

will list the artefacts that have been generated for Investigation A.

In the previous examples, we demonstrated module loading using the command

```
import "file://ftkeys.ftil"
```

In practice, we recommend that each investigation will create a sub-object called "`modules`" to its top-level investigation object, and will place module proxy objects under it. This can then be done like so:

```
import "file://ftkeys.ftil" to investigation_a.modules
```

The placing of the `object` says that it is a module used by Investigation A, and its name indicates that it was generated by importing `"ftkeys.ftil"`.

The user can now determine which modules have already been imported for their investigation by a simple

```
dir(investigation_a.modules)
```

and the names themselves provide the answers the user is looking for.

If "`dir()`" returns labels instead of just variable names, the user would also be able to verify regarding each module proxy object that it references the module it is supposed to reference, which provides protection against any potentially misleading name.

If "`dir()`" does not return the label, the user can check it using

```
display(investigation_a.modules.ftkeys.metadata.label)
```

or by running

```
confirm_label(investigation_a.modules.ftkeys, "FinTracer Keys Module{Signature}")
```

If the convenience features for `confirm_label` discussed in Section 3.10 are implemented, this can even be shortened to

```
confirm_label(investigation_a.modules.ftkeys, "FinTracer Keys Module")
```

Once an investigation wraps up, all investigation objects can be removed from the system in their entirety using

```
del investigation_a
```

and this will unload any modules that are no longer in use. Modules that are in use by a different investigation will remain.

4.3 Account sets

We now move to the case-dependent portions of the algorithm, i.e. those that cannot be handled entirely offline. This begins with specifying a set of accounts that are of interest, which are the accounts whose RE-crossing properties FinTracer is to ascertain. For example, these may be all accounts opened in the last month.

There are two distinct scenarios in which an account set can be specified. The more straightforward way to define a set of accounts is for AUSTRAC to specify these directly, by account number. However, more commonly AUSTRAC will not know specific account numbers. It will want to communicate to the REs a description of the target accounts, and each RE, individually, will determine which accounts match the description.

These two cases differ in terms of their privacy requirements. If AUSTRAC provides the REs with account lists, these lists are entirely known to both AUSTRAC and to the REs. There is therefore no secret to be kept private between the two (only between the REs and themselves). This is in contrast to a list of accounts set by the REs from a description sent by AUSTRAC. In this latter case, AUSTRAC does not know the identity of the matching accounts, nor even their total number.

This is therefore a "secret" held from AUSTRAC. We define our account-set managing `objects` in a way that allows them to keep track of which secrets were involved in their definitions. We refer to these as the account sets' `sources`. Each `source`, in turn, keeps track of how much information the FTIL user's actions have so far potentially revealed about each secret. This accounting involves the tracking of two numbers: `epsilon` and `delta`. These will ultimately be used in Section 4.9, where we will deal with how AUSTRAC actually retrieves any information. In principle, this source tracking implements differential privacy guarantees, the mathematics of which is explained in Section 5.4.

4.3.1 The account set object

We begin by defining a couple of helper scripts that construct an account-set managing `object` from a raw distributed `set<AccountID>` variable. The script will accept as an explicit input parameter whether or not the new `object` created is managing secret information. This is functionality we only want to be used from appropriate contexts, for which reason we will mark the script with the directive `NOCERT`. Note, however, that even had the script been certified there would not be any way to activate it from an uncontrolled context: it is an internal module script that does not have any direct interface to the outside world.

The idea of account sets is that they manage an underlying "`accounts`" field, which is a distributed `set<AccountID>`, defined only on the peer nodes, and containing at each RE node only those accounts that are managed by said RE.

```
def NewSource(ftcore, source):
  """
    Usage:
      NewSource is a convenience script that generates a new object signifying
        new information that is managed by the peer nodes and is secret from
        AUSTRAC.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
    Modified:
      None.
    Outputs:
      source -- The new object crated, to be labelled "FinTracer Tag Source".
  """

  source = object()
  on(_peer_ids):
    source.epsilon = 0.0
    source.delta = 0.0
  label(source, "FinTracer Tag Source{Signature}")


def NewAccountSet(accounts, secret, account_set) NOCERT:
  """
    Usage:
      NewAccountSet creates a new account-set managing object from a raw
        distributed set<AccountID>, optionally also managing a new source.
```

```
  Inputs:
    accounts -- The set<AccountID> describing which accounts to place in
                the set.
    secret -- Is the account set a secret from AUSTRAC? If so, creating the
              object also creates a new information source that needs to be
              tracked. This should be a shared Boolean.
  Modified:
    None.
  Outputs:
    account_set -- The new object to be created, to be labelled
                   "FinTracer Account Set".
"""

assert(secret.metadata.type == bool)
assert(secret.metadata.shared == True)
assert(CoordinatorID in secret.metadata.scope)

_peer_ids = setminus(scope(), nodeset(CoordinatorID))

on(_peer_ids):
  assert(accounts.metadata.type == set<AccountID>)
  assert(accounts.metadata.scope == _peer_ids)

# In a script labelling an object, one would quite often want to first
# construct a temporary object, then copy it into the final object before
# labelling it. This is done here by first constructing "_account_set", and
# then copying it into "account_set". The reason for this is to protect
# against the quirks of FTIL's parameter passing algorithm. Specifically,
# against the sending of the same value as several different parameters.
# Consider, for example, what would have happened had a user called the
# script with parameters
# NewAccountSet(accounts, secret, accounts)
# Had the script started with a direct "account_set = object()".
# this line would have overwritten the "accounts" parameter, so when later
# assigning "accounts" as an attribute of "account_set", what would be
# assigned would not have been valid data.
# This idiom is repeated in most labelling scripts that follow.

_account_set = object()
on(_peer_ids) _account_set.accounts = accounts

_account_set.sources = catalogue()

if (secret):
  NewSource(ftcore, _account_set_source)  # "ftcore" is unused.
                                          # It will remain uninitialised.
  _account_set.sources[_account_set_source.metadata.id] = _account_set_source

label(_account_set.sources, "FinTracer Tag Source Catalogue{Signature}")

account_set = _account_set

# In the next line we delete the temporary object before labelling the
# final return value. This is also part of the idiom, and is meant to
```

```
  # allow labelling to be done in-place.
  del _account_set

  # An alternative to this idiom would have been to label the temporary object
  # and then copy the labelled version to its destination variable.
  # The reason we do it this way is so that any script receiving "account_set"
  # as a return value from NewAccountSet will get it pre-label-confirmed, and
  # will not need to re-confirm it.
  label(account_set, "FinTracer Account Set{Signature}")
```

We will also provide for account sets a cloning script. Note that, as will be the case almost everywhere in our example code, payload data is never unnecessarily copied.

```
def CloneToNewAccountSet(ftcore, account_set, new_account_set):
  """
    Usage:
      CloneToNewAccountSet creates a new account-set managing object from an
        existing one, retaining all the information from the original.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      account_set -- The existing account set.
    Modified:
      None.
    Outputs:
      new_account_set -- The new object to be created, to be labelled
                         "FinTracer Account Set".
  """

  confirm_label(account_set, "FinTracer Account Set{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  _new_account_set = object()
  on(_peer_ids) _new_account_set.accounts = account_set.accounts

  _new_account_set.sources = catalogue()

  for _s in account_set.sources.keys():
    _new_account_set.sources[_s] = _account_set_source.sources[_s]

  label(_new_account_set.sources, "FinTracer Tag Source Catalogue{Signature}")

  new_account_set = _new_account_set
  del _new_account_set
  label(new_account_set, "FinTracer Account Set{Signature}")
```

As account sets are essentially wrappers for sets, it makes sense to endow them with some set functionality. For this, we first create a helper script much like the previous one, which creates a new account set from operands that already track their sources. The new set will have as its sources the union of the sources of all the operands it was computed from.

```
def NewAccountSetFromOperands(accounts, sources_a, sources_b,
                              account_set) NOCERT:
  """
    Usage:
```

```
      NewAccountSetFromOperands is a helper script that creates a new
        account-set managing object from a raw distributed set<AccountID>,
        but also initialises its own source catalogue as the union of two
        existing source catalogues.
    Inputs:
      accounts -- The set<AccountID> describing which accounts to place in
                  the set.
      sources_a, sources_b -- The two source catalogues to be merged.
    Modified:
      None.
    Outputs:
      account_set -- The new object to be created, to be labelled
                     "FinTracer Account Set".
  """

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):
    assert(accounts.metadata.type == set<AccountID>)
    assert(accounts.metadata.scope == _peer_ids)

  confirm_label(sources_a, "FinTracer Tag Source Catalogue{Signature}")
  confirm_label(sources_b, "FinTracer Tag Source Catalogue{Signature}")

  _account_set = object()
  on(_peer_ids) _account_set.accounts = accounts

  _account_set.sources = catalogue()

  for _s in sources_a.keys():
    _account_set.sources[_s] = _sources_a[_s]

  for _s in sources_b.keys():
    _account_set.sources[_s] = _sources_b[_s]

  label(_account_set.sources, "FinTracer Tag Source Catalogue{Signature}")

  account_set = _account_set
  del _account_set
  label(account_set, "FinTracer Account Set{Signature}")
```

With these preliminaries in place, the next three scripts implement set union, set intersection and set difference, respectively.

```
def AccountSetUnion(ftcore, set_a, set_b, result):
  """
    Usage:
      AccountSetUnion computes the union of two FinTracer account sets.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      set_a, set_b -- The operands.
    Modified:
      None.
    Outputs:
```

```
      result -- The new account_set to be created.
  """

  confirm_label(set_a, "FinTracer Account Set{Signature}")
  confirm_label(set_b, "FinTracer Account Set{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids) _accounts = union(set_a.accounts, set_b.accounts)

  NewAccountSetFromOperands(_accounts, set_a.sources, set_b.sources, result)


def AccountSetIntersection(ftcore, set_a, set_b, result):
  """
    Usage:
      AccountSetIntersection computes the intersection of two FinTracer
        account sets.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      set_a, set_b -- The operands.
    Modified:
      None.
    Outputs:
      result -- The new account_set to be created.
  """

  confirm_label(set_a, "FinTracer Account Set{Signature}")
  confirm_label(set_b, "FinTracer Account Set{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids) _accounts = intersection(set_a.accounts, set_b.accounts)

  NewAccountSetFromOperands(_accounts, set_a.sources, set_b.sources, result)


def AccountSetDifference(ftcore, set_a, set_b, result):
  """
    Usage:
      AccountSetDifference computes the difference of two FinTracer
        account sets.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      set_a, set_b -- The operands.
    Modified:
      None.
    Outputs:
      result -- The new account_set to be created.
  """

  confirm_label(set_a, "FinTracer Account Set{Signature}")
  confirm_label(set_b, "FinTracer Account Set{Signature}")
```

```
  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids) _accounts = setminus(set_a.accounts, set_b.accounts)

  NewAccountSetFromOperands(_accounts, set_a.sources, set_b.sources, result)
```

4.3.2 Account sets from lists

With our account set managing object infrastructure in place, we turn to the code that initialises these meaningfully.

We begin with the simpler case, where AUSTRAC has specific account numbers, and therefore no new "secrets" are created. In this case, we can use a script such as the one below.

```
def AccountsFromList(ftcore, austrac_list, accounts):
  """
    Usage:
      AccountsFromList generates an account set object based on an explicit
        list provided by AUSTRAC in the form of a set of "AccountAddr"s, which
        is a form where both the AccountID and the managing RE are explicitly
        listed. In the final account set object, each RE will see only those
        accounts belonging to them. The script also verifies that the managing
        RE stated in the AccountAddr entity matches what can be automatically
        inferred from knowledge of the BSB. The script halts if there are any
        discrepancies between the two.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      austrac_list -- The set of AccountAddrs explicitly provided by AUSTRAC.
    Modified:
      None.
    Outputs:
      accounts -- The distributed set<AccountID> generated from AUSTRAC's list.
  """

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(CoordinatorID):
    assert(austrac_list.metadata.scope == nodeset(CoordinatorID))
    assert(austrac_list.metadata.type == set<AccountAddr>)

    # First we verify that the correct REs are listed.
    _problem_list = austrac_list[austrac_list[@].node_id !=
                            BranchToBank(austrac_list[@].account_id.bsb_num)]
    assert(_problem_list.size() == 0)

  _account_transmitter = transmitter<set<AccountID>>()

  # We partition the list according to the owning RE...
  on(CoordinatorID) for _n in _peer_ids:
    _account_transmitter[_n] = austrac_list[austrac_list[@].node_id == _n]

  # ... and transmit to each RE only its portion of the list.
  transmit(_account_transmitter)

  # Ultimately, we pack all this in an account set object.
```

```
# The parameter "False" indicates that nothing here is secret from AUSTRAC.
on(_peer_ids) _raw_accounts = _account_transmitter[CoordinatorID]
NewAccountSet(_raw_accounts, False, accounts)
```

This script performs a number of different functions:

- First, it receives a set of desired accounts from the AUSTRAC user. The use of the "`set`" type here is important, in that it implicitly ascertains that the inputs are unique. Readers who feel that `lists` are better because they retain a specific order may consider the filtering operation that partitions the list, which scrambles its order in any case.
- Second, it ascertains that the account list given to it is visible only to AUSTRAC before attempting any further action. This is a stop-gap against any incorrect use of this script, that may potentially result in REs sharing some account information due to an inadvertent action by AUSTRAC.
- Third, it compares for each account the RE information provided to it by the AUSTRAC user with the RE information that can be derived automatically by use of the account's BSB information. Because these lists will later on be sent to REs, the program needs to be very sure that no RE receives information it shouldn't. As such, it trusts neither the user-provided information nor the automatic information to determine where each account number is sent. The two must be in complete agreement, before any account values are distributed.
- Finally, the account list is split and sent out, so that each RE receives exactly those accounts in the list that are managed by it.

4.3.3 Account sets from description

As mentioned, defining account sets by explicit lists is not the expected common scenario. More commonly, stipulating an account set will be done by AUSTRAC communicating a particular description and REs finding which accounts match said description. This is the case in which the new set is kept secret from AUSTRAC, and we will use the source tracking of our account set objects to ensure this.

Here, too, there are two possible scenarios to consider. Is the description one that should be interpreted by the human RE-side operators, or is the description meant for automatic parsing by the RE-side system?

Below is code to support the process of defining an account set by use of a human-readable description. (See Section 6.1 for more on how such a description is to be structured.)

```
def AccountsFromDescription(ftcore, description_string, accounts):
  """
    Usage:
      A way to generate an account set based on a human-readable
        description provided by AUSTRAC.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      description_string -- AUSTRAC's description of the desired set.
    Modified:
      None.
    Outputs:
      accounts -- The generated account set.
  """

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))
```

```
  # We use a set type to ensure account uniqueness.
  on(_peer_ids) _raw_accounts = Read<set<AccountID>>(description_string)

  # The parameter "True" indicates that the set is secret from AUSTRAC.
  NewAccountSet(_raw_accounts, True, accounts)
```

Note that in the previous example we defined accounts by means of a variable whose base type was "`AccountAddr`". This time, by contrast, we are reading a set of "`AccountID`". The difference is that the `AccountAddr` type also stores explicitly information about which RE the account belongs to. This is not something we need when the RE operators themselves are providing the input.

If during later processing one wishes to have the information of managing financial institution explicitly available, one can use the following script to convert a set of `AccountID` into a set of `AccountAddr`.

We mark the script below `NOCERT` because it manipulates raw account set data directly, without managing the data's sources. Presently, this script is not utilised by any of our other scripts because it is unneeded. In the future, however, an appropriate context might be found that does require it, at which point it can be used as soon as a script providing this context is certified.

```
def AccountAddrsFromAccountIDs(ftcore, accounts, account_addrs) NOCERT:
  """
    Usage:
      This script expands a container of AccountIDs into a similar container
        of AccountAddrs, by adding the information of the managing RE.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      accounts -- The set of AccountIDs.
    Modified:
      None.
    Outputs:
      account_addrs -- The set of AccountAddrs generated.
  """

  # assert(accounts.metadata.type == set<AccountID>)
  account_addrs = AccountAddr(BranchToBank(accounts[@].bsb_num), accounts[@])
```

In the code above, we commented out the "`assert`" statement. If this statement exists, the user can be sure that what is being manipulated is a set of `AccountID`s and what is being created is a set of `AccountAddr`s. By taking out the "`assert`", the command becomes more generic. It will work on any iterable whose value type is `AccountID`. This flexibility has the price of less rigorous type-checking.

4.3.4 Account sets from SQL query

The final scenario to be dealt with is one where AUSTRAC needs to communicate the target set of accounts in the form of a description that can be handled automatically by the RE-side system. Typically, this should be the best and most common scenario, because direct manual intervention will lead to human error and uncheckable processes.

To do this, AUSTRAC describes the desired account IDs as the output of a SQL query, to be run on the Auxiliary DB. The code for this is as follows.

```
def AccountsFromSQL(ftcore, sql_string, accounts):
  """
    Usage:
      A way to generate an account set based on a SQL description provided by
        AUSTRAC.
```

```
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      sql_string -- AUSTRAC's SQL description of the desired set.
    Modified:
      None.
    Outputs:
      accounts -- The generated account set.
  """

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids) _raw_accounts = AuxDBRead<AccountID>(sql_string)

  # The parameter "True" indicates that the set is secret from AUSTRAC.
  NewAccountSet(_raw_accounts, True, accounts)
```

4.4 Tags

4.4.1 The FinTracer "tag"

Before continuing, consider what the principles are behind the privacy guarantees of the FinTracer algorithm. FinTracer is built on the idea that there are *tags*, mapping accounts into ElGamal (or, more generally, semi-homomorphically encrypted) ciphertexts, and these ciphertexts are encrypted with a key that only AUSTRAC has. As long as all processing of the tags is done on the peer nodes, away from AUSTRAC, and only final query results are returned to AUSTRAC, the algorithm is safe.

We will, later on, define strict rules regarding what any peer can send to any other peer, to make sure no information leak is possible (for example, by observing the *number* of ciphertexts sent, rather than the ciphertext values themselves), but when it comes to local processing of tag values at the peer nodes, nothing that treats ciphertexts as semi-homomorphically encrypted black boxes can present a security problem.

Note 4.4.1. We will, in particular, never assume about tag values that they are refreshed. Tag values, while processed locally, can be open to repetition attacks without this compromising the safety of the algorithm. Any script communicating tag values will take this into account and refresh the tags before they are communicated.

When actually describing the basic FinTracer algorithm, in later sections, we will need essentially no local processing of tags. However, the point of FTIL is that it allows future-proofing for later, more complex variations of the algorithm that may be required to tackle investigation-driven, real-world queries later on. To do this, we now define the tag type in a way that supports essentially "everything" that falls under the umbrella of such local processing. In fact, we'll introduce significantly more scripts than are strictly necessary, so as to make sure that later implementations are not only possible but also efficient.

None of these scripts needs to be certified to support the basic FinTracer algorithm, but if they *are* certified, then later investigations can run queries that execute elaborate variations on FinTracer, without this requiring any further certifications from REs. These variations will all be privacy preserving, because all will use the same privacy principles that keep FinTracer safe to begin with.

The following is our definition of a FinTracer tag. The tag will have two member attributes: a "`map`" that maps `AccountID`s to ciphertexts and a "`keymgr`" key manager, which documents which key was used to encrypt the ciphertext. To reiterate: the core of FinTracer's security comes from the knowledge that this key is only known to AUSTRAC and cannot be communicated to any other party.

In addition to these, tags manage their own `sources`, much like account sets do. Indeed, the two `object` types share sources between them.

Note 4.4.2. In our list, below, of "everything" that FinTracer tags can support, conspicuously absent are any scripts that move information between `tag.map` keys. This is important, because it underpins our calculations regarding differential privacy.

Our FinTracer propagation script will be an exception to this rule, and will, indeed, require special handling. See Section 10.1 for a discussion.

```
def TagAdd(tag, other):
  """
    Usage:
      TagAdd sums into the tag the ciphertext values from 'other'.
    Inputs:
      other -- The FinTracer tag object to take values from.
    Modified:
      tag -- The FinTracer tag object to sum values to.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(other, "FinTracer Tag{Signature}")
  assert(tag.keymgr.shared_pub_key == other.keymgr.shared_pub_key)

  # Note that this implementation is resilient to the case that a tag is
  # added to itself.

  modifying(tag):
    tag.map[@] += other.map[@]
    modifying(tag.sources):
      for _s in other.sources.keys():
        # We could have added an assert that would test that the scope() equals
        # both tag.metadata.scope and other.metadata.scope, but if anyone
        # tries running this script on any other scope, the next line would
        # fail, anyway, rolling back everything (to the beginning of the
        # 'modifying(tag)' block.
        tag.sources[_s] = other.sources[_s]


def TagSubtract(tag, other):
  """
    Usage:
      TagSubtract subtracts from the tag the ciphertext values from 'other'.
    Inputs:
      other -- The FinTracer tag object to take values from.
    Modified:
      tag -- The FinTracer tag object to subtract values from.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(other, "FinTracer Tag{Signature}")
  assert(tag.keymgr.shared_pub_key == other.keymgr.shared_pub_key)
```

```
  # Note that this implementation is resilient to the case that a tag is
  # subtracted from itself.

  modifying(tag):
    tag.map[@] -= other.map[@]
    modifying(tag.sources):
      for _s in other.sources.keys():
        # We could have added an assert that would test that the scope() equals
        # both tag.metadata.scope and other.metadata.scope, but if anyone
        # tries running this script on any other scope, the next line would
        # fail, anyway, rolling back everything (to the beginning of the
        # 'modifying(tag)' block.
        tag.sources[_s] = other.sources[_s]


def TagInvert(tag):
  """
    Usage:
      TagInvert switches all tag values to their negatives.
    Inputs:
      None.
    Modified:
      tag -- The FinTracer tag object whose values get inverted.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")

  modifying(tag) tag.map[@] = -tag.map[@]


def TagSet(tag, other):
  """
    Usage:
      TagSet replaces in 'tag' the ciphertext values from 'other'.
        Values not associated with keys of 'other' are not modified.
    Inputs:
      other -- The FinTracer tag object to take values from.
    Modified:
      tag -- The FinTracer tag object to set values in.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(other, "FinTracer Tag{Signature}")
  assert(tag.keymgr.shared_pub_key == other.keymgr.shared_pub_key)

  # Note that this implementation is resilient to the case that a tag is
  # set to itself.

  modifying(tag):
```

```
    tag.map[@] = other.map[@]
    modifying(tag.sources):
      for _s in other.sources.keys():
        # We could have added an assert that would test that the scope() equals
        # both tag.metadata.scope and other.metadata.scope, but if anyone
        # tries running this script on any other scope, the next line would
        # fail, anyway, rolling back everything (to the beginning of the
        # 'modifying(tag)' block.
        tag.sources[_s] = other.sources[_s]


def TagMult(tag, factor):
  """
    Usage:
      TagMult multiplies the tag's values by 'factor'.
    Inputs:
      factor -- The multiplication factor. An integer or Ed25519Int.
    Modified:
      tag -- The FinTracer tag object to be modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")

  modifying(tag) tag.map[@] *= factor


def TagSetOne(tag, accounts):
  """
    Usage:
      TagSetOne sets tag values in 'accounts' to an encrypted '1' value.
    Inputs:
      accounts -- An account set. The tag keys to be modified in 'tag'.
    Modified:
      tag -- The FinTracer tag object to be modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(accounts, "FinTracer Account Set{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  modifying(tag):
    on(_peer_ids) tag.map[accounts.accounts[@]] = {tag.keymgr.one}
    modifying(tag.sources):
      for _s in accounts.sources.keys():
        # We could have added an assert that would test that the scope() equals
        # both tag.metadata.scope and accounts.metadata.scope, but if anyone
        # tries running this script on any other scope, the next line would
        # fail, anyway, rolling back everything (to the beginning of the
        # 'modifying(tag)' block.
```

```
        tag.sources[_s] = accounts.sources[_s]


def TagSetZero(tag, accounts):
  """
    Usage:
      TagSetZero sets tag values in 'accounts' to an encrypted '0' value.
    Inputs:
      accounts -- An account set. The tag keys to be modified in 'tag'.
    Modified:
      tag -- The FinTracer tag object to be modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(accounts, "FinTracer Account Set{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  modifying(tag):
    on(_peer_ids) tag.map[accounts.accounts[@]] = {tag.keymgr.zero}
    modifying(tag.sources):
      for _s in accounts.sources.keys():
        # We could have added an assert that would test that the scope() equals
        # both tag.metadata.scope and accounts.metadata.scope, but if anyone
        # tries running this script on any other scope, the next line would
        # fail, anyway, rolling back everything (to the beginning of the
        # 'modifying(tag)' block.
        tag.sources[_s] = accounts.sources[_s]


def TagSetConstant(tag, accounts, c):
  """
    Usage:
      TagSetConstant sets tag values in 'accounts' to an encrypted 'c' value.
    Inputs:
      accounts -- An account set. The tag keys to be modified in 'tag'.
      c -- The value to be set.
    Modified:
      tag -- The FinTracer tag object to be modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(accounts, "FinTracer Account Set{Signature}")

  # We now prepare an encrypted version of 'c'.
  _encrypted_c = tag.keymgr.one * c
  # In principle, the line above can be implemented more efficiently than
  # _encrypted_c = ElGamalEncrypt(c, tag.keymgr.shared_pub_key)
  # because there is no need for randomness. However, either works.
  # Also, depending on how FTIL is implemented, there may be a substantial
```

```
  # advantage to using 'c' rather than 'Ed25519Int(c)', because if 'c' is an
  # int, the multiplication can be handled more efficiently (because the range
  # of an int is much smaller than the range of an Ed25519Int.
  # On the other end of the scale, FTIL may not support multiplication by an
  # int at all, in which case 'c' will have to be replaced by 'Ed25519Int(c)',
  # because FTIL does not support implicit conversion.

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  modifying(tag):
    on(_peer_ids) tag.map[accounts.accounts[@]] = {_encrypted_c}
    modifying(tag.sources):
      for _s in accounts.sources.keys():
        # We could have added an assert that would test that the scope() equals
        # both tag.metadata.scope and accounts.metadata.scope, but if anyone
        # tries running this script on any other scope, the next line would
        # fail, anyway, rolling back everything (to the beginning of the
        # 'modifying(tag)' block.
        tag.sources[_s] = accounts.sources[_s]


def TagAddOne(tag, accounts):
  """
    Usage:
      TagAddOne adds to tag values in 'accounts' an encrypted '1' value.
    Inputs:
      accounts -- An account set. The tag keys to be modified in 'tag'.
    Modified:
      tag -- The FinTracer tag object to be modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(accounts, "FinTracer Account Set{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  modifying(tag):
    on(_peer_ids) tag.map[accounts.accounts[@]] += {tag.keymgr.one}
    modifying(tag.sources):
      for _s in accounts.sources.keys():
        # We could have added an assert that would test that the scope() equals
        # both tag.metadata.scope and accounts.metadata.scope, but if anyone
        # tries running this script on any other scope, the next line would
        # fail, anyway, rolling back everything (to the beginning of the
        # 'modifying(tag)' block.
        tag.sources[_s] = accounts.sources[_s]


def TagAddZero(tag, accounts):
  """
    Usage:
      TagAddZero adds to tag values in 'accounts' an encrypted '0' value.
```

```
        The effect on the plaintext is that additional map keys may be added.
        (Note that no additional randomisation is added, so this does not
        act as a "Refresh".)
    Inputs:
      accounts -- An account set. The tag keys to be modified in 'tag'.
    Modified:
      tag -- The FinTracer tag object to be modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(accounts, "FinTracer Account Set{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  modifying(tag):
    on(_peer_ids) tag.map[accounts.accounts[@]] += {tag.keymgr.zero}
    modifying(tag.sources):
      for _s in accounts.sources.keys():
        # We could have added an assert that would test that the scope() equals
        # both tag.metadata.scope and accounts.metadata.scope, but if anyone
        # tries running this script on any other scope, the next line would
        # fail, anyway, rolling back everything (to the beginning of the
        # 'modifying(tag)' block.
        tag.sources[_s] = accounts.sources[_s]


def TagAddConstant(tag, accounts, c):
  """
    Usage:
      TagAddConstant adds to tag values in 'accounts' an encrypted 'c' value.
    Inputs:
      accounts -- An account set. The tag keys to be modified in 'tag'.
      c -- The value to be added.
    Modified:
      tag -- The FinTracer tag object to be modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(accounts, "FinTracer Account Set{Signature}")

  # We now prepare an encrypted version of 'c'.
  _encrypted_c = tag.keymgr.one * c

  # In principle, the line above can be implemented more efficiently than
  # _encrypted_c = ElGamalEncrypt(c, tag.keymgr.shared_pub_key)
  # because there is no need for randomness. However, either works.
  # Also, depending on how FTIL is implemented, there may be a substantial
  # advantage to using 'c' rather than 'Ed25519Int(c)', because if 'c' is an
  # int, the multiplication can be handled more efficiently (because the range
  # of an int is much smaller than the range of an Ed25519Int.
```

```
  # On the other end of the scale, FTIL may not support multiplication by an
  # int at all, in which case 'c' will have to be replaced by 'Ed25519Int(c)',
  # because FTIL does not support implicit conversion.

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  modifying(tag):
    on(_peer_ids) tag.map[accounts.accounts[@]] += {_encrypted_c}
    modifying(tag.sources):
      for _s in accounts.sources.keys():
        # We could have added an assert that would test that the scope() equals
        # both tag.metadata.scope and accounts.metadata.scope, but if anyone
        # tries running this script on any other scope, the next line would
        # fail, anyway, rolling back everything (to the beginning of the
        # 'modifying(tag)' block.
        tag.sources[_s] = accounts.sources[_s]


def TagFilterOut(tag, accounts):
  """
    Usage:
      TagFilterOut removes from the domain of tag.map the set "accounts".
    Inputs:
      accounts -- An account set. The set of accounts to remove from the
                  domain.
    Modified:
      tag -- The FinTracer tag object to be modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(accounts, "FinTracer Account Set{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  modifying(tag):
    on(_peer_ids) del tag.map[accounts.accounts]
    modifying(tag.sources):
      for _s in accounts.sources.keys():
        # We could have added an assert that would test that the scope() equals
        # both tag.metadata.scope and accounts.metadata.scope, but if anyone
        # tries running this script on any other scope, the next line would
        # fail, anyway, rolling back everything (to the beginning of the
        # 'modifying(tag)' block.
        tag.sources[_s] = accounts.sources[_s]


def TagReduce(tag, accounts):
  """
    Usage:
      TagReduce changes the domain of tag.map to be exactly the set accounts.
        It can be used both to filter tag values and to add zeroes.
    Inputs:
```

```
      accounts -- An account set. The set of accounts to use as a new domain.
    Modified:
      tag -- The FinTracer tag object to be modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(accounts, "FinTracer Account Set{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  modifying(tag):
    # We use sub-assignment because the scope of tag.map is different to the
    # scope of accounts.accounts.
    on(_peer_ids) tag.map[] = dict{tag.map.lookup(i, tag.keymgr.zero)
                    over i = accounts.accounts.iter}
    modifying(tag.sources):
      for _s in accounts.sources.keys():
        # We could have added an assert that would test that the scope() equals
        # both tag.metadata.scope and accounts.metadata.scope, but if anyone
        # tries running this script on any other scope, the next line would
        # fail, anyway, rolling back everything (to the beginning of the
        # 'modifying(tag)' block.
        tag.sources[_s] = accounts.sources[_s]


def TagEmpty(tag):
  """
    Usage:
      TagEmpty empties the tag's value map.
    Inputs:
      None.
    Modified:
      tag -- The FinTracer tag object to be modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  modifying(tag):
    tag.map.empty()
    # The following 'if' statement allows one to run this script both when
    # the scope of the script equals the scope of 'tag' and when not. The
    # code relies on the Compute Manager's ability to compute 'scope()'.
    if scope() == tag.metadata.scope:
      modifying(tag.sources):
        tag.sources.empty()


def TagSanitise(tag):
```

```
  """
    Usage:
      TagSanitise sanitises the tag's value map (multiplies by random nonzero).
    Inputs:
      None.
    Modified:
      tag -- The FinTracer tag object to be modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")

  # If Sanitise() on dictionaries is available as a built-in function:
  modifying(tag) Sanitise(tag.map)

  # Otherwise:
  # modifying(tag) tag.map[@] = Sanitise(tag.map[@])

  # Note that in the first variation it is in-place, in the second not.


def TagClone(tag, other):
  """
    Usage:
      TagClone sets the tag's map to be the same as "other"'s, but without any
        data copying. The command does nothing if "tag" and "other" are the
        same object.
    Inputs:
      other -- The tag object whose data is cloned.
    Modified:
      tag -- The FinTracer tag object to be modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(other, "FinTracer Tag{Signature}")
  assert(tag.keymgr == other.keymgr)

  # The following 'if' statement is needed in order to take care of the
  # potential situation in which a tag is cloned to itself.
  if tag != other:
    modifying(tag):
      # We didn't
      # assert(scope() == tag.metadata.scope
      #           and scope() == other.metadata.scope)
      # but if anyone attempts to run TagClone at a scope other than the scope
      # of 'tag' and the scope of 'other', the next line will fail.
      tag.map = other.map
      modifying(tag.sources):
        # By design, if tag != other then also tag.sources != other.sources,
        # so there is no danger in emptying tag.sources.
        tag.sources.empty()
```

```
        for _i in other.sources.keys():
          tag.sources[_i] = other.sources[_i]


def TagUsedBudget(tag, epsilon, delta):
  """
    Usage:
      TagUsedBudget returns the differential privacy budget already used up.
    Inputs:
      tag -- The tag being queried.
    Modified:
      None.
    Outputs:
      epsilon, delta -- The returned values.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):
    _temp_epsilon = 0.0
    _temp_delta = 0.0
    for _i in tag.sources.keys():
      _temp_epsilon = max(_temp_epsilon, tag.sources[_i].epsilon)
      _temp_delta = max(_temp_delta, tag.sources[_i].delta)

    epsilon = _temp_epsilon
    delta = _temp_delta


def TagExpendBudget(tag, epsilon, delta):
  """
    Usage:
      TagExpendBudget Registers new differential privacy budget used up
        by the sources.
    Inputs:
      epsilon, delta -- The newly used amount of differential privacy budget.
    Modified:
      tag -- The tag being modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):
    assert(epsilon.metadata.type == float)
    assert(epsilon > 0.0)
    assert(delta.metadata.type == float)
    assert(delta > 0.0)
```

```
  modifying(tag.sources):  # Used here to ensure an all-or-nothing update.
    for _i in tag.sources.keys():
      modifying(tag.sources[_i]):
        on(_peer_ids):
          tag.sources[_i].epsilon += epsilon
          tag.sources[_i].delta += delta


def TagExpendBudgetWrapper(tag, epsilon, delta):
  """
    TagExpendBudgetWrapper is a wrapper for TagExpendBudget, here for
      certification reasons.
  """

  TagExpendBudget(tag, epsilon, delta)


def TagUsedBudgetWrapper(tag, epsilon, delta):
  """
    TagUsedBudgetWrapper is a wrapper for TagUsedBudget, here for
      certification reasons.
  """

  TagUsedBudget(tag, epsilon, delta)


def TagCloneWrapper(tag, other):
  """
    TagCloneWrapper is a wrapper for TagClone, here for certification
      reasons.
  """

  TagClone(tag, other)


def TagSanitiseWrapper(tag):
  """
    TagSanitiseWrapper is a wrapper for TagSanitise, here for certification
      reasons.
  """

  TagSanitise(tag)


def TagEmptyWrapper(tag):
  """
    TagEmptyWrapper is a wrapper for TagEmpty, here for certification reasons.
  """

  TagEmpty(tag)


def TagFilterOutWrapper(tag, accounts):
  """
```

```
    TagFilterOutWrapper is a wrapper for TagFilterOut, here for certification
      reasons.
  """

  TagFilterOut(tag, accounts)


def TagReduceWrapper(tag, accounts):
  """
    TagReduceWrapper is a wrapper for TagReduce, here for certification
      reasons.
  """

  TagReduce(tag, accounts)


def TagAddConstantWrapper(tag, accounts, c):
  """
    TagAddConstantWrapper is a wrapper for TagAddConstant, here for
      certification reasons.
  """

  TagAddConstant(tag, accounts, c)


def TagAddZeroWrapper(tag, accounts):
  """
    TagAddZeroWrapper is a wrapper for TagAddZero, here for certification
      reasons.
  """

  TagAddZero(tag, accounts)


def TagAddOneWrapper(tag, accounts):
  """
    TagAddOneWrapper is a wrapper for TagAddOne, here for certification
      reasons.
  """

  TagAddOne(tag, accounts)


def TagSetConstantWrapper(tag, accounts, c):
  """
    TagSetConstantWrapper is a wrapper for TagSetConstant, here for
      certification reasons.
  """

  TagSetConstant(tag, accounts, c)


def TagSetZeroWrapper(tag, accounts):
  """
```

```
    TagSetZeroWrapper is a wrapper for TagSetZero, here for certification
      reasons.
  """

  TagSetZero(tag, accounts)


def TagSetOneWrapper(tag, accounts):
  """
    TagSetOneWrapper is a wrapper for TagSetOne, here for certification
      reasons.
  """

  TagSetOne(tag, accounts)


def TagMultWrapper(tag, factor):
  """
    TagMultWrapper is a wrapper for TagMult, here for certification reasons.
  """

  TagMult(tag, factor)


def TagSetWrapper(tag, other):
  """
    TagSetWrapper is a wrapper for TagSet, here for certification reasons.
  """

  TagSet(tag, other)


def TagInvertWrapper(tag):
  """
    TagInvertWrapper is a wrapper for TagInvert, here for certification
    reasons.
  """

  TagInvert(tag)


def TagSubtractWrapper(tag, other):
  """
    TagSubtractWrapper is a wrapper for TagSubtract, here for certification
    reasons.
  """

  TagSubtract(tag, other)


def TagAddWrapper(tag, other):
  """
    TagAddWrapper is a wrapper for TagAdd, here for certification reasons.
  """
```

```
  TagAdd(tag, other)


def NewTag(ftcore, keymgr, tag):
  """
    Usage:
      NewTag creates a new empty tag.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      keymgr -- The key manager to be used for the new tag's values.
    Modified:
      None.
    Outputs:
      tag -- The new tag to be created, to be labelled "FinTracer Tag".
  """

  confirm_label(keymgr, "FinTracer Key{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  _tag = object()
  _tag.map = dict<AccountID, ElGamalCipher>()
  _tag.keymgr = keymgr

  _tag.sources = catalogue()
  label(_tag.sources, "FinTracer Tag Source Catalogue{Signature}")

  _tag.ExpendBudget = ::TagExpendBudgetWrapper
  _tag.UsedBudget = ::TagUsedBudgetWrapper

  _tag.Sanitise = ::TagSanitiseWrapper
  _tag.Empty = ::TagEmptyWrapper
  _tag.Reduce = ::TagReduceWrapper
  _tag.FilterOut = ::TagFilterOutWrapper
  _tag.AddConstant = ::TagAddConstantWrapper
  _tag.AddZero = ::TagAddZeroWrapper
  _tag.AddOne = ::TagAddOneWrapper
  _tag.SetConstant = ::TagSetConstantWrapper
  _tag.Clone = ::TagCloneWrapper
  _tag.SetZero = ::TagSetZeroWrapper
  _tag.SetOne = ::TagSetOneWrapper
  _tag.Mult = ::TagMultWrapper
  _tag.Set = ::TagSetWrapper
  _tag.Invert = ::TagInvertWrapper
  _tag.Subtract = ::TagSubtractWrapper
  _tag.Add = ::TagAddWrapper

  tag = _tag
  del _tag
  label(tag, "FinTracer Tag{Signature}")


def CloneToNewTag(ftcore, tag, new_tag):
```

```
"""
  Usage:
    CloneToNewTag creates a new tag, "new_tag", that clones the data of "tag".
      The object new_tag is always new (meaning it will always initially have
      just one reference), and new_tag.map is always reused from
      tag.map (meaning that no data replication occurs).
      Works also if "tag" and "new_tag" are the same, with the same
      guarantees.
  Inputs:
    ftcore -- The FinTracer core module proxy object.
    tag -- The original tag to be cloned.
  Modified:
    None.
  Outputs:
    new_tag -- The new tag to be created.
"""

confirm_label(ftcore, "FinTracer Core Module{Signature}")
confirm_label(tag, "FinTracer Tag{Signature}")

ftcore.NewTag(tag.keymgr, _temp_tag)

# No need to
# confirm_label(_temp_tag, "FinTracer Tag{Signature}")
# because it was already pre-confirmed at NewTag.
_temp_tag.Clone(tag)

new_tag = _temp_tag
del _temp_tag
confirm_label(new_tag, "FinTracer Tag{Signature}")
```

The above may seem like significant amounts of code, but it covers efficient implementation of essentially anything one may wish to do in local processing of tags. (Most of these scripts will not be used in this entire document, and are meant to support future uses.)

Additionally, each script is provided with a wrapper script. This allows each RE to choose whether to

1. Certify the core operations, in which case these are declared safe for each RE to apply them at will;

2. Certify the wrapper scripts, in which case these are declared safe for use, but only when applied simultaneously over all peer nodes; or

3. Not certify them at all, at which point algorithms using them will need to be certified individually.

For the reasons above, it is good practice to wrap any script that performs a "`modifying()`" operation by a wrapper script, as is demonstrated here. However, for reasons of brevity, this example section will not include such wrappers for the rest of its `modifying` scripts.

Note 4.4.3. Arguably, some of the wrappers presented above are superfluous: e.g., there is no real need to have a `TagCloneWrapper`, given that `TagClone` will fail if called at a scope different to the scope of its tag. The use of wrappers in this case is purely illustrative.

4.4.2 Tag initialisation

To run FinTracer, we must first construct a "tag", v. We have previously described v as a vector of ciphertext values, but in Section 4.4.1 tags were actually stored as dictionaries mapping from accounts to the tag value associated with each account (an ElGamal ciphertext).

In FinTracer, tag values are initially "1" (and more generally a nonzero value) for accounts that are of interest, and "0" otherwise.

One reason to prefer storing v as a dictionary rather than a vector is because it is not immediately clear which are all accounts that should be initialised as zeroes. In theory, we can set to a zero value any account managed by the RE, but this would result in an unnecessarily slow and cumbersome execution of the algorithm due to unneeded bloating of the dictionary.

Instead, the accounts that will be the dictionary keys are ideally all accounts that we wish to have participating in the calculation, all accounts whose tag value, at any point during the FinTracer run, might be associated with a nonzero value.

Though such a tag can be created, in practice FinTracer executions are modular, so we may not know at any given moment what processing of the tag values we may wish to perform in the future. This being the case, accounts that we presently don't think need representation in v may become relevant later on. A dictionary is a convenient storage format to solve this problem, allowing us to add more accounts as we need them.

In fact, it is possible to initialise the dictionary only with the account values that are to receive "1" initialisation, this being the minimal definition of v.

Let us use the tools of Section 4.4.1 to initialise a tag in this way. This can be done as follows.

```
def TagFromAccounts(ftcore, keymgr, accounts, tag):
  """
    Usage:
      TagFromAccounts generates a new FinTracer tag in the system, as a
        dictionary, initially containing encrypted "1"s for all accounts listed
        in 'accounts' and with no other key. Nonexistent keys in tag
        dictionaries are assumed to correspond to zero values.
    Inputs:
      ftcore -- The FinTracer core module proxy object.
      keymgr -- The FinTracer key manager related to this tag.
      accounts -- The account set whose elements will get "1" values in the tag.
    Modified:
      None.
    Outputs:
      tag -- The newly generated FinTracer tag object.
  """

  confirm_label(ftcore, "FinTracer Core Module{Signature}")

  ftcore.NewTag(keymgr, tag)

  # No need to
  # confirm_label(tag, "FinTracer Tag{Signature}")
  # because it was already pre-confirmed in NewTag.

  # No need to
  # confirm_label(accounts, "FinTracer Account Set{Signature}")
  # because this will be confirmed in tag.SetOne.
  tag.SetOne(accounts)
```

4.4.3 Tag enrichment

Though we have initially created our tag, v, as a dictionary containing only the *tag-positive* accounts, i.e. those accounts whose tag value is nonzero, it may be of value to add to the keys of the dictionary other accounts, with zero tag values. We refer to the list of all accounts that are associated with a tag v, regardless of the tag's value, as the *tabulated* accounts, or the *v-tabulated* accounts.

How the *tag-zero* tabulated accounts are chosen makes no functional difference to the algorithm. As long as the tabulated accounts are a superset of the tag-positive accounts, the algorithm will run properly. However, sometimes it may be a concern that an attacker may infer the approximate size of the tag-positive set by, for example, observing each node's processing speed. Though this is typically not an issue, and our later examples will work with the raw-form FinTracer tags, for completion we include the following code that can be used to enrich the tag with zero-valued accounts. It is basically a wrapper for "`tag.AddZero`".

The identity of the accounts added makes no real difference. Only their total amount matters.

```
def EnrichTag(ftcore, tag, accounts):
  """
    Usage:
      EnrichTag (not used by any other script in this document) is a helper
        script, adding a value to each of a specified set of keys in a
        dictionary.
        This is not necessary to do in normal FinTracer operations, but
        may have impacts on execution speed of the first FinTracer iteration,
        and hence may have implications on non-data attacks (regarding which,
        see the chapter on Cryptanalysis).
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      accounts -- An account set. Keys to be added to the tag if not already
                  there.
    Modified:
      tag -- The tag dictionary whose values are being modified.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")

  tag.AddZero(accounts)
```

4.5 Tag propagation

4.5.1 Constructing a two-sided graph

Now that we've defined a list of accounts that are of interest, the next step is to define a relationship between accounts that is of interest. For example, x may be related to y if x has sent more than \$300 to y on more than two separate occasions during the month of May of this year. Alternatively, x may be related to y if there was ever any money exchanged between them, in any direction.

A FinTracer user may wish to define the relationship as in the first example when, for example, trying to follow the money in investigating a money laundering operation, whereas the same user may wish to define the relationship as in the second example when, for example, trying to find which accounts belong to known associates of each other.

There is a fine point here the should be clarified. This point is in the tension between the following two statements, both of which are true.

1. The conditions listed above for a relationship between accounts is in-principle a "two-sided-viewable" condition, in the sense that both the RE managing x and the RE managing y can resolve, independently, whether x relates to y in the relationship defined.

2. The RE managing x and the RE managing y may disagree about whether x relates to y. This is due to various types of data discrepancies between them, including data quality issues.

As an example, consider that while it is no secret to either $R(x)$ or $R(y)$ how many times x transferred money to y in the month of May, a transaction that one RE lists as having happened in May can for many reasons be listed at the other RE as having occurred in either April or June. For this reason, the two REs may reach minor disagreements regarding the overall count of the number of over-$300 transactions between the accounts in May, and can consequently reach a disagreement regarding whether or not the edge should be included.

In order to avoid this, we start the algorithm with a synchronisation step, in which each pair of REs communicate with each other the parts of the graph that in-principle both parties see. We will use as the basic, two-sided-viewable, agreed FinTracer graph the intersection, i.e. the set of those relationships where both $R(x)$ and $R(y)$ agree that the relationship exists.

This can be done as follows. Here, `start_date` and `end_date` define the date range for transactions of interest, and `conditional` is a SQL database query that retrieves the account pairs that are of interest, within the given date range.

For reasons of storage efficiency and downstream processing efficiency, we separate the transactions in the result based on their RE of origin and destination.

```
def NewTwoSidedGraph(ftcore, conditional, start_date, end_date, tsv_graph):
  """
    Usage:
      This script generates the edge-information for a two-sided viewable
        propagation graph, based on a query into the Transaction Store.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      conditional -- A query string to retrieve the desired transaction
                       information from the peer nodes' Transaction Stores.
      start_date -- Start of date range of the query.
      end_date -- End of date range of the query.
    Modified:
      None.
    Outputs:
      tsv_graph -- The output two-sided-graph object, to be labelled
                   "FinTracer Two-Sided Graph". Its members are:
        tx_in -- A transmitter<set<pair<AccountID, AccountID>>> object, where
                   each RE stores their (AccountID, AccountID) pairs that match
                   the conditions of 'conditional', among incoming transactions
                   to that RE. Separate source REs get stored separately by the
                   transmitter.
        tx_out -- Same as tx_in, but for outgoing transactions, and separated
                    according to target RE.
  """

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  _tsv_graph = object()

  on(_peer_ids):
    # Retrieve the relevant transactions from a single RE's perspective.
    _trans1sv = TransactionsRead(conditional, start_date, end_date)
```

```
    # The return values are initialised as empty
    _tsv_graph.tx_in = transmitter<set<pair<AccountID, AccountID>>>()
    _tsv_graph.tx_out = transmitter<set<pair<AccountID, AccountID>>>()

    # We keep temporary values showing the counterparty's view.
    _remote_in = transmitter<set<pair<AccountID, AccountID>>>()
    _remote_out = transmitter<set<pair<AccountID, AccountID>>>()

    for _n in _peer_ids:
      # We strip the AccountAddr pairs into AccountID pairs, filter only the
      #   incoming transactions and separate by counterparty RE.
      _tsv_graph.tx_in[_n] = set{(_trans1sv[i].first.account_id,
                      _trans1sv[i].second.account_id) over i = _trans1sv.iter
                        with _trans1sv[i].first.node_id == _n and
                             _trans1sv[i].second.node_id == MyID}
      _remote_in[_n] = _tsv_graph.tx_in[_n]  # No data copying here.

      _peer_ids_n = setminus(_peer_ids, nodeset(_n))
      on(_peer_ids_n):
        # Repeating as above, but for outgoing transactions.
        # We do not recalculate local transactions.
        _tsv_graph.tx_out[_n] = set{(_trans1sv[i].first.account_id,
                        _trans1sv[i].second.account_id) over i = _trans1sv.iter
                          with _trans1sv[i].first.node_id == MyID and
                               _trans1sv[i].second.node_id == _n}
        _remote_out[_n] = _tsv_graph.tx_out[_n]  # No data copying here.

    # The final result is the intersection of the local and counterparty views.
    transmit(_remote_in)
    transmit(_remote_out)
    for _n in _peer_ids:
      _peer_ids_n = setminus(_peer_ids, nodeset(_n))
      on(_peer_ids_n):
        _tsv_graph.tx_in[_n].intersect(_remote_out[_n])
        _tsv_graph.tx_out[_n].intersect(_remote_in[_n])

    # Local transactions share the same memory in tx_out[MyID] and tx_in[MyID].
    _tsv_graph.tx_out[MyID] = _tsv_graph.tx_in[MyID]  # No copying here, either.

  tsv_graph = _tsv_graph
  del _tsv_graph

  label(tsv_graph, "FinTracer Two-Sided Graph{Signature}")
```

Note that because we're splitting the transactions by REs, we don't need to store `nodeid` information explicitly, so a transaction can now safely be stored as a pair of `AccountID`s.

In terms of certifications, the script `NewTwoSidedGraph` performs privileged `transmit` operations and will therefore need to run in privileged mode, so certification will be required either of it or of a script up the execution stack from it. Certifying it directly indicates that its action is safe irrespective of any context it might be run in. Certifying the larger script that calls it indicates that the extra context is needed.

In this particular case the script is run as one of the first actions in setting up a FinTracer run, so there will be very little extra context to work on. This is why the script was designed from the

get-go to be safe without any needed extra context.

Presently, the only context calling `NewTwoSidedGraph` is the script `Operator1DirFromSQL` introduced later on. Without either one of these being certified, it is impossible presently to run FinTracer propagation code at all. If only the script `Operator1DirFromSQL` is certified but not `NewTwoSidedGraph`, this indicates that the only way to generate a two-sided-viewable graph is through `Operator1DirFromSQL` (or any other script that will be coded for this purpose in the future that the REs decide to certify).

In practice, because calling the script `NewTwoSidedGraph` is the first operation performed by the script `Operator1DirFromSQL`, there is no security or privacy advantage not to certify `NewTwoSidedGraph` directly.

4.5.2 Constructing a one-sided graph

In the previous section we showed how to construct a graph using the information in the Transaction Store which is guaranteed to be (in-principle) visible to both parties.

A different type of query is one that requires information that is not visible to both sides. We may want to ask REs to exclude some transactions (or even some accounts entirely), for example, based on information that they have but may not wish to make public.

For example, we may decide to omit from the graph any account that transacts with more than X other accounts, based on some threshold X. The REs performing the account filtering may not want to share with other REs which accounts have been filtered.

There are many different ways to handle this situation. We will not cover them all here. However, what we present here is handling for a somewhat generic case. We will send to each RE a SQL query (via a string variable, "`conditional`") that will define which pairs we are interested in, as a query into the Auxiliary DB.

What we will use as the FinTracer propagation graph is the intersection of the one-sided-viewable graph and the two-sided-viewable graph, where by one-sided-viewable graph we mean those pairs that have been selected to be part of the graph, independently, by the REs on *both* ends of each pair. Thus, for a transaction to be part of our final propagation graph, it must satisfy four separate criteria, two of which relate to the two-sided-viewable query conditions and two of which relate to the one-sided-viewable query conditions. Only if both participating REs agree that both the one-sided and the two-sided conditions are satisfied is an edge formed in the propagation graph.

In running FinTracer, we will execute it in terms of inter-node communications in a way that only exposes the two-sided-viewable graph, but internally, in each node, we will prune our results, so that the function calculated by FinTracer will be as though the input graph is the intersection graph.

```
def OneSidedTransactions(ftcore, conditional, osv_tx_in, osv_tx_out):
  """
    Usage:
      Find the account pairs matching a SQL conditional. This is good
        for generating one-sided-viewable graphs.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      conditional -- A SQL query retrieving the requested transaction
        set from the peer nodes' Auxiliary DBs.
    Modified:
      None.
    Outputs:
      osv_tx_in -- A transmitter<set<pair<AccountID, AccountID>>> variable,
                   storing the relevant AccountID pairs from incoming
                   transactions, separated according to source RE.
      osv_tx_out -- Same as osv_tx_in, but for outgoing transactions, and
```

```
                    separated according to target RE.
  """

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):
    _temp = AuxDBRead<pair<AccountID, AccountID>>(description)

    osv_tx_in = transmitter<set<pair<AccountID, AccountID>>>()
    osv_tx_out = transmitter<set<pair<AccountID, AccountID>>>()

    # A user my run this script with the same identifier (i.e., name) for both
    # osv_tx_in and osv_tx_out. This would make them into the same variable.
    # Presently, the return value in this case would be the same as what would
    # normally be osv_tx_out. If, instead, the FTIL programmer wishes the
    # script to detect this and abort execution, this can be done via
    #
    #   assert(osv_tx_in != osv_tx_out)

    for _n in _peer_ids:
      osv_tx_in[_n] = set{_temp[i] over i = _temp.iter
                with BranchToBank(_temp[i].first.account_id.bsb_num) == _n
                  and BranchToBank(_temp[i].second.account_id.bsb_num) == MyID}

      _peer_ids_n = setminus(_peer_ids, nodeset(_n))
      on(_peer_ids_n):
        osv_tx_out[_n] = set{_temp[i] over i = _temp.iter
                with BranchToBank(_temp[i].first.account_id.bsb_num) == MyID
                  and BranchToBank(_temp[i].second.account_id.bsb_num) == _n}

    osv_tx_out[MyID] = osv_tx_in[MyID]
```

All in all, the structure of this code is very similar to that of `NewTwoSidedGraph` given above, as is the format of its output. Specific points to note regarding the script include:

1. Unlike in the query to the Transaction Store, when querying the Auxiliary DB we do not specify the date-range information via explicit parameters like `start_date` and `end_date`. Instead, the date range is presumed to be part of the conditional encoded in the SQL query.

2. In this particular example, the SQL query retrieves the `AccountID`s and the system automatically uses `BranchToBank` to add RE information. Asking the query to provide the RE information would have been just as easy to do, assuming this information is in the Auxiliary DB (which it should be, given that all of the Transaction Store data is mirrored in the Auxiliary DB).

3. The code ignores any transaction where the processing node isn't listed as a party to the transaction. Such transactions are unexpected, so keeping them creates the risk that they will be misinterpreted and cause a privacy concern in downstream processing.

4. Local transactions are handled once, without duplication of processing or duplication of memory: `osv_tx_out[MyID]` and `osv_tx_in[MyID]` reference the same entity.

5. Unlike in `NewTwoSidedGraph`, there are no guarantees regarding fit-for-use here, so the returned values are not packed into an `object` or labelled.

4.5.3 Extracting graph vertices

The script below is a helper script, not used elsewhere in this document, that extracts the vertices in a graph defined by a pair of `transmitters`, such as "`(tsv_graph.tx_in, tsv_graph.tx_out)`" or "`(osv_tx_in, osv_tx_out)`". It can be used, for example, to enrich a tag, as described in Section 4.4.3, by use of the return value, `vertices`, as the `accounts` parameter in `EnrichTag`.

```
def Vertices(ftcore, tx_in, tx_out, vertices):
  """
    Usage:
      Vertices reports on the overall set of vertices in the graph described
        by tx_in and tx_out. If this graph is used to propagate tags, these
        are the only accounts that might at any point receive nonzero tags,
        other than those accounts that were initially seeded with nonzero
        values.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      tx_in -- A set of incoming transactions (stored as a transmitter,
                  separating transactions by source RE).
      tx_out -- A set of outgoing transactions (stored as a transmitter,
                   separating transactions by destination RE).
    Modified:
      None.
    Outputs:
      vertices -- The generated set of AccountID vertices.
  """

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):
    # We construct the result in a temporary variable, to make sure we do not
    # accidentally erase an input variable (such as tx_in or tx_out) before
    # needing its value.
    # This is another idiom in frequent use in our code. As can be seen, the
    # construction in a temporary variable is important whether or not the
    # variable is later to be labelled.
    _vertices = set<AccountID>()

    for _n in _peer_ids:
      _temp = tx_in[_n][@].second
      _vertices.union(_temp)
      _temp = tx_out[_n][@].first
      _vertices.union(_temp)

    vertices = _vertices
```

4.5.4 The FinTracer propagation graph object

Now that we have defined both the vertices and the edges of our target graph, let us define this as an `object` instance. Below is a script that receives both the two-sided and one-sided-viewable graphs, and constructs from them a FinTracer propagation graph `object`. The user is free to construct a one-sided graph in any way they deem fit, not just using the script provided, but the two-sided graph needs to have been generated properly, because its two-sided viewability is key to the privacy guarantees of the algorithm.

It is possible to construct other types of FinTracer propagation graphs, e.g. ones without one-sided pruning at all, or ones with one-sided pruning that filters solely based on vertices and not based on edges. What is presented here is both entirely generic and not substantially less efficient than supporting any of these special cases.

The only sanity checking required for the definition of the propagation graph is that the one-sided-viewable data is a sub-graph of the two-sided-viewable graph. We ensure this by intersecting the two.

```
def NewPropagationGraph(ftcore, tsv_graph, osv_in, osv_out, graph):
  """
    Usage:
      NewPropagationGraph creates a FinTracer graph object, which encompasses
        information about both the two-sided-viewable and one-sided-viewable
        aspects of tag propagation. Tags will ultimately be propagated along
        the one-sided-viewable graph, but all communication will be as though
        propagation is according to the two-sided-viewable graph.
    Inputs:
      ftcore -- A module proxy object. (Unused.)
      tsv_graph -- A FinTracer two-sided-viewable graph object.
      osv_in -- One-sided viewable incoming transactions.
      osv_out -- One-sided viewable outgoing transactions.
        osv_out[MyID] is ignored. It is assumed that it equals osv_in[MyID].
    Modified:
      None.
    Outputs:
      graph -- The newly-generated graph object, to be labelled
                 "FinTracer Propagation Graph".
  """

  confirm_label(tsv_graph, "FinTracer Two-Sided Graph{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  _graph = object()
  _graph.tsv = tsv_graph

  on(_peer_ids):
    _graph.osv_tx_in = transmitter<set<pair<AccountID, AccountID>>>()
    _graph.osv_tx_out = transmitter<set<pair<AccountID, AccountID>>>()

    for _n in _peer_ids:
      _graph.osv_tx_in[_n] = intersect(_graph.tsv.tx_in[_n], osv_in[_n])

      _peer_ids_n = setminus(_peer_ids, nodeset(_n))
      on(_peer_ids_n):
        _graph.osv_tx_out[_n] = intersect(_graph.tsv.tx_out[_n], osv_out[_n])

    _graph.osv_tx_out[MyID] = _graph.osv_tx_in[MyID]

  graph = _graph
  del _graph

  label(graph, "FinTracer Propagation Graph{Signature}")
```

4.5.5 Coordinating graph communication

Now that we have set up the FinTracer propagation graph, it is time to convert this graph into a communication regimen that will allow us to transfer tag data between REs.

Now, if all one needed to do at this point was, given a graph G and a tag v, to calculate $M_G v$, this would have been trivial to do in FTIL. However, it is likely that we will wish to use this M_G many times over. For example (as explained in Section 4.6), there is much intel value in being able to compute $M_G^k v$, which is to say that we apply M_G on v k times over, compositionally. In such a situation, it pays to perform some coordination ahead of time, so that any later application of M_G will require the least possible amount of added communication.

Specifically, we will want later communication between REs to be solely in the form of encrypted tag values (typically: as a memblock of such values), so in this initial coordination step all REs now must coordinate with all other REs how many tag values will be sent over in each application of M_G, and how to interpret the tag value in each position.

There are two distinct items that need to be coordinated. First, all REs should agree on a general communications strategy between each other. The choice of strategy determines *which* tag values need to be communicated between each pair of REs.

Second, given an agreed strategy, the REs communicate to each other the *order* in which they will send the necessary values.

Note 4.5.1. The strategy does not need to be decided on globally. It can be that each pair of REs agrees on a separate strategy between each other (although, presumably the choice of strategy itself carries information that REs may not wish to circulate too freely).

The SIMD architecture of FTIL makes the *coordination* of such individual communication strategies potentially slightly less efficient than the coordination of a single global strategy, but the actual execution of the strategies should be as efficient in either case. We do not implement such individualised strategies here, solely to keep the code as simple as possible.

Let us now introduce three possible communication strategies. We begin with the simplest and least efficient of the strategies, which we call the *uncompacted* communications strategy. Here, we wish each $R(x)$ to communicate the value of $v[\tilde{x}]$ to each $R(y)$ separately for each edge (x, y) in the graph G.

At face value, it may seem like this communication strategy is woefully inefficient: if the graph contains both (x, y) and (x, z), with $R(y) = R(z)$, why should $R(x)$ communicate $v[\tilde{x}]$ to $R(y)$ twice? The answer is in the fact that the graph in question, G', that determines the communication patterns between REs is solely the two-sided-viewable graph. The actual graph used by the REs, G, is the narrower one-sided-viewable graph, which we derived by intersecting the two-sided-viewable graph with the one-sided-viewable graph. The communications strategy that communicates a tag value for each edge separately is one that provides the greatest amount of flexibility in specifying this underlying, one-sided-viewable graph.

The two scenarios to keep in mind are the following.

- It may be the case that $R(x)$ does not want the edge (x, y) in the graph. In order to omit the edge from the graph without alerting $R(y)$, all $R(x)$ needs to do is to communicate a zero value to $R(y)$, instead of communicating $v[\tilde{x}]$, when delivering the tag value associated with (x, y). The RE $R(y)$ can then sum this value (blindly) into the new tag value for y, not knowing that this summation will not alter the result. Our choice of an uncompacted communications strategy allows $R(x)$ to modify the edge (x, y) in this way, without this modification impacting any other edge, such as for example any (x, z).

- Alternatively or additionally, it may be the case that $R(y)$ does not want the edge (x, y) in the graph. In order to omit the edge from the graph without alerting $R(x)$, all $R(y)$ needs to do is to ignore the tag value communicated to it that is associated with the edge (x, y) and not add it into the new tag value for y. This will cause the edge not to impact the new tag value for y, without this causing any impact to any other edge, such as, for example, any (w, y), with $R(w) = R(x)$.

The following code prepares a communication regimen for uncompacted communication. Such a regimen is composed of three elements:

rm: A mapping detailing how incoming communication is to be mapped into the node's internal tag representation (as in: "the tag value received at position 7 should be associated with account x"),

rm: A mapping detailing how the internal tag representation is to be mapped into outgoing communication (as in: "the tag value associated with account x should be the 9th tag value sent out), and

rm: The total number of tag values that the RE should send to each peer node when calculating $M_G v$.

```
def PrepUncompacted(graph, in_mapping, out_mapping, out_size):
  """
    Usage:
      PrepUncompacted prepares a communications regimen that will be used to
        propagate tags along the graph specified by 'graph'. A communications
        regimen is composed of a triplet (in_mapping, out_mapping, out_size).
        See below for the meaning of the individual variables. The script
        prepares a communications regimen related to propagating the tag values
        in an 'uncompacted' way, which is to say that each edge in the graph
        has the tag value that propagates along it communicated separately.
    Inputs:
      graph -- The FinTracer graph object.
    Modified:
      None.
    Outputs:
      in_mapping -- Stored as a transmitter<memblock<pair<int, AccountID>>>,
        this variable determines how incoming tag values should be processed.
        Each pair value (A, B) associated with some RE r in this transmitter
        indicates that when receiving a list (memblock) of encrypted tag values
        from r, the A'th value should be added to the output tag value
        associated with account B.
      out_mapping -- Stored as a transmitter<memblock<pair<AccountID, int>>>,
        this variable determines how outgoing tag values should be processed.
        Each pair value (A, B) associated with some RE r in this transmitter
        indicates that when sending a list (memblock) of encrypted tag values
        to r, the tag value associated with account A should be added to the
        B'th value communicated out.
     out_size -- Stored as a transmitter<int>, this number tells each RE how
        many values it should send out to each of the other REs.
  """

  confirm_label(graph, "FinTracer Propagation Graph{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):

    _tx_list = transmitter<memblock<pair<AccountID, AccountID>>>()
    _out_mapping = transmitter<memblock<pair<AccountID, int>>>()
    _in_mapping = transmitter<memblock<pair<int, AccountID>>>()
    _out_size = transmitter<int>()
```

```
for _n in _peer_ids:
  _out_size[_n] = graph.tsv.tx_out[_n].size()

  # We use _tx_list to store the values of graph.tsv.tx_out[_n] in an
  # arbitrary order. This will correspond to the final sending order of tag
  # values.
  _tx_list[_n] = memblock<pair<AccountID, AccountID>>(graph.tsv.tx_out[_n])

  _outgoing = graph.osv_tx_out[_n]
  _out_dict = dict(_tx_list[_n], range(_out_size[_n]))
  _out_mapping[_n] = memblock{(_outgoing[i].first, _out_dict[_outgoing[i]])
                              over i = _outgoing.iter}

  del _outgoing
  del _out_dict

transmit(_tx_list)

for _n in _peer_ids:
  _incoming = graph.osv_tx_in[_n]
  _in_dict = dict(_tx_list[_n], range(_tx_list[_n].size()))
  _in_mapping[_n] = memblock{(_in_dict[_incoming[i]], _incoming[i].second)
                             over i = _incoming.iter}

  del _incoming
  del _in_dict

out_mapping = _out_mapping
in_mapping = _in_mapping
out_size = _out_size
```

Note 4.5.2. Recall that when synchronising the views of the two-sided-viewable transaction graph in Section 4.5.1 all information necessary for this coordination step had already been exchanged, so it may seem a little superfluous that back in Section 4.5.1 we exchanged the information using `rm`s, which give no order information, while now we are re-sending a subset of the same information only using a `rm` entity, solely for the purpose of communicating an arbitrarily-chosen order. In truth, a more elaborate (and harder to implement) process may have saved this extra communication. However, given that what is exchanged here is `rm`s, which are much smaller than `rm`s, the extra overhead does not seem to be worth the added complication.

Note 4.5.3. In the above script and the two scripts that follow, we use the idiom of constructing return values in temporary values and only assigning them as a final step. This is done for illustrative purposes and is not really necessary here. The reason it is not necessary here is that these scripts are never exposed as object methods, so there is no way to call them from outside their own module. It is easily verifiable that no script within this module calls any of these scripts with the kind of dubious parameters that this form of defensive programming is meant to protect against.

We now introduce two other communication strategies. The former is *Incoming-compacted* communication and the latter *Outgoing-compacted* communication.

In incoming-compacted communication, for each account y and each RE r, r communicates a single tag value that encapsulates the information relating to all edges in the two-sided-viewable graph G' that are of the type (x, y), where $R(x) = r$.

While more concise than the uncompacted communication strategy, this strategy is not always viable. Its problem is that if $R(y)$ is managing a one-sided-viewable graph, G, that is different to G', it needs to be able to use the information from $R(x)$ in order to update the tag value of y in a way consistent with G, rather than with G'. The rule is that a graph is *Incoming-compactable* if for every such y and each such r, either all edges (x, y) in G' for which $R(x) = r$ are in G or all of them are not in G.

This is a common situation (arising, for example, when the REs want to prune entire vertices from the graph), but is certainly not inevitable.

The following code checks whether a given graph is incoming compactable (This is returned in the variable `is_in_compactable`), and if so returns the appropriate communication regimen for it.

```
def PrepIncomingComp(graph, is_in_compactable, in_mapping, out_mapping,
                     out_size):
  """
    Usage:
      PrepIncomingComp prepares a communications regimen that will be used to
        propagate tags along the graph specified by 'graph'. See
        'PrepUncompacted' for details. The script prepares a communications
        regimen in which all values that are to impact a single tag value at
        the receiving RE are sent out summed together.
    Inputs:
      graph -- The FinTracer graph object.
    Modified:
      None.
    Outputs:
      is_in_compactable -- A local Boolean determining whether the input graph
        fits this type of communications regimen. If it does not, the other
        output variables are untouched.
      in_mapping, out_mapping, out_size -- If is_in_compactable is true, the
        meaning of these variables is as in 'PrepUncompacted'.
  """

  confirm_label(graph, "FinTracer Propagation Graph{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  _temp_compactable = transmitter<bool>()

  on(_peer_ids):
    _temp_compactable[CoordinatorID] = True
    _accts_in_set = transmitter<set<AccountID>>()
    _tx_in_accounts = transmitter<memblock<AccountID>>()
    for _n in _peer_ids:
      _incoming = graph.tsv.tx_in[_n]

      # Extracting the set of accounts that receive tags on the
      # two-sided-viewable graph, removing any duplicates
      _tx_in_accounts_set = _incoming[@].second

      # Converting to a memblock, to place in an arbitrary order.
      # This will be the sending order.
      _tx_in_accounts[_n] = memblock<AccountID>(_tx_in_accounts_set)
```

```
      _in_osv_tx = graph.osv_tx_in[_n]

      # Removing all duplicates by converting to a set.
      _accts_in_set[_n] = _in_osv_tx[@].second

      _incoming.filter[_incoming[@].second in _accts_in_set[_n]]

      # The next line checks for in-compactability. If every two-sided-viewable
      # transaction leading to a "destination account that appears in the
      # one-sided-viewable graph" also appears as a transaction in the
      # one-sided-viewable graph, the graph is, by definition, in-compactable.
      _temp_compactable[CoordinatorID]
         &= (_incoming.size() == _in_osv_tx.size())

  # Because we are coordinating a single global strategy, all flags computed
  # locally are now collated at the coordinator node.
  transmit(_temp_compactable)

  on(CoordinatorID):
    _is_in_compactable_local = True
    for _n in _peer_ids:
      _is_in_compactable_local &= _temp_compactable[_n]

  # We now make our locally computed variable shared among all nodes in scope.
  _is_in_compactable = broadcast(_is_in_compactable_local)
  del _is_in_compactable_local

  if(_is_in_compactable):
    on(_peer_ids):
      _in_mapping = transmitter<memblock<pair<int, AccountID>>>()
      for _n in _peer_ids:
        _in_dict = dict(_tx_in_accounts[_n], range(_tx_in_accounts[_n].size()))
        _in_mapping[_n] = memblock{(_in_dict[_accts_in_set[_n][i]],
           _accts_in_set[_n][i]) over i = _accts_in_set[_n].iter}
        del _in_dict

        # An alternative to the previous 3 commands would have been:
        # _in_mapping[_n] = memblock{(i, _tx_in_accounts[_n][i])
        #                     over i = _tx_in_accounts[_n].iter
        #                     with _tx_in_accounts[_n][i] in _accts_in_set[_n]}

      _out_mapping = transmitter<memblock<pair<AccountID, int>>>()
      _remote_in_accounts = transmitter<memblock<AccountID>>()
      for _n in _peer_ids:
        _remote_in_accounts[_n] = _tx_in_accounts[_n]

      # We now perform a 'transmit'. Note, however, that because transmitting
      # is a write operation, this only impacts the transmitted variable
      # _remote_in_accounts, which is now copied away from and does not
      # influence the value of _tx_in_accounts
      transmit(_remote_in_accounts)

      _out_size = transmitter<int>()
      for _n in _peer_ids:
```

```
        _out_size[_n] = _remote_in_accounts[_n].size()
        _remote_dict = dict(_remote_in_accounts[_n],
                                range(_remote_in_accounts[_n].size())
        _out_osv_tx = graph.osv_tx_out[_n]
        _out_mapping[_n] = memblock{(_out_osv_tx[i].first],
          _remote_dict[_out_osv_tx[i].second]) over i = _out_osv_tx.iter}

  # Completing the idiom by copying all temporary variables to their final
  # destination.
  is_in_compactable = _is_in_compactable
  on(_peer_ids):
    out_mapping = _out_mapping
    in_mapping = _in_mapping
    out_size = _out_size
```

The last compacting option is to be *outgoing-compacted*. This occurs when for each account x and each RE r, the RE $R(x)$ reports together the values associated with all edges (x, y) such that $R(y) = r$. If $G = G'$, this value is simply $v[\tilde{x}]$. However, again, not all transaction graphs are compactable in this way. If the one-sided-viewable graph requires from r distinct treatments for distinct edges that all originate from x, the graph is not *outgoing-compactable*.

The script below is a mirror-image of `PrepIncomingComp`. It determines whether an input graph is outgoing-compactable, and if it is, it computes the communication regimen for it.

```
def PrepOutgoingComp(graph, is_out_compactable, in_mapping, out_mapping,
                     out_size):
  """
    Usage:
      PrepOutgoingComp prepares a communications regimen that will be used to
        propagate tags along the graph specified by 'graph'. See
        'PrepUncompacted' for details. The script prepares a communications
        regimen in which each communicated value corresponds to a single tag
        value at the source RE.
    Inputs:
      graph -- The FinTracer graph object.
    Modified:
      None.
    Outputs:
      is_out_compactable -- A local bool determining whether the input graph
        fits this type of communications regimen. If it does not, the other
        output variables are untouched.
      in_mapping, out_mapping, out_size -- If is_out_compactable is true, the
        meaning of these variables is as in 'PrepUncompacted'.
  """

  confirm_label(graph, "FinTracer Propagation Graph{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  _temp_compactable = transmitter<bool>()

  on(_peer_ids):
    _temp_compactable[CoordinatorID] = True
    _accts_out_set = transmitter<set<AccountID>>()
    _tx_out_accounts = transmitter<memblock<AccountID>>()
```

```
    for _n in _peer_ids:
      _outgoing = graph.tsv.tx_out[_n]

      # Extract origin accounts. Remove duplicates.
      _tx_out_accounts_set = _outgoing[@].first

      # Set in arbitrary order.
      _tx_out_accounts[_n] = memblock<AccountID>(_tx_out_accounts_set)

      _out_osv_tx = graph.osv_tx_out[_n]

      # Extract 1sv origin accounts. Remove duplicates.
      _accts_out_set[_n] = _out_osv_tx[@].first

      _outgoing.filter[_outgoing[@].first in _accts_out_set[_n]]

      # The next line checks for out-compactability. If every two-sided-viewable
      # transaction originating from a "source account that appears in the
      # one-sided-viewable graph" also appears as a transaction in the
      # one-sided-viewable graph, the graph is, by definition, out-compactable.
      _temp_compactable[CoordinatorID]
        &= (_outgoing.size() == _out_osv_tx.size())

  # Because we are coordinating a single global strategy, all flags computed
  # locally are now collated at the coordinator node.
  transmit(_temp_compactable)

  on(CoordinatorID):
    _is_out_compactable_local = True
    for _n in _peer_ids:
      _is_out_compactable_local &= _temp_compactable[_n]

  # We now make our locally computed variable shared among all nodes in scope.
  _is_out_compactable = broadcast(_is_out_compactable_local)
  del _is_out_compactable_local

  if(_is_out_compactable):
    on(_peer_ids):
      _out_size = transmitter<int>()
      _out_mapping = transmitter<memblock<pair<<AccountID, int>>>()

      for _n in _peer_ids:
        _out_size[_n] = _tx_out_accounts[_n].size()
        _out_dict = dict(_tx_out_accounts[_n],
                         range(_tx_out_accounts[_n].size()))
        _out_mapping[_n] = memblock{(_accts_out_set[_n][i],
                                     _out_dict[_accts_out_set[_n][i]])
                                       over i = _accts_out_set[_n].iter}
        del _out_dict

      _in_mapping = transmitter<memblock<pair<int, AccountID>>>()

      _remote_out_accounts = transmitter<memblock<AccountID>>()
```

```
      for _n in _peer_ids:
        _remote_out_accounts[_n] = _tx_out_accounts[_n]

      transmit(_remote_out_accounts)

      for _n in _peer_ids:
        _remote_dict = dict(_remote_out_accounts[_n],
                            range(_remote_out_accounts[_n].size())
        _in_osv_tx = graph.osv_tx_in[_n]
        _in_mapping[_n] = memblock{(_remote_dict[_in_osv_tx[i].first],
          _in_osv_tx[i].second) over i = _in_osv_tx.iter}

  is_out_compactable = _is_out_compactable
  on(_peer_ids):
    out_mapping = _out_mapping
    in_mapping = _in_mapping
    out_size = _out_size
```

4.5.6 The basic FinTracer operator

Now that we have constructed a communication regimen that defines how tag values are propagated in the system, it is time to actually program the script that executes a single forward FinTracer step.

For reasons that will become clear momentarily, we wish this script to work as a method of an `object` instance, `op`, which we will construct later on and label "`Operator 1Dir`".

The code that performs this single forward step is the following. Interestingly, not only does the code work with any of the communication regimens discussed above, it is completely oblivious to the question of which regimen is the one actually being used.

Note 4.5.4. If one wants to implement different communications regimen between different pairs of REs, this obliviousness of `rm` to the choice of regimen is key.

```
def FTForward(op, tag, output_tag):
  """
    Usage:
      FTForward propagates a tag's values along an input graph, with the
        result then being stored in output_tag. The script is structured as a
        method of an object (with the label "Operator 1Dir") that handles
        FinTracer tag propagation, and, in particular, stores the necessary
        graph data by means of a chosen communications regimen. (See
        'PrepUncompacted' for an explanation of communications regimens.)
        output_tag may be the same variable as tag.
        The script should be executed at exactly the same scope as 'tag', and
        this is the scope in which the new 'output_tag' will be created.
        (If 'output_tag' existed previously, it should have had the same scope
        as the script's execution scope. This is almost invariably the case
        with FTIL output variables.)
    Inputs:
      op -- The FinTracer operator object.
      tag -- The tag to be propagated.
    Modified:
      None.
    Outputs:
      output_tag -- The resulting tag
```

```
  """

  # This script is resilient to tag and output_tag being the same.

  confirm_label(op, "Operator 1Dir{Signature}")
  confirm_label(tag, "FinTracer Tag{Signature}")
  assert(scope() == tag.metadata.scope)

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):
    _t = transmitter<memblock<ElGamalCipher>>()
    for _n in _peer_ids:
      _t[_n] = memblock<ElGamalCipher>(op.out_size[_n], tag.keymgr.zero)

      # The next line, despite the '=', actually performs a summing.
      _t[_n][op.out_mapping[_n][@].second] =
                                      {tag.map[op.out_mapping[_n][@].first]}

      _peer_ids_n = setminus(_peer_ids, nodeset(_n))
      on(_peer_ids_n) tag.keymgr.ElGamalRefresh(_t[_n])

    transmit(_t)

  op.ftcore.NewTag(tag.keymgr, output_tag)

  # No need to
  # confirm_label(output_tag, "FinTracer Tag{Signature}")
  # because it was already pre-confirmed in NewTag.

  modifying(output_tag):
    on(_peer_ids):
      for _n in _peer_ids:
        # The next line performs a summation.
        output_tag.map[op.in_mapping[_n][@].second] =
                                    {_t[_n][op.in_mapping[_n][@].first]}
    modifying(output_tag.sources):
      op.ftcore.NewSource(_source)
      output_tag.sources[_source.metadata.id] = _source
```

If the output tag was a vector, w and the input tag was a vector v, the script above is a script that implements $w \leftarrow M_G v$. In some circumstances, however, it is more conducive to have a script that assumes a pre-initialised w, which performs $w \leftarrow w + M_G v$. The following script performs this variation.

```
def FTForwardInc(op, tag, output_tag):
  """
    Usage:
      Same as 'FTForward', except that instead of simply writing the resulting
        tag value to output_tag, this method adds the result to any previously
        existing value in output_tag.
        output_tag may be the same variable as tag.
    Inputs:
      op -- The FinTracer operator object.
```

```
    tag -- The tag to be propagated.
  Modified:
    output_tag -- The tag to which the result gets added.
  Outputs:
    None.
"""

# This script is resilient to tag and output_tag being the same.

confirm_label(op, "Operator 1Dir{Signature}")
confirm_label(tag, "FinTracer Tag{Signature}")
confirm_label(output_tag, "FinTracer Tag{Signature}")
assert(tag.keymgr == output_tag.keymgr)

_peer_ids = setminus(scope(), nodeset(CoordinatorID))

on(_peer_ids):
  _t = transmitter<memblock<ElGamalCipher>>()
  for _n in _peer_ids:
    _t[_n] = memblock<ElGamalCipher>(op.out_size[_n], tag.keymgr.zero)

    # The next line, despite the '=', actually performs a summing.
    _t[_n][op.out_mapping[_n][@].second] =
                                    {tag.map[op.out_mapping[_n][@].first]}

    _peer_ids_n = setminus(_peer_ids, nodeset(_n))
    on(_peer_ids_n) tag.keymgr.ElGamalRefresh(_t[_n])

  transmit(_t)

modifying(output_tag):
  on(_peer_ids):
    for _n in _peer_ids:
      output_tag.map[op.in_mapping[_n][@].second] +=
                                  {_t[_n][op.in_mapping[_n][@].first]}
  modifying(output_tag.sources):
    op.ftcore.NewSource(_source)
    output_tag.sources[_source.metadata.id] = _source
```

Some notes regarding the two scripts above:

1. The fact that refreshing happens under the internal "`on()`" clause causes no refreshing to occur when an RE communicates tag values to itself, a process that does not require any physical sending. This saves on the usage of encrypted zeroes, the generation of which is perhaps the heaviest part of the entire algorithm.

2. Of the online portions of the FinTracer algorithm, the heaviest parts are likely to be the mass reduction commands in the above two scripts. From an implementation standpoint, these are the lines most important to have running efficiently, and specifically are the ones most important to have running efficiently on a GPU.

3. When we defined the FinTracer tag object, none of the tag operations allowed transfer of information between different account IDs in the tag's mapping. This was important, because it allows proper maintenance of differential privacy guarantees. FinTracer tag propagation breaks this rule, in that tags do pass information between accounts, but for the purpose

of software implementation we use here the approximation that no *usable* information is passed between accounts in FinTracer tag propagation. The value $M_G v$ certainly contains information, but this information is considered independent to the information content of v. The way to convey this to the system is to assign to $M_G v$ the new information source "`_source`". This approximation may not hold in practice, and in Section 10.1 we discuss means other than privacy-preserving protocols that can be used to mitigate the risks posed by this issue.

It is now possible to package everything we have constructed into the basic FinTracer operator.

```
def Operator1DirFromGraph(ftcore, graph, op):
  """
    Usage:
      Operator1DirFromGraph constructs an object to serve as a FinTracer tag
        propagation operator, and labels it "Operator 1Dir".
    Inputs:
      ftcore -- The FinTracer core module.
      graph -- The graph object from which the operator is constructed.
    Modified:
      None.
    Outputs:
      op -- The generated FinTracer operator object, to be labelled
              "Operator 1Dir".
  """

  confirm_label(ftcore, "FinTracer Core Module{Signature}")
  confirm_label(graph, "FinTracer Propagation Graph{Signature}")

  _op = object()

  PrepOutgoingComp(graph, is_out_compactable, _op.in_mapping, _op.out_mapping,
                   _op.out_size)
  if not is_out_compactable:  # Note that this is a shared variable.
    PrepIncomingComp(graph, is_in_compactable, _op.in_mapping, _op.out_mapping,
                     _op.out_size)
    if not is_in_compactable:  # Note that this is a shared variable.
      PrepUncompacted(graph, _op.in_mapping, _op.out_mapping, _op.out_size)

  _op.ftcore = ftcore

  _op.Forward = ::FTForward
  _op.ForwardInc = ::FTForwardInc

  op = _op
  del _op

  label(op, "Operator 1Dir{Signature}")
```

The construction of the FinTracer operator object is a good opportunity to address some questions that come up in many similar contexts in FTIL. We will frame these as specific questions about the above code and provide concrete answers to these questions, but the reader should see our answers are pointing to general principles that are applicable elsewhere as well. We will not address the same set of questions with every script and every object, but the reader can revisit the discussion here to reach analogous conclusions.

Do these scripts need to be certified? Script certification is never mandatory, but not certifying a script can make it unusable, restrict its usage to particular contexts, or limit what it can perform. If the question is "Can these scripts be certified?", the answer is that scripts are good candidates for certification if they've been designed to be safe even if used from an uncontrolled environment (e.g., from the command line). Scripts that do not meet this criterion are best tagged by the modifier "`NOCERT`" so as to prevent their accidental certification. All example code in this chapter is either safe or flagged as "`NOCERT`". Some code is never exposed outside its own module, so literally can't be used from an uncontrolled environment, regardless of certification. A different question is "Should an RE certify this script?". This is a question ultimately left for each RE to decide individually. It boils down to whether that RE feels comfortable with allowing the script to run its privileged operations, whether from the known, controlled environments or, if applicable, from uncontrolled environments. In this particular case, the scripts perform a "`transmit`", so will not be able to work outside privileged mode. The RE needs to be satisfied that the way the propagation graph was constructed and the way that it is used guarantees that RE's privacy requirements (which it was designed to do). Yet a further question is "Does the code rely on certification?". In this case, the script is certainly intended for certification: it uses privileged operations, and the intent is to use it considerably from the command line (an uncontrolled environment) making it require certification for its core functionality. Moreover, the intent, as demonstrated in Section 4.6, is to use this first tag propagation function as the basic Lego block using which all our operator arithmetic will be defined. By certifying this one building block, every operator that can be derived from it will be fully-functional out-of-the-box with no need for any further certification. If these three scripts are not certified, on the other hand, this severely hampers the versatility that FinTracer can provide without case-by-case certification, so lowers the value of the platform and of the FTIL-based solution as a whole.

What does the label "`rm`" mean? Unlike other languages, where declarations signify particular interface guarantees that can then be verified automatically (e.g., by a compiler), in FTIL a label has no particular technical interpretation. Part of the Purgles solution will be a roster, maintained by AUSTRAC and visible to the REs, which will give a precise, semantic definition of what each label guarantees regarding its underlying object. An RE should not certify a script if it labels anything that may, at any save point in the code no matter how briefly, fail to satisfy the declared guarantees. In the case of "`Operator 1Dir`", the intent was that the label guarantees the existence of methods "`.Forward`" and "`.ForwardInc`" that propagate tags along some graph, either overwriting the result or summing to it. The label does not guarantee, however, the existence of members like "`out_mapping`", "`in_mapping`" and "`out_size`". We will, in fact, later create other objects, with the same label, which do not have these member attributes.

How can the code rely on unguaranteed member attributes? This question is about the following: if "`rm`" doesn't guarantee the existence of member attributes like "`rm`", "`rm`" and "`rm`", how can the script code in `FTForward` and `FTForwardInc` rely on them? It is easy to verify that `FTForward` and `FTForwardInc` are only used inside their own module by `Operator1DirFromGraph`. This is a script that sets up a specific type of `object` in a specific type of way and then binds it with the two scripts discussed. Even had the `object` not been labelled at all, there is no way to run these scripts other than as methods of an `object` that was created by `Operator1DirFromGraph`. Hence, `FTForward` and `FTForwardInc` can freely assume any property that is guaranteed by an `object` created this way.

So, why do we confirm the "`rm`" label? On the face of it, if the label of the script parameter `op` does not guarantee that it has any particular member attributes, if `FTForward` and `FTForwardInc` do not call any of the `op` `object`'s method attributes, and if, as discussed above, the very fact that their code is called implies that `op` was created by `Operator1DirFromGraph`, what reason is there to confirm this `object`'s label? Actually, such confirmation is very much necessary, and serves two purposes. First, while calling either of `FTForward` and

`FTForwardInc` indicates that the `object` was created by the script `Operator1DirFromGraph`, it doesn't guarantee that the execution of this script was successful. Perhaps it aborted somewhere in the middle of the code, e.g. due to a user interrupt? By confirming the label we verify that the `object`'s construction was completed successfully. Second, if the label is not confirmed, this opens the possibility that while the `object` was created by `Operator1DirFromGraph`, it may not have been created in privileged mode. Consider a user who manually creates a graph and then manually labels it as two-sided viewable. That is a weak label, not to be trusted, but it is still a label. A user can use this graph and run `Operator1DirFromGraph` in an unprivileged context (if it is not certified) in order to create an untrustable but weakly labelled operator `object`. If this `object` is now used to trigger, e.g., `FTForward` (assuming `FTForward` is certified) it is only this label confirmation that detects the weakness of the label and stops the script before it `transmit`s tag values to uncontrolled destinations.

4.5.7 The FinTracer core module

There is no hard-and-fast rule regarding how to divide scripts into modules. The two main considerations to balance are:

- Scripts in the same module can always call each other, regardless of certification, whereas calling a script from another module requires having an access object, having that object labelled and having that label certified. and if the calling script is running in privileged mode, the label must additionally be strong. This makes scripts in the same module more easily interoperable, even if they are not suitable for use from an uncontrolled environment.
- On the other hand, if any single script inside a module needs to be updated, the entire module will need updating, and all of the previously-certified scripts will require re-certification in order to regain their lost functionality.

Many scripts do not need certification at all, however, making for them the latter point moot.

With these considerations in mind, let us separate the functionality that has been defined thus far into its own module, even though there will be additional functionality later on that can equally fit in one module with all of the above.

The `__init` script for this module may look something like this (noting that many of these scripts are merely convenience scripts, and not really a necessity for the core FinTracer module).

```
def __init(ftcore):
  """
    Usage:
      __init script for the core FinTracer module.
    Inputs:
      None.
    Modified:
      None.
    Outputs:
      ftcore -- The new module proxy object generated, to be labelled
                  "FinTracer Core Module".
  """

  ftcore = object()

  ftcore.NewSource = ::NewSource
  ftcore.CloneToNewAccountSet = ::CloneToNewAccountSet
  ftcore.AccountSetUnion = ::AccountSetUnion
  ftcore.AccountSetIntersection = ::AccountSetIntersection
```

```
  ftcore.AccountSetDifference = ::AccountSetDifference

  ftcore.AccountsFromList = ::AccountsFromList
  ftcore.AccountsFromDescription = ::AccountsFromDescription
  ftcore.AccountAddrsFromAccountIDs = ::AccountAddrsFromAccountIDs
  ftcore.AccountsFromSQL = ::AccountsFromSQL
  ftcore.NewTag = ::NewTag
  ftcore.CloneToNewTag = ::CloneToNewTag
  ftcore.TagFromAccounts = ::TagFromAccounts
  ftcore.EnrichTag = ::EnrichTag

  ftcore.NewTwoSidedGraph = ::NewTwoSidedGraph
  ftcore.OneSidedTransactions = ::OneSidedTransactions
  ftcore.Vertices = ::Vertices
  ftcore.NewPropagationGraph = ::NewPropagationGraph
  ftcore.Operator1DirFromGraph = ::Operator1DirFromGraph

  label(ftcore, "FinTracer Core Module{Signature}")
```

Suppose that this module's file, with the preamble

```
FTIL1.0
module ftcore
```

was stored in the URL "`file://ftcore.ftil`". Then running a basic FinTracer run may have looked something like this.

First, we want to import the FinTracer core module and define an operator.

```
# If we want to use a module library object but don't already
# have one, we begin by creating a new one.
modules = object()

# Import module
import "file://ftcore.ftil" to modules

# Create two-sided-viewable graph from a SQL conditional and date range.
modules.ftcore.NewTwoSidedGraph(conditional, start_date, end_date, tsv_graph)

# Add any whitelisting. Here this is defined based on a SQL conditional.
modules.ftcore.OneSidedTransactions(conditional, osv_tx_in, osv_tx_out)

# Create the propagation graph
modules.ftcore.NewPropagationGraph(tsv_graph, osv_in, osv_out, graph)

# Create the operator
modules.ftcore.Operator1DirFromGraph(graph, op)
```

If we think that this is a common-enough way to set up the operator, we can provide a convenience script for it. This does not need to be certified, nor even to be a signed script. Here is what such functionality might look like in an unsigned script:

```
def Operator1DirFromSQL(ftcore, two_sided_cond, start_date, end_date,
                        one_sided_cond, op):
  """
    Usage:
      Operator1DirFromSQL is a convenience function for creating a
```

```
        FinTracer one-directional operator from SQL queries.
    Inputs:
      ftcore -- The FinTracer core module proxy object.
      two_sided_cond -- The SQL conditional defining the two-sided query.
      start_date -- Earliest date to match the two-sided query.
      end_date -- Latest date to match the two-sided query.
      one_sided_cond -- SQL conditional defining one-sided query,
                          not time-bounded.
    Modified:
      None.
    Outputs:
      op -- The new FinTracer operator, to be labelled "FinTracer 1Dir".
  """

  confirm_label(ftcore, "FinTracer Core Module{Signature}")

  ftcore.NewTwoSidedGraph(two_sided_cond, start_date, end_date, _tsv_graph)
  ftcore.OneSidedTransactions(one_sided_cond, _osv_in, _osv_out)
  ftcore.NewPropagationGraph(_tsv_graph, _osv_in, _osv_out, _graph)
  ftcore.Operator1DirFromGraph(_graph, op)
```

The above script performs the same actions as the command-line session above it, with the only difference being that the script, at its end, deletes all intermediate variables.

With the convenience script in place, the command-line session becomes simply

```
import "file://ftcore.ftil" to modules
Operator1DirFromSQL(modules.ftcore, two_sided_cond, start_date, end_date,
                    one_sided_cond, op)
```

Note that `modules.ftcore` now needs to be sent to the script as a parameter, which is FTIL's way of importing functionality into a script.

Had `Operator1DirFromSQL` been a module script, which it is not, it could have also imported functionality from any other module that its module was "`using`", without requiring the extra module proxy to be sent explicitly as a parameter to the script. (The module proxy would have been sent automatically, instead, to the module's `__init` as part of its creation.) We demonstrate this process in Section 4.11, where we not only define the script as a module script, but also call it from another script.

One problem with defining `Operator1DirFromSQL` as a non-module script is that unsigned scripts can only be invoked by the user who defined them. Scripts whose functionality is intended for broader access need to be defined as module scripts. Making a script into a module script does not require any certification.

Once the graph has been set up, the next step is to create a FinTracer tag, v. This will also involve importing our other module, namely the key manager module. Perhaps we wish to define the accounts in the tag's initial value by use of a SQL query. Such a process may look like this:

```
import("file://ftkeys.ftil") to modules
modules.ftkeys.NewMgr(keymgr)
keymgr.AddZeroes()  # Populating the key manager's zero stockpile.

modules.ftcore.AccountsFromSQL(conditional, accounts)
modules.ftcore.TagFromAccounts(keymgr, accounts, v)
```

It is now possible to calculate $w = M_G v$ by

```
op.Forward(v, w)
```

and to compute $w \leftarrow w + M_G v$ by

```
op.ForwardInc(v, w)
```

4.6 FinTracer operator arithmetic

4.6.1 Composing operators

Consider what we have so far. We have the ability to specify an account set, S, to create a tag, v, that is 1 for all accounts in the set and 0 for all other accounts, to define a propagation graph, G, stipulating for which (ordered) account pair (x, y), account x connects to account y, and we are able to calculate $w = M_G v$.

The new values in w are now no longer restricted to be 0 or 1. If an account y connects from c different accounts in S, the tag value of y in w will be c. A convenient mathematical way to think about this is that $w[\tilde{y}]$ counts the number of *walks* of length 1 in the graph G that start at any account in S and ends at y.

In intel usage we are often just interested to know whether any such walks exist, in which case we can simply compare the result to zero.

(A *walk* in a graph G is a standard concept in graph theory. It is a sequence of vertices of G, $v_1, \ldots, v_k$, such that for each i in $[1, k)$, (v_i, v_{i+1}) is an edge of G. Vertices in the sequence may repeat. This is different to a *path* in the graph, which is defined in the same way, except with the restriction that vertices in a path may not repeat.)

What makes this construction exciting is that FinTracer operators can be easily combined in a variety of ways.

For example, suppose that `op1` and `op2` are two FinTracer operators which we want to combine in the "either/or" sense. This is to say, we want to answer the question "How many walks of length 1 lead from a vertex in S to y by going either through the propagation graph of `op1` or through the propagation graph of `op2`?"

We can easily create a new FinTracer operator that computes this:

```
def FTForwardSum(op, tag, output_tag):
  """
    Usage:
      Equivalent to 'FTForward', but used for operators that describe the sum
        of two other operators, rather than operators from a graph. The script
        is structured as a method of this operator.
    Inputs:
      op -- The operator.
      tag -- The tag to be propagated.
    Modified:
      None.
    Outputs:
      output_tag -- The resulting tag.
  """

  confirm_label(op, "Operator 1Dir{Signature}")
  confirm_label(tag, "FinTracer Tag{Signature}")

  op.ftcore.CloneToNewTag(tag, _temp_tag)

  op.op1.Forward(_temp_tag, output_tag)
  op.op2.ForwardInc(_temp_tag, output_tag)


def FTForwardIncSum(op, tag, output_tag):
  """
    Usage:
      Equivalent to 'FTForwardInc', but for operators that describe the sum
```

```
        of two other operators, rather than operators from a graph. The script
        is structured as a method of this operator.
    Inputs:
      op -- The operator.
      tag -- The tag to be propagated.
    Modified:
      output_tag -- The tag to which the result is added.
    Outputs:
      None.
  """

  confirm_label(op, "Operator 1Dir{Signature}")
  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(output_tag, "FinTracer Tag{Signature}")
  assert(tag.keymgr == output_tag.keymgr)

  op.ftcore.CloneToNewTag(tag, _temp_tag)

  op.op1.ForwardInc(_temp_tag, output_tag)
  op.op2.ForwardInc(_temp_tag, output_tag)


def Operator1DirSum(ftop1dir, op1, op2, op):
  """
    Usage:
      Constructor of a FinTracer operator that is the sum of two other ops.
    Inputs:
      ftop1dir -- The FinTracer one-directional operator module proxy object.
      op1, op2 -- The original two operators.
    Modified:
      None.
    Outputs:
      op -- The constructed operator, to be labelled "Operator 1Dir".
  """

  confirm_label(ftop1dir, "Operator 1Dir Module{Signature}")
  confirm_label(op1, "Operator 1Dir{Signature}")
  confirm_label(op2, "Operator 1Dir{Signature}")

  _op = object()
  _op.op1 = op1
  _op.op2 = op2
  _op.ftcore = ftop1dir.ftcore
  _op.Forward = ::FTForwardSum
  _op.ForwardInc = ::FTForwardIncSum
  op = _op
  del _op

  label(op, "Operator 1Dir{Signature}")
```

Note 4.6.1. As before, the question of whether or not `Operator1DirSum` has to be labelled depends on how the script will be used. There is no harm in certifying it, but if it is not certified it can still be used in non-privileged contexts. The script `Operator1DirSum` performs a `label` operation, so if it is run in a non-privileged mode (meaning that neither it nor any script higher

up in the execution stack was certified), this will cause `op`'s label to be weak. The outcome is that this operator `object` will be usable in non-privileged contexts, where the weakness of its label does not matter, but not in privileged contexts, that require a higher level of assurance.

Scripts that do not perform privileged commands themselves, but only call other scripts that perform privileged commands do not require privileged mode, and essentially all downstream uses that we will introduce for the summed operator can be run in this way, non-privileged.

Note 4.6.2. The scripts in the above example are intended for placement in a module called `ftop1dir`. This is one of several modules constructed in this chapter. As explained before, there is no hard and fast rule regarding what functionality belongs in a single module and which should be separated, and it can certainly be argued that a more natural solution would have been to roll all scripts in this chapter, with the possible exception of the scripts in `ftkeys`, into a single module. The separation of `ftcore` from the rest of the FinTracer functionality is meant to demonstrate how functionality can be built up over time and across multiple modules in FTIL, and how functionality that is not available in an existing module can be effectively inserted in a new module.

Given two graphs, G and H, for which we are able to compute $M_G v$ and $M_H v$, the above code computes $(M_G + M_H)v$, simply by utilising the equality $(M_G + M_H)v = M_G v + M_H v$.

A different type of composition is to compute $(M_H M_G)v$. This answers the question of how many walks lead from S to y by first taking a step on the graph G and then a second step on the graph H. We compute this by utilising the equality $(M_H M_G)v = M_H(M_G v)$. This is operator composition, rather than operator summation, and can be attained by the following code.

```
def FTForwardCompose(op, tag, output_tag):
  """
    Usage:
      Same as 'FTForwardSum', except this is for the composition of two
        operators, rather than for their sum.
    Inputs:
      op -- The compositional operator. The script is structured as its method.
      tag -- The input tag.
    Modified:
      None.
    Outputs:
      output_tag -- The resulting tag.
  """

  confirm_label(op, "Operator 1Dir{Signature}")
  confirm_label(tag, "FinTracer Tag{Signature}")

  op.op1.Forward(tag, output_tag)
  op.op2.Forward(output_tag, output_tag)


def FTForwardIncCompose(op, tag, output_tag):
  """
    Usage:
      Same as 'FTForwardIncSum', except this is for the composition of two
        operators, rather than for their sum.
    Inputs:
      op -- The operator. The script is structured as its method.
      tag -- The tag to be propagated.
    Modified:
      output_tag -- The tag to which the result is added.
    Outputs:
```

```
      None.
  """

  confirm_label(op, "Operator 1Dir{Signature}")
  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(output_tag, "FinTracer Tag{Signature}")
  assert(tag.keymgr == output_tag.keymgr)

  op.op1.Forward(tag, _temp_tag)
  op.op2.ForwardInc(_temp_tag, output_tag)


def Operator1DirCompose(ftop1dir, op1, op2, op):
  """
    Usage:
      Constructor of a FinTracer operator that is the composition of two other
        operators.
    Inputs:
      ftop1dir -- The FinTracer one-directional operator module proxy object.
                  (Unused.)
      op1, op2 -- The original two operators.
    Modified:
      None.
    Outputs:
      op -- The constructed operator, to be labelled "Operator 1Dir".
  """

  confirm_label(op1, "Operator 1Dir{Signature}")
  confirm_label(op2, "Operator 1Dir{Signature}")

  _op = object()
  _op.op1 = op1
  _op.op2 = op2
  _op.Forward = ::FTForwardCompose
  _op.ForwardInc = ::FTForwardIncCompose
  op = _op
  del _op

  label(op, "Operator 1Dir{Signature}")
```

Note that we just described three different methods to create an `object` instance with the label "`Operator 1Dir`". The idea is that even though `object` instances with this label may be distinct and derived in various ways, they provide the same type of privacy guarantees and can therefore be used in the same contexts. This allows us to combine the operator construction methods described in any way we want. For example, suppose that the scripts described above all belong to the module "`modules.ftop1dir`" and that the core FinTracer module is in "`modules.ftcore`", and suppose that we are interested in walks of length 4 that go in each step through either an edge in G or an edge in H, all we have to do is:

```
modules.ftcore.Operator1DirFromGraph(G, M_G)
modules.ftcore.Operator1DirFromGraph(H, M_H)
modules.ftop1dir.Operator1DirSum(M_G, M_H, op)
modules.ftop1dir.Operator1DirCompose(op, op, op)
modules.ftop1dir.Operator1DirCompose(op, op, op)
```

The "Operator1DirSum" places in op the operator $M_G + M_H$, the first "Compose" command replaces it with $(M_G + M_H)^2$, and the final "Operator1DirCompose" assigns to op our desired final operator, $(M_G + M_H)^4$. A simple

```
op.Forward(v, w)
```

which is the usual way to run operators and does not require any knowledge of how the operator is constructed, now computes $w = (M_G + M_H)^4 v$.

Note 4.6.3. Arguably, we are being a little over-cautious with our use of confirm_labels. There is no real need for the operators defined in ftop1dir to confirm the label of, for example, the tag parameter. Not only are no methods of tag called, the script's first operation is always a call to another script that also has a responsibility to verify the label of tag. By being more selective about when we confirm a label, operations like $(M_G+M_H)^4$ can significantly reduce the number of verifications they trigger. Because this is likely to have only a negligible cost in run-time, however, we are taking a stricter approach here, where such responsibilities are never delegated.

We remind the reader again that the semantic meaning of a label like "Operator 1Dir" is not something defined, guaranteed or policed by the language. It is something communicated by the FTIL programmer as part of documents external to the Purgles Platform and is policed by the various script certifiers. In the particular case of "Operator 1Dir", we defined this label to ascertain that every object so labelled must

1. Support a Forward method with the same interface that provides privacy-preserving tag propagation.
2. Support a ForwardInc method with the same interface that behaves identically to their Forward method, except that the target variable is added to rather than overwritten.
3. In both methods, using the input tag also as the output parameter is allowed and works as expected.

Additional attributes, both method and member, are allowed.

Regarding the last condition, note that due to the nature of parameter passing in FTIL scripts, reuse of the same variable in multiple parameter positions in a script is in general not guaranteed to function as expected. When one is overwriting one parameter value, one may in this also be overwriting any of the other parameters. Our .Forward and .ForwardInc methods, however, have been programmed so as to be immune to this problem, and we have also programmed Operator1DirSum and Operator1DirCompose such that commands such as

```
Operator1DirCompose(op, op, op)
```

Will work as expected. Readers can glance at the code for the script NewTag for an explanation of how to make code resilient to such issues. *Caveat emptor*: some scripts provided in this document have not been hardened in this way, and may fail if executed with inappropriate parameters. However, any script that labels object instances should provide such protection, in order to guarantee the validity of the object instance that they label.

4.6.2 Indirect links

The big promise of FinTracer is that it allows one to investigate paths along transaction graphs. For example, if we use a graph in which (x, y) is an edge if x has transferred at least $\$1,000$ to y, the existence of a path of length k from x to z indicates that x has effectively transferred at least $\$1,000$ to z through $k-1$ intermediaries. We can now answer the question of whether such money transfer has occurred without worrying about the identity of these intermediaries, or which RE they belong to (as long as they belong to a participating RE).

A script that can be used to answer such a question is the following:

```
def FTForwardPathExact(op, tag, output_tag):
  """
    Usage:
      Structured as a method of op, this is equivalent to 'FTForward', but
        for operators that calculate k applications of a base operator.
    Inputs:
      op -- The operator
      tag -- The input tag value
    Modified:
      None.
    Outputs:
      output_tag -- The resulting tag value.
  """

  confirm_label(op, "Operator 1Dir{Signature}")
  confirm_label(tag, "FinTracer Tag{Signature}")

  # Our first step is to clone tag into output_tag.
  # If output_tag and tag are the same variable, this does nothing.
  # If output_tag and op are the same variable, the operation will fail because
  # one cannot reassign an object while inside one of its own methods.
  # For these reasons, it is not necessary here to use temporary objects.
  op.ftcore.CloneToNewTag(tag, output_tag)

  # Note that op.k is shared at the scope of op, which includes the coordinator.
  for _i in range(op.k):
    op.op1.Forward(output_tag, output_tag)


def FTForwardIncPathExact(op, tag, output_tag):
  """
    Usage:
      Structured as a method of op, this is equivalent to 'FTForwardInc', but
        for operators that calculate k applications of a base operator.
    Inputs:
      op -- The operator
      tag -- The input tag value
    Modified:
      output_tag -- The tag to which the result is added.
    Outputs:
      None.
  """

  confirm_label(op, "Operator 1Dir{Signature}")
  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(output_tag, "FinTracer Tag{Signature}")
  assert(tag.keymgr == output_tag.keymgr)

  op.ftcore.CloneToNewTag(tag, _temp_tag)

  # Note that op.k is shared at the scope of op, which includes the coordinator.
  for _i in range(op.k):
    op.op1.Forward(_temp_tag, _temp_tag)
```

```
  output_tag.Add(_temp_tag)


def Operator1DirPathExact(ftop1dir, op1, k, op):
  """
    Usage:
      Constructor of a FinTracer operator that is the k-fold composition of
        another operator (i.e., performs k steps of its base operator).
    Inputs:
      ftop1dir -- The FinTracer one-directional operator module proxy object.
      op1 -- The base operator.
      k -- The number of steps.
    Modified:
      None.
    Outputs:
      op -- The constructed operator, to be labelled "Operator 1Dir".
  """

  confirm_label(ftop1dir, "Operator 1Dir Module{Signature}")
  confirm_label(op1, "Operator 1Dir{Signature}")

  on(CoordinatorID):
    # The first of the assert statements below is, in principle, superfluous.
    # If it fails, the other two asserts will surely also fail. It is here
    # mainly because having it presumably leads to more informative error
    # messages in the case of an abort.
    assert(CoordinatorID in k.metadata.scope)
    assert(k.metadata.type == int)
    assert(k >= 0)

  _op = object()
  _op.op1 = op1
  _op.k = broadcast(k)
  _op.ftcore = ftop1dir.ftcore
  _op.Forward = ::FTForwardPathExact
  _op.ForwardInc = ::FTForwardIncPathExact
  op = _op
  del _op

  label(op, "Operator 1Dir{Signature}")
```

Note that this is significantly more code than what would have been necessary if all we needed was to find the paths of length k. To do that, a simple script with the code

```
output_tag = tag
for _i in range(k):
  op1.Forward(output_tag, output_tag)
```

would have sufficed. The added lines of FTIL code are used to wrap this script inside an `object` instance with the label "`Operator 1Dir`", allowing it to be used from this point on as a building-block in all further construction.

Finding paths of length k is nice, but for intel purposes one rarely needs to be looking for paths of some exact, specified length. Path lengths are best kept as thresholds. The code below propagates a tag through paths of any length between 1 and the chosen k.

```
def FTForwardPathAtMost(op, tag, output_tag):
  """
    Usage:
      Structured as a method of op, this is equivalent to 'FTForward', but for
        operators that calculate between-1-and-k applications of a base
        operator.
    Inputs:
      op -- The operator
      tag -- The input tag value
    Modified:
      None.
    Outputs:
      output_tag -- The resulting tag value.
  """

  confirm_label(op, "Operator 1Dir{Signature}")
  confirm_label(tag, "FinTracer Tag{Signature}")

  op.ftcore.CloneToNewTag(tag, _temp_tag)

  op.ftcore.NewTag(tag.keymgr, output_tag)

  # No need to
  # confirm_label(output_tag, "FinTracer Tag{Signature}")
  # because it was already pre-confirmed in NewTag.

  # Note that op.k is shared at the scope of op, which includes the coordinator.
  for _i in range(op.k):
    op.op1.Forward(_temp_tag, _temp_tag)
    output_tag.Add(_temp_tag)


def FTForwardIncPathAtMost(op, tag, output_tag):
  """
    Usage:
      Structured as a method of op, this is equivalent to 'FTForwardInc', but
        for operators that calculate up-to-k applications of a base operator.
    Inputs:
      op -- The operator
      tag -- The input tag value
    Modified:
      output_tag -- The tag to which the result is added.
    Outputs:
      None.
  """

  confirm_label(op, "Operator 1Dir{Signature}")
  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(output_tag, "FinTracer Tag{Signature}")
  assert(tag.keymgr == output_tag.keymgr)

  op.ftcore.CloneToNewTag(tag, _temp_tag)

  # Note that op.k is shared at the scope of op, which includes the coordinator.
```

```
  for _i in range(op.k):
    op.op1.Forward(_temp_tag, _temp_tag)
    output_tag.Add(_temp_tag)


def Operator1DirPathAtMost(ftop1dir, op1, k, op):
  """
    Usage:
      Constructor of a FinTracer operator that is the sum of all j-fold
        compositions of another operator, op, for all j in 1 <= j <= k.
        Good for searching paths of "up to length k".
    Inputs:
      ftop1dir -- The FinTracer one-directional operator module proxy object.
      op1 -- The base operator.
      k -- The maximum number of steps.
    Modified:
      None.
    Outputs:
      op -- The constructed operator, to be labelled "Operator 1Dir".
  """

  confirm_label(ftop1dir, "Operator 1Dir Module{Signature}")
  confirm_label(op1, "Operator 1Dir{Signature}")

  on(CoordinatorID):
    # The first of the assert statements below is, in principle, superfluous.
    # If it fails, the other two asserts will surely also fail. It is here
    # mainly because having it presumably leads to more informative error
    # messages in the case of an abort.
    assert(CoordinatorID in k.metadata.scope)
    assert(k.metadata.type == int)
    assert(k>0)

  _op = object()
  _op.op1 = op1
  _op.k = broadcast(k)
  _op.ftcore = ftop1dir.ftcore
  _op.Forward = ::FTForwardPathAtMost
  _op.ForwardInc = ::FTForwardIncPathAtMost
  op = _op
  del _op

  label(op, "Operator 1Dir{Signature}")
```

4.6.3 The FinTracer one-directional operator module

Let us now pack the functionality of one-directional operators into a module. Suppose we call this module "`file://ftop1dir.ftil`".

The module file will begin with the lines

```
FTIL1.0
module ftop1dir
using ftcore
```

will include the scripts above, and will conclude with an `__init` script as follows.

```
def __init(ftop1dir, modules):
  """
    Usage:
      __init script for the core FinTracer module.
    Inputs:
      modules -- The modules object. We use modules.ftcore
    Modified:
      None.
    Outputs:
      ftop1dir -- The new module proxy object generated, to be labelled
                  "Operator 1Dir Module".
  """

  confirm_label(modules.ftcore, "FinTracer Core Module{Signature}")

  ftop1dir = object()

  ftop1dir.ftcore = modules.ftcore

  ftop1dir.PathAtMost = ::Operator1DirPathAtMost
  ftop1dir.PathExact = ::Operator1DirPathExact
  ftop1dir.Compose = ::Operator1DirCompose
  ftop1dir.Sum = ::Operator1DirSum

  label(ftop1dir, "Operator 1Dir Module{Signature}")
```

4.6.4 A note on versioning

The definition of FTIL allows for the idea that module files will receive updates as time goes by. As a result, new FTIL module files may appear, which will also want to label their module proxy object "Operator 1Dir Module", for example.

The way that FTIL handles versioning (not as an extra feature but rather as a result of the way that `objects` behave in FTIL at large) is that any references to the module proxy object, or to its underlying module, do not get updated if a new module is imported, even if the new module has the same name and overrides the variable holding the original module proxy object.

An extreme case of this is in module dependencies. Above, we see the module `ftop1dir` having a `using` dependency to `ftcore`. If `ftcore` is updated, The `ftop1dir` module proxy object will still maintain a reference to the old `ftcore`, and any `object` that was generated by the `ftop1dir` module proxy object that utilises `ftcore` functionality will continue to do so by calling the old module.

If a variable created by the new `ftcore` is sent to a script in the existing `ftop1dir`, that variable can, in theory, carry a reference to the new `ftcore` module (though `ftcore objects` in this chapter don't), which would make the two modules work side-by-side.

If one wishes to remove the old `ftcore` from the system altogether, the only way to do so is to delete all operator `objects` created by `ftop1dir` that may still carry references to the old module, and then re-import `ftop1dir`.

4.7 Querying an explicit list

So far in our description, we showed how REs are able to compute various functions of our choosing in a privacy-preserving manner, but we have not shown how AUSTRAC can gain access to any results from these computations.

We will return to the topic of function computation in later sections, and show many far more powerful ways to use FinTracer's tag propagation mechanisms, but before doing so, we turn to provide some means by which AUSTRAC can actually retrieve computation results. This, too, is a topic we will expand on in later sections, introducing more powerful methods for querying.

Though this arrangement of sections necessarily means that our examples hop around between topics more than is strictly necessary, its advantage is in introducing here, quite early in the examples chapter, enough machinery in order to effectively apply FinTracer for intel purposes, even if this is only the simplest and most basic application type of the algorithm.

The final piece still missing from the puzzle before the system is of any practical value, the reading of tag values which this section will now fill in, is different in character to the computation of functions which the previous section dealt with, in that (with a few notable exceptions) during computation no party receives any information whatsoever, making the plugging of information leaks a fairly straightforward task. Here, by design, parties do receive information. AUSTRAC receives tag values, and the REs receive the information of which accounts are being queried. (We deal in a separate section with the possibility of oblivious querying, in which REs do not receive any information at all. In our present case, it is by design that REs do receive this extra information.)

The important issues in implementing a tag querying procedure are the following.

1. We do not want to provide AUSTRAC with any information other than whether these accounts have tag values that are zero or nonzero. For example, AUSTRAC should not be able to determine for specific accounts whether or not any walks of a specified type lead to them;

2. We do not wish to provide any RE with any information other than which account numbers are the ones being uncovered to AUSTRAC.

3. Given that this process provides information about individual accounts, we wish to limit the number of accounts being queried; and

4. If the list of accounts was generated by means of a description, we only want the REs to communicate to AUSTRAC the account numbers for those accounts for which the tag value is nonzero. However, in this case we allow the REs to learn which accounts have nonzero tags.

Note 4.7.1. This list is illustrative. In reality, it will almost inevitably be the case that the platform will be programmed so as to allow REs to know which tags were nonzero even when the list of accounts is not generated by means of a description. Supporting this requires a minor tweak in `RetrieveTagFromList`, below. The reader is invited to add the missing code, as an exercise.

The simplest scenario is one where AUSTRAC has an explicit list of accounts to be queried. For example, the user might have generated such a list by a function call

```
on(CoordinatorID) AUSTRAClist = Read<set<AccountAddr>>("Accounts to query on")
```

Here, it is very clear what information should be transmitted to each of the parties. The following is a script that does precisely this, divulging no additional information.

```
def RetrieveTagFromList(ftretrieve, AUSTRAClist, tag, policy, positives):
  """
    Usage:
      Given an explicit list of accounts provided by AUSTRAC, this script
        returns the subset of the accounts whose tag value is nonzero.
        The script provides safeguards, by means of a policy object, regarding
        how many accounts can be queried in this way.
    Inputs:
```

```
    ftretrieve -- The module proxy object.
    AUSTRAClist -- The list of accounts, given as a set<AccountAddr>.
    tag -- The tag to be queried
    policy -- The policy object.
  Modified:
    None.
  Outputs:
    positives -- a local transmitter<set<AccountID>> object storing the
                 result.
"""

confirm_label(ftretrieve, "FinTracer Retrieval Module{Signature}")
confirm_label(tag, "FinTracer Tag{Signature}")
confirm_label(policy, "Policy For RetrieveTagFromList{Signature}")

_peer_ids = setminus(scope(), nodeset(CoordinatorID))

on(CoordinatorID):
  limit(AUSTRAClist.size() <= policy.upper_bound_on_queries,
          "List is too large to be queried.")
  if AUSTRAClist.size() > policy.confirm_threshold_on_queries:
    confirm("List being queried has " + str(AUSTRAClist.size()) +
      " elements. Are you sure you wish to proceed?")

# All prerequisite checks on AUSTRAClist are done as part of
# "AccountsFromList". In particular, it is ascertained there that the list is
# viewable by the coordinator node and only by it.
ftretrieve.ftcore.AccountsFromList(AUSTRAClist, _accounts)

_returnedtag = transmitter<dict<AccountID, ElGamalCipher>>()
on(_peer_ids):
  _returnedtag[CoordinatorID] =
    dict{tag.map.lookup(i, tag.keymgr.zero) over i = _accounts.iter}
  del _accounts

  tag.keymgr.ElGamalRefresh(_returnedtag[CoordinatorID])

  # Using a built-in sanitisation command.
  Sanitise(_returnedtag[CoordinatorID])

transmit(_returnedtag)
on(CoordinatorID):
  positives = transmitter<set<AccountID>>()
  for _n in _peer_ids:
    _decrypted = ElGamalDecrypt(_returnedtag[_n][@], tag.keymgr.priv_key)
    # Above line may also be available as
    #        "_decrypted = ElGamalDecrypt(_returnedtag[_n], keymgr.priv_key)"
    _nomatch = Ed25519(0)
    positives[_n] = set<AccountID>{acct over acct = _decrypted.iter
                                      with _decrypted[acct] != _nomatch}
```

In practice, this script belongs to `ftcore`. However, as we are constructing the code bit by bit and have already specified what belongs in `ftcore`, we are presenting tag retrieval here as its own module, `ftretrieve`. As we shall see, FTIL handles this situation gracefully.

The script above uses `transmit`, so will need to run in privileged mode. As a result, its `confirm_label` commands will require a strong label. In particular, the `policy object` will require a strong label.

Here is (partial) code that can be used to generate the "`policy`" object instance. Presumably, this script will be run from the command line, and will be certified individually. This is the FTIL way of having each RE ratify the choice of thresholds used.

```
def NewPolicyForRetrieveTagFromList(ftretrieve, policy):
  """
    Usage:
      Constructor of a policy object for RetrieveTagFromList.
    Inputs:
      ftretrieve -- The module proxy object. (Unused.)
    Modified:
      None.
    Outputs:
      policy -- The constructed object, to be labelled
                  "Policy For RetrieveTagFromList".
  """

  policy = object()

  on(CoordinatorID):
    policy.upper_bound_on_queries = ... # Some number here
    policy.confirm_threshold_on_queries = ... # Some number here

  label(policy, "Policy For RetrieveTagFromList{Signature}")
```

Some comments regarding the above code:

1. The return value, given in "`positives`", is constructed as an entity of type "`transmitter<set<AccountID>>`". This choice of type splits the list according to which RE the accounts belong to. It would have been trivial to create a joined result set by taking the union of all parts of "`positives`", although we recommend in this case to store for each element its full `AccountAddr` data, rather than just its `AccountID` (as was done with `AUSTRAClist`). Our choice of return type is both more economical and more amenable to downstream processing.

2. Given that any information returned on any account is an intrusion into the privacy of the account owner (however justified), the script has safeguards to prevent overly-large lists from being queried at once. This prevents accidental querying of large populations and makes intentional querying of large populations both cumbersome and easily detectable in logs. The script provides two levels of size thresholds for querying: one requiring explicit user confirmation, and one beyond which querying is not allowed at all. (Although, of course, a tenacious user can circumvent this safeguard by splitting their lists.)

3. The two thresholds used can be given by explicit literals in the code, but in this case we preferred to implement them as variables ("`policy.confirm_threshold_on_queries`" and "`policy.upper_bound_on_queries`"). Because these variables are attributes of a labelled `object` instance, they are constants. If "`NewPolicyForRetrieveTagFromList()`" is the only script that can generate an `object` instance with this label, the user cannot circumvent the policy by trying to overwrite either the `policy` variable itself or its attributes. This method allows the programmer to manage policy parameters in a well-organised way, and without cluttering the global namespace. Even better would have been to separate the policy parameters to a different module entirely. This would have allowed AUSTRAC and the REs

to occasionally revise the values in these policies more easily: only the policy module would have required re-confirmation in this case, rather than the entire retrieval code.

4. Previous versions of this code did not "`Refresh()`" before performing a "`Sanitise()`". See Section 10.2 for why refreshing a tag even when sanitising it is nevertheless important.

5. In the code we demonstrate two potential ways to invoke the "`ElGamalDecrypt()`" function. In one case, the function acts on a single `ElGamalCipher` and we invoke it via a mass operation so as to act on an entire dictionary. In the other, we assume that `ElGamalDecrypt` can also act directly on a dictionary `object` to decrypt all the dictionary's values. In practice, the question of whether we will want to implement an `ElGamalDecrypt()` function on a dictionary will probably come down to how much efficiency boost we can get in this, compared to decrypting each ciphertext individually.

6. Decrypting an `ElGamalCipher` results in an `Ed25519`. This is not the same as an `Ed25519Int` variable, which acts as a type of integer. By contrast, the main ways of checking the value of an `Ed25519` are (1) comparing it to a desired value, and (2) looking its value up in a `hashmap`. Here, we chose the former method, as we merely need to check whether the value is zero or nonzero.

7. To compare an `Ed25519` value to zero, we prepare the comparison entity, "`_nomatch`", initialised by the value "`Ed25519(0)`". In a mature language, one would not need "`_nomatch`" at all. Rather, one would simply write "`Ed25519(0)`" in place of "`_nomatch`" where it is used, and assume that somewhere along the way a compiler/optimiser will recognise this to be a constant and will not attempt to run the conversion operations again and again each time the comparison against zero is invoked. FTIL is not a mature language and we shouldn't expect it to have a sophisticated optimiser. (In fact, sophisticated optimisation may open the door to various cryptographic attacks. FTIL was specifically designed to allow the user to know exactly what is executed, in order to be able to certify each protocol run.) By creating the "`_nomatch`" entity ourselves, we explicitly tell the executor to store the necessary constant.

Perhaps the most important comment to make about the code for "`RetrieveTagFromList`" is about what it *doesn't* contain. Specifically, the code does not contain any limitation regarding which accounts and which tags AUSTRAC can query. The Purgles Platform's wider governance structures are important in managing this aspect. In-software, we merely enable throttling of the rate at which AUSTRAC can query the system.

Throttling aside, however, AUSTRAC can query the value of any tag at any account. This freedom will remain also in later sections of this chapter.

We conclude this section by providing the rest of the `ftretrieve` module code.

```
FTIL1.0
module ftretrieve
using ftcore

def __init(ftretrieve, modules):
  """
    Usage:
      __init script for the FinTracer retrieval module.
    Inputs:
      modules -- The modules object. We use modules.ftcore
    Modified:
      None.
    Outputs:
      ftretrieve -- The new module proxy object generated, to be labelled
                    "FinTracer Retrieval Module".
  """
```

```
confirm_label(modules.ftcore, "FinTracer Core Module{Signature}")

ftretrieve = object()

ftretrieve.ftcore = modules.ftcore

ftretrieve.TagFromList = ::RetrieveTagFromList
ftretrieve.NewPolicy = ::NewPolicyForRetrieveTagFromList

label(ftretrieve, "FinTracer Retrieval Module{Signature}")
```

In actual practice, we do not expect `ftretrieve` to be in use at all. The querying methods of Section 4.9 are more generic, and can do everything that `ftretrieve` does. They also have the following advantages:

1. The REs not only learn the identity of the tag-positive accounts, but do so before AUSTRAC, this being in some cases a design criterion; and
2. They feature `policy` parameters that are peer-side variables, rather than coordinator-side. This allows each RE to set their own thresholds, applicable to only their portion of the retrieved data.

In light of this, REs may choose not to certify the code in `ftretrieve` at all.

The `ftretrieve` code does, however, have the advantage of being a more lightweight computation than the alternatives presented in Section 4.9.

4.8 Differential privacy

We now turn to the question of how to query a tag value when AUSTRAC does not have an explicit list of accounts at its disposal, but rather merely a description, which the REs can, in turn, translate into a set of accounts, S.

What AUSTRAC would like to discover is as before: of the accounts x in the set of accounts S, which are the ones satisfying $v[\tilde{x}] \neq 0$? (Or, alternatively, which are the ones satisfying $v[\tilde{x}] = 0$?) The answer to this question should be visible both to AUSTRAC and to the REs at the end of querying (each RE only receiving the information regarding the accounts that they, themselves, manage), but what we do not want AUSTRAC to find out is which other account numbers the REs may have, whether part of the set S or not, and what we do not want the REs to find out is what tag values accounts outside of set S have.[1]

Note that in this model the REs themselves know S, so can infer from the accounts that have nonzero tag values which have zero tag values, and vice versa.

The general method in which queries based on descriptions are resolved is that the REs create a random permutation, Π, over S, and then send the tag values to AUSTRAC in the random order implied by Π, without any information regarding which value relates to which account. AUSTRAC replies with the list of positions that are of interest to it (say, those positions corresponding to positive tags) and the account numbers corresponding to these positions are revealed. This is, of course, on the assumption that the list to be revealed is not too large. There are policy limits regarding the number of accounts that can be queried.

In this naive description, there are two pieces of information that AUSTRAC receives that are beyond what it strictly must obtain. These are

[1] In this model, a malicious RE actor can almost inevitably discover some tag values outside of set S simply by acting as if the desired set was some S' that is slightly larger and includes the extra accounts that the RE wishes to inspect. We did not implement more involved protocols that prevent this. An alternative, however, is to use the oblivious querying protocols of Section 4.13, where the REs do not gain information about tag values, and so the problem never arises to begin with.

1. The number of nonzero tag values in S, and
2. The number of zero tag values in S.

(In a successfully-executing query, AUSTRAC naturally learns one of these two, and only the other is superfluous information. In a query that fails due to exceeding thresholds, however, AUSTRAC shouldn't be exposed to either.)

This is not ideal for a privacy-preserving algorithm, because, for example, if S is small and AUSTRAC has some information regarding some accounts in it, these totals may provide it with additional information regarding the rest of the accounts.

Unfortunately, in our algorithmic setup it is almost impossible to hide from AUSTRAC the approximate sizes of the sets involved, and all our algorithms do leak this information. The solutions remain viable, however, if we refine our goals somewhat: instead of trying to hide all information relating to the set sizes, we want to make it difficult for AUSTRAC to be able to tell if any single account was subtracted from or added to S. This is a condition known as *differential privacy*. It is useful because legal requirements are usually to prevent a party from discovering information about a specific (personally-identifiable) account, rather than to prevent the discovery of general statistical information.

Differential privacy is tailored specifically for such a situation. It deals with the problem by adding to the list to be permuted an additional number, z_0, of zero-valued tags and an additional number, z_1, of nonzero-valued tags, where z_0 and z_1 are random numbers, chosen according to a distribution that makes it hard for AUSTRAC to determine the exact number prior to the injection of noise. Ideally, z_0 and z_1 are much smaller than the total size of the data to be communicated, so the extra communication due to differential privacy is negligible.

Before going into the technicalities of tag querying, in this section we delve into differential privacy, and how it is managed and preserved by our algorithms.

4.8.1 (ϵ, δ)-differential privacy

The method we will use is (ϵ, δ)-differential privacy. The mathematics of this method are described in detail in Section 5.4, and its important limitations are discussed in Section 10.1. For the moment, however, only the following details are important.

The idea of (ϵ, δ)-differential privacy is that noise is generated to mask a value from an opponent such that other than with probability δ, the distribution of values seen by the opponent is essentially the same, with or without the single change in underlying condition that the noise is meant to protect against. Here, "essentially the same" means that the probability of receiving any outcome is the same with and without, up to a multiplicative factor of e^{ϵ}, where e is Euler's constant. In our case, the change we are trying to conceal is the addition or removal of a single account from S.

Let $D_{\epsilon,\delta}$ be a distribution such that for any integer N, if the opponent sees $N + z$, where z is a nonnegative integer distributed $D_{\epsilon,\delta}$, then the value of N is (ϵ, δ)-differentially private against a change of N by 1. Then we can allot two random values, z_0 and z_1, independently from this distribution, and add z_0 fake accounts with zero tags and z_1 fake accounts with non-zero tags to make our S (ϵ, δ)-differentially private against our chosen changes.

It turns out that the noise distribution $D_{\epsilon,\delta}$ with minimal expectation that satisfies these conditions, subject to the added constraint that the ϵ condition is satisfied for every positive z, is one where there is some nonnegative integer t and some real y, such that for z values in the range $0 \leq z < t$, the probability of attaining this value in $D_{\epsilon,\delta}$ is $\delta e^{\epsilon z}$, whereas for $z \geq t$ it is $y e^{(t-z)\epsilon}$.

If $t > 0$, $\delta e^{(t-2)\epsilon} < y \leq e^{t\epsilon}$ must hold.

The following is code that computes t and y based on ϵ and δ.

```
def DifferentialPrivacyParameters(epsilon, delta, t, y):
  """
    Usage:
      DifferentialPrivacyParameters calculates the parameters of a
```

```
        minimal-expectation distribution that attains
        (epsilon, delta)-differential privacy against a change of magnitude 1.
    Inputs:
      epsilon, delta -- The differential privacy parameters.
    Modified:
      None.
    Outputs:
      t -- Number of values before reaching y.
      y -- Height of the peak of the distribution.
  """

  t = max(0, \
        ceil(log((((1 - exp(-epsilon)) ** 2 - delta * (1 - exp(-epsilon))) \
          / (delta * (1 - exp(-2 * epsilon)))) + 1) / epsilon))

  y = 1 + (delta - 1) * exp(-epsilon) - delta * exp((t - 1) * epsilon)
```

These parameters lead to a distribution with the following minimal expectation.

```
def DifferentialPrivacyExpectation(epsilon, delta, t, y, E):
  """
    Usage:
      DifferentialPrivacyExpectation computes the expectation of an
        (epsilon, delta)-differentially private noise distribution with
        parameters t and y.
    Inputs:
      epsilon, delta -- The differential privacy parameters.
      t -- Number of values before reaching y.
      y -- Height of the peak of the distribution.
    Modified:
      None.
    Outputs:
      E -- Expectation of the distribution.
  """

  E = y * ((t / (1 - exp(-epsilon)))
              + exp(-epsilon) / (1 - exp(-epsilon)) ** 2) \
      + ((t - 1) * delta * exp((t - 1) * epsilon)) / (1 - exp(-epsilon)) \
      + delta * (exp(-epsilon) - exp((t - 2) * epsilon)) \
        / (1 - exp(-epsilon)) ** 2
```

The case $t = 0$ is in many ways different to the case $t > 0$, but in both scripts above the same equations hold for both cases. Had that not been the situation, a ternary operator could have been used to compute the function properly. An example of case-based computation can be found in the next script, which allots a random element from the distribution. The cases here are not $t = 0$ vs $t > 0$, but rather whether the element allotted is from the ascending or the descending portion of the distribution.

```
def AllotDifferentialPrivacyNoise(epsilon, delta, t, y, z):
  """
    Usage:
      AllotDifferentialPrivacyNoise generates a random value from the
        differential privacy noise distribution with parameters
        epsilon, delta, t and y.
    Inputs:
```

```
      epsilon, delta -- The differential privacy parameters.
      t -- Number of values before reaching y.
      y -- Height of the peak of the distribution.
    Modified:
      None.
    Outputs:
      z -- Random value generated
  """

  _r = RandomFloat(1 - (1 - exp(-epsilon)) / y, 1)

  _temp = (_r > 0) ? float(_r : 1 + (_r * y / (delta * exp((t - 1) * epsilon))))

  z = t + floor(-sgn(_r) * log(_temp)) / epsilon
```

Note that in FTIL both sides of a ternary operator get evaluated. For this reason, it is not possible to compute directly something like

```
(_r > 0) ? float(log(_r) : log(-_r))
```

because the evaluation will fail on at least one side. We have structured our code, above, to avoid this problem.

4.8.2 Our differential privacy paradigm

We use for differential privacy the following paradigm whose mathematical justification is discussed in Section 5.4.

It should be noted, however, that such differential privacy is not a perfect mechanism, and shouldn't be used as a sole measure of privacy preservation. A discussion of the method's limitations and how to manage them is given in Section 10.1.

In our paradigm, each tag is associated with *sources* being "where the information came from".

Tracking of sources begins when defining account sets, and continues when these sets are used to initialise or modify tags. In principle, tags and sets initialised as empty have no sources, but when a set is defined by a description (whether that description is resolved by a human operator or by a SQL database) that description forms its own, new source. Only account sets defined explicitly on the coordinator side do not contribute to the source calculus. This is because their contents are already known to the coordinator.

Any calculation from multiple operands takes all sources from all operands.

Our differential privacy calculations now relate not to the tags being manipulated, but rather to the information sources being exposed through these manipulations, these being all sources contributing to each manipulated tag.

Each source keeps track of its *privacy budget used*. This budget comes in the form of two numbers, ϵ and δ, both of which are initially zero.

The maximal ϵ and δ any source can reach are specified as part of a policy `object`.

Whenever we perform an operation that has a cost in differential privacy, we ask the user to specify how much privacy budget is to be used for this operation. If the answer is (ϵ', δ'), we first make sure that the sources of all query `objects` participating in the operation have enough budget left. We then perform the operation, padding both the number of zero tags and the number of non-zero tags by an amount of noise individually guaranteeing $(\epsilon'/m, \delta'/m)$-differential privacy, where m is the maximal number of values relating to the same account that can be exposed to AUSTRAC as a result of the operation.

The operation is then performed as usual, and the budget used by each source is increased by (ϵ', δ').

As a courtesy to the FTIL user, the software monitors the expected amount of noise items that are to be inserted due to a given operation, and throttles the operation if the inserted noise is unreasonably high.

4.8.3 The policy object

The following (partial) code implements the differential privacy policy `object`. Unlike in the previous example of a policy `object`, this policy `object` is not just a passive entity, delivering some thresholds. Rather, in this case the policy `object` is also the `object` in charge of creating the differential privacy noise. This design choice is meant to make the policy parameters not just be abstract numbers, but rather have concrete meaning in terms of how they impact noise generation.

Specifically, our design will include an `object` in charge of the noise distribution parameters and the generation of noise instances, and the policy `object` will know how to spawn such distribution `object`s whenever required.

Here is the code for the distribution-managing `object`.

```
def DistributionNoise(dist, z):
  """
    Usage:
      DistributionNoise allots a value from a noise distribution.
    Inputs:
      dist -- The distribution object.
    Modified:
      None.
    Outputs:
      z -- The allotted value.
  """

  confirm_label(dist, "Differential Privacy Distribution{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):
    AllotDifferentialPrivacyNoise(dist.epsilon, dist.delta, dist.t, dist.y, z)


def DistributionNoisePair(dist, z0, z1):
  """
    Usage:
      DistributionNoisePair allots two independent values from a
        noise distribution
    Inputs:
      dist -- The distribution object.
    Modified:
      None.
    Outputs:
      z0 -- Number of fake zero-valued ciphers to add to query object.
      z1 -- Number of fake nonzero-valued ciphers to add to query object.
  """

  confirm_label(dist, "Differential Privacy Distribution{Signature}")

  dist.Noise(z0)
  dist.Noise(z1)


def ExpectedNoise(dist, expectation):
  """
```

```
    Usage:
      ExpectedNoise returns the expected amount of elements to be added as
        noise for differential privacy purposes. This is the amount for zero
        and nonzero elements combined.
    Inputs:
      dist -- The distribution object.
    Modified:
      None.
    Outputs:
      expectation -- The expected amount.
  """

  confirm_label(dist, "Differential Privacy Distribution{Signature}")

  expectation = 2 * dist.E
```

The user will not be able to create this type of distribution-managing `object` directly, without further context. Rather, it is created by a differential privacy policy `object`. This allows, for example, for better parameter checking at creation time.

```
def NewNoiseDistribution(dp_policy, epsilon, delta, dist):
  """
    Usage:
      NewNoiseDistribution creates a noise distribution object.
    Inputs:
      dp_policy -- The differential privacy policy.
      epsilon, delta -- The budget used (assuming each account is used only
                        once).
    Modified:
      None.
    Outputs:
      dist -- The created distribution, to be labelled
              "Differential Privacy Distribution".
  """

  confirm_label(dp_policy, "Differential Privacy Policy{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):
    # We check for epsilon and delta not just lower bounds but also upper
    # bounds. This is not really necessary to do, but provides errors earlier,
    # if the user inputs numbers that are completely off the scale.
    assert(epsilon.metadata.type == float)
    assert(epsilon > 0 and epsilon <= dp_policy.epsilon)
    assert(delta.metadata.type == float)
    assert(delta > 0 and delta <= dp_policy.delta)

  _dist = object()

  on(_peer_ids):
    _dist.epsilon = epsilon / 2
    _dist.delta = delta / 2
```

```
    DifferentialPrivacyParameters(_dist.epsilon, _dist.delta, _dist.t, _dist.y)
    DifferentialPrivacyExpectation(_dist.epsilon, _dist.delta, _dist.t, _dist.y,
                                   _dist.E)

    limit(2 * _dist.E <= dp_policy.max_expectation,
            "Expected noise amount too high. More noise budget is needed.")

  _dist.Noise = ::DistributionNoise
  _dist.NoisePair = ::DistributionNoisePair
  _dist.Expectation = ::ExpectedNoise

  dist = _dist
  del _dist

  label(dist, "Differential Privacy Distribution{Signature}")
```

In practice, this is a contrived example, because the distribution `object` does not maintain a reference to the original policy for later use throughout its lifespan, and the "better parameter checking" is not really needed: the parameter problem will cause the program to abort the first time invalid parameters are utilised anyway, even without this extra check at initialisation time.

The example is given here mainly for didactic reasons, to demonstrate the ways in which FTIL `object` management can be done.

Below is the code that generates a new differential privacy `object`.

```
def NewDPPolicy(ftquery, dp_policy):
  """
    Usage:
      NewDPPolicy creates a new policy object based on given differential
        privacy parameters.
    Inputs:
      ftquery -- The FinTracer Query module proxy object. (Unused.)
    Modified:
      None.
    Outputs:
      dp_policy -- The created policy object, to be labelled
                "Differential Privacy Policy"
  """

  dp_policy = object()

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):
    dp_policy.epsilon = ... # The epsilon budget usage allowed.
    dp_policy.delta = ... # The delta budget usage allowed.

    dp_policy.max_expectation = ... # Cannot add noise beyond this amount.

  dp_policy.NewDistribution = ::NewNoiseDistribution

  label(dp_policy, "Differential Privacy Policy{Signature}")
```

In the partial code above, the values of the differential policy parameters `dp_policy.epsilon`, `dp_policy.delta` and `dp_policy.max_expectation` are meant to be initialised from literals, meaning that they will be shared values, equal on all peer nodes. However, this is not the only

way to initialise these, and our code throughout this chapter is designed to also support an alternate implementation in which each peer has its own, distinct value for these parameters. This is meant to support a situation in which each RE arrives independently to a conclusion regarding what value of these parameters they are comfortable with. The policy is, in any case, applied separately for each peer, to the portion of the data that is returned from that peer.

4.9 Querying based on a description

We can now turn to implementing the querying methodology, as described. The idea, again, is to pad the result with fake values, send the values without their associated account numbers in a random order to AUSTRAC, and have AUSTRAC respond with the relevant positions in the random permutation.

We will first program a script that determines the final result, which will be an internal module script, not associated with any `object` and therefore not callable from outside the module, and then describe two scripts that utilise it to implement the full end-to-end query, which will differ based on whether only AUSTRAC should keep the result or whether a copy should also be retained on the RE end.

4.9.1 Filtering

Note that typically we are not interested in retrieving every positive tag value, but only those tag values that are within our target account set, S, which is defined by a description. Our querying code will not need to know about this process, however. It assumes that it is provided with a tag whose keys are exactly S.

The set S can be created in various ways. For example by

```
ftcore.AccountsFromSQL(sql_query, S)
```

or

```
ftcore.AccountsFromDescription(description_string, S)
```

Once it exists, reducing the tag to only the set S can be done using

```
tag.Reduce(S)
```

4.9.2 The retrieval policy object

The following generates a policy `object` for retrieval. It incorporates all information needed for managing differential privacy (via a differential privacy policy `object`), as well as some additional system constraints regarding the size of result sets that can be returned.

```
def NewRetrievalPolicy(ftquery, policy):
  """
    Usage:
      NewRetrievalPolicy creates a new policy object for value retrieval.
    Inputs:
      ftquery -- The FinTracer Query module proxy object.
    Modified:
      None.
    Outputs:
      policy -- The created policy object, to be labelled
                "Retrieval Policy"
  """
```

```
confirm_label(ftquery, "FinTracer Query Module{Signature}")

_peer_ids = setminus(scope(), nodeset(CoordinatorID))

policy = object()

ftquery.NewDPPolicy(policy.dp_policy)

on(CoordinatorID):
  # The following limits are overall query return size limits, and are
  # tested at the coordinator

  # Cannot query if expected num beyond threshold.
  policy.max_approx_query = ...

  # Requires user confirmation.
  policy.softmax_approx_query = ...

on(_peer_ids):
  # The following limit relates to per-peer query return sizes, and is
  # tested by each peer individually. It can be different for each peer.

  policy.max_exact_query = ... # Cannot return beyond this many results.

label(policy, "Retrieval Policy{Signature}")
```

We will use this policy `object` for all query retrieval policy management relating to querying over account sets that have been defined by descriptions rather than by account lists.

4.9.3 Creating a query result object

There are multiple way in which we may want to query a tag, and multiple ways to use the result of such a query. To help us manage this complexity, our scripts for utilising the query result will take as a parameter an `object` that specifies how the tag should be queried.

From a privacy standpoint, AUSTRAC, in this design, has the right to request any set that it deems necessary in order to uphold the AML/CTF act, whereas the REs, in this design, have a right to know what information they have reported to AUSTRAC. This leads to a curious situation in which peer nodes should see the final results of computations, but should not be privy to intermediate results.

Given that we have separated the calculation of the results from the retrieval by AUSTRAC of the same results, the natural FTIL way to specify this is not to certify the scripts that calculate the results, but only those that also trigger result retrieval. We use the directive `NOCERT` to make sure none of the result-calculating scripts get inadvertently certified.

Remark for the advanced reader 4.9.1. It's worth pointing out that this is a very unusual use for `NOCERT`. Typically, one uses `NOCERT` in a situation where one expects the calling script to perform some action prior to activating the `NOCERT`-labelled script, in order to set for the called script an appropriate context. Typically, it serves no purpose to not certify a script, "`A`", if some certified script, "`B`" calls "`A`", with user-chosen parameters, as the first thing it does. The reason for this is that an operator can simply preempt the execution of "`B`" once the underlying call to "`A`" has been completed. This would cause the code of "`A`" to execute in privileged mode without any additional context.

In this particular example, however, the situation is different in that the privacy policy we need to adhere to is that the REs are allowed to see any of their data *that AUSTRAC requests*. For

this reason, in this particular case, it is the very running of script "B" that provides the adequate context for "A": what "A" needs is not a software context, materialisable in the form of FTIL variable states, but rather a semantic context—as soon as AUSTRAC requests the information, it is allowed for the REs to see it, whether or not the software ultimately manages to deliver the information to AUSTRAC.

The portion of the process that computes the result but still does not return it to the coordinator is implemented in `FindNonzeroes` and `FindZeroes` below, where the former is the script that retrieves the account IDs for those accounts that have nonzero tag values and the latter retrieves those that have zero tag values. We also define in `TransmitTagValues` the code that is common to both scenarios, so as to avoid code duplication.

```
def TransmitTagValues(tag, dp_policy, epsilon, delta,
                      accounts, perm, plaintext, noise_exp):
  """
    Usage:
      TransmitTagValues is a helper script that sorts "tag"'s values in an
        arbitrary order, pads them, sends them to the coordinator and
        decrypts them. This is the first step in retrieving tag values.
    Inputs:
      tag -- The tag being queried.
      dp_policy -- The differential privacy policy.
      epsilon, delta -- How much differential privacy budget to use.
    Modified:
      None.
    Outputs:
      accounts -- The keys of "tag", in an arbitrary sorting order.
      perm -- The random permutation used.
      plaintext -- The padded plaintext tag values, in the order of "accounts".
      noise_exp -- Expected amount of noise tags generated.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(dp_policy, "Differential Privacy Policy{Signature}")

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):
    tag.UsedBudget(_used_epsilon, _used_delta)  # Budget already expended.
    # Asserting that there is still enough budget left for the operation.
    assert(epsilon + _used_epsilon <= dp_policy.epsilon)
    assert(delta + _used_delta <= dp_policy.delta)

  dp_policy.NewDistribution(epsilon, delta, _dist)

  on(_peer_ids):
    _dist.Expectation(noise_exp)
    _dist.NoisePair(_z0, _z1)

    # Saving as a memblock rather than as a set imposes on "accounts"
    # an arbitrary order.
    accounts = memblock<AccountID>(tag.map.keys())

    perm = RandomPerm(accounts.size() + _z0 + _z1)
```

```
  _tagvalues = transmitter<memblock<ElGamalCipher>>()
  on(_peer_ids):
    _tagvalues[CoordinatorID] =
      memblock<ElGamalCipher>(perm.size(), tag.keymgr.zero)
    _tagvalues[CoordinatorID][perm[i]] =
      {tag.map[accounts[i]]} over i = accounts.iter
    _tagvalues[CoordinatorID][perm[i + accounts.size()]] =
       {tag.keymgr.one} over i = arrayiter(_z1)

    tag.keymgr.ElGamalRefresh(_tagvalues[CoordinatorID])
    Sanitise(_tagvalues[CoordinatorID])

  tag.ExpendBudget(epsilon, delta)
  transmit(_tagvalues)

  on(CoordinatorID):
    plaintext = transmitter<memblock<Ed25519>>()
    for _n in _peer_ids:
      plaintext[_n] = ElGamalDecrypt(_tagvalues[_n], tag.keymgr.priv_key)


def FindNonzeroes(query_object, tag, policy, epsilon, delta, results) NOCERT:
  """
    Usage:
      FindNonzeroes determines the accounts that have nonzero values in a tag,
        but still does not return these results to the coordinator.
        The script aborts if the result set is too large, and provides
        two checkpoints for this: one before the exact size is known, and one
        after.
        This script should not be certified, in order to not enable calculation
        of the results without retrieving them. By not certifying this script,
        one can only execute it from within a valid context. For example, it
        can be run from RunQuery, if that script is certified. The use of the
        "NOCERT" modifier ensures that this is the case.
    Inputs:
      query_object -- Object managing this method.
      tag -- The tag being queried.
      policy -- The retrieval policy.
      epsilon, delta -- How much differential privacy budget to use.
    Modified:
      None.
    Outputs:
      results -- The results, still on the peer nodes, stored as a
                 set<AccountID>.
  """

  confirm_label(query_object, "FinTracer Query Object{Signature}")
  confirm_label(policy, "Retrieval Policy{Signature}")

  _noise_exp = transmitter<float>()

  TransmitTagValues(tag, policy.dp_policy, epsilon, delta,
                    _accounts, _perm, _plaintext, _noise_exp[CoordinatorID])
```

```
  transmit(_noise_exp)

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(CoordinatorID):
    _nomatch = Ed25519(0)
    _positions = transmitter<set<int>>()
    _total = 0
    for _n in _peer_ids:
      _positions[_n] = set{i over i = _plaintext[_n].iter
                                          with _plaintext[_n][i] != _nomatch}
      _total += _positions[_n].size() - _noise_exp[_n]

    limit(total <= policy.max_approx_query, "Expected result size too large.")
    if _total > policy.softmax_approx_query:
      confirm("Expected result size is " + str(_total) +
        ". Are you sure you wish to proceed?")

  transmit(_positions)

  on(_peer_ids):
    _invperm = dict<int, int>()
    _invperm[_perm[i]] = {i} over i = _perm.iter
    results = set{_accounts[_invperm[i]]
                                  over i = _positions[CoordinatorID].iter
                                        with _invperm[i] < _accounts.size()}

    limit(results.size() <= policy.max_exact_query,
            "Final result size too large.")


def FindZeroes(query_object, tag, policy, epsilon, delta, results) NOCERT:
  """
    Usage:
      FindZeroes determines the accounts that have zero values in a tag,
        but still does not return these results to the coordinator.
        The script aborts if the result set is too large, and provides
        two checkpoints for this: one before the exact size is known, and one
        after.
        This script should not be certified, in order to not enable calculation
        of the results without retrieving them. By not certifying this script,
        one can only execute it from within a valid context. For example, it
        can be run from RunQuery, if that script is certified. The use of the
        modifier NOCERT ensures that this is the case.
    Inputs:
      query_object -- Object managing this method.
      tag -- The tag being queried.
      policy -- The retrieval policy.
      epsilon, delta -- How much differential privacy budget to use.
    Modified:
      None.
    Outputs:
      results -- The results, still on the peer nodes, stored as a
                    set<AccountID>.
```

```
  """

  confirm_label(query_object, "FinTracer Query Object{Signature}")
  confirm_label(policy, "Retrieval Policy{Signature}")

  _noise_exp = transmitter<float>()

  TransmitTagValues(tag, policy.dp_policy, epsilon, delta,
                    _accounts, _perm, _plaintext, _noise_exp[CoordinatorID])

  transmit(_noise_exp)

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(CoordinatorID):
    _nomatch = Ed25519(0)
    _positions = transmitter<set<int>>()
    _total = 0
    for _n in _peer_ids:
      _positions[_n] = set{i over i = _plaintext[_n].iter
                                          with _plaintext[_n][i] == _nomatch}
      _total += _positions[_n].size() - _noise_exp[_n]

    limit(_total <= policy.max_approx_query, "Expected result size too large.")
    if _total > policy.softmax_approx_query:
      confirm("Expected result size is " + str(_total) +
        ". Are you sure you wish to proceed?")

  transmit(_positions)

  on(_peer_ids):
    _invperm = dict<int, int>()
    _invperm[_perm[i]] = {i} over i = _perm.iter
    results = set{_accounts[_invperm[i]]
                              over i = _positions[CoordinatorID].iter
                                        with _invperm[i] < _accounts.size()}

    limit(results.size() <= policy.max_exact_query,
            "Final result size too large.")
```

We now want to implement in FTIL a form of polymorphism: we want to be able to use `FindNonzeroes` and `FindZeroes` interchangeably from various other scripts (providing different contexts for their use) without having to provide separate code—requiring separate certification—according to whether we want to retrieve zeroes or nonzeroes in any specific invocation of these contexts.

The way to do this is to create `objects` that provide the same method interface to each of the two scripts `FindNonzeroes` and `FindZeroes`. This allows us to later send these wrapper objects as script parameters to the various context scripts that will want to use the retrieval scripts.

Below is the code that creates the two desired wrapper `objects`.

```
def NewNonzeroesQuery(ftquery, query_object):
  """
    Usage:
      NewNonzeroesQuery creates a query object that returns the nonzero entries
```

```
        of a tag.
    Inputs:
      ftquery -- The FinTracer Query module. (Unused.)
    Modified:
      None.
    Outputs:
      query_object -- The created object, labelled "FinTracer Query Object".
  """

  query_object = object()

  query_object.Query = ::FindNonzeroes

  label(query_object, "FinTracer Query Object{Signature}")


def NewZeroesQuery(ftquery, query_object):
  """
    Usage:
      NewZeroesQuery creates a query object that returns the zero entries
        of a tag.
    Inputs:
      ftquery -- The FinTracer Query module. (Unused.)
    Modified:
      None.
    Outputs:
      query_object -- The created object, labelled "FinTracer Query Object".
  """

  query_object = object()

  query_object.Query = ::FindZeroes

  label(query_object, "FinTracer Query Object{Signature}")
```

Ultimately, the functionality we want to expose to the coordinator-side user is one that returns the final results to the coordinator. This is a "top level context", presumably certified so it can be utilised directly from the FTIL command-line. However, as discussed, there are actually multiple querying contexts we want to allow. Specifically, we implement here two: first, in `RunQuery` we implement code that simply transfers the final results to the coordinator once they have been computed; then, in `RunQueryAndStore`, we implement a similar top-level context that additionally also stores the result on the peer nodes, so that it can also be utilised as an intermediate result for later computation.

Importantly, neither script is aware of whether it is used to retrieve zeroes or nonzeroes, as determining this is handled by the query object parameter, delivering FTIL's version of polymorphism.

```
def RunQuery(ftquery, query_object, tag, policy, epsilon, delta,
             final_results):
  """
    Usage:
      RunQuery runs the query in query_object, with the given parameters, and
        returns the result to the coordinator as final_results.
    Inputs:
```

```
      ftquery -- The FinTracer Query module proxy object. (Unused.)
      query_object -- The FinTracer query object.
      tag -- The tag being queried.
      policy -- The retrieval policy.
      epsilon, delta -- How much differential privacy budget to use.
    Modified:
      None.
    Outputs:
      final_results -- The results, stored on the coordinator as a
                       transmitter<set<AccountID>>.
  """

  confirm_label(query_object, "FinTracer Query Object{Signature}")

  query_object.Query(tag, policy, epsilon, delta, _remote_results)

  final_results = transmitter<set<AccountID>>()

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):
    final_results[CoordinatorID] = _remote_results

  del _remote_results  # Saving on the need to copy-on-write during 'transmit'.

  transmit(final_results)


def RunQueryAndStore(ftquery, query_object, tag, policy, epsilon, delta,
                     final_results, remote_results)
  """
    Usage:
      RunQueryAndStore is identical to RunQuery, except that the result is
        stored also in the peer nodes, for later use.
    Inputs:
      ftquery -- The FinTracer Query module proxy object. (Unused.)
      query_object -- The FinTracer query object.
      tag -- The tag being queried.
      policy -- The retrieval policy.
      epsilon, delta -- How much differential privacy budget to use.
    Modified:
      None.
    Outputs:
      final_results -- The results, stored on the coordinator as a
                       transmitter<set<AccountID>>.
      remote_results -- The results, stored on the peer nodes as a distributed
                        set<AccountID>.
  """

  confirm_label(query_object, "FinTracer Query Object{Signature}")

  query_object.Query(tag, policy, epsilon, delta, remote_results)

  final_results = transmitter<set<AccountID>>()
```

```
  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  on(_peer_ids):
    final_results[CoordinatorID] = remote_results

  transmit(final_results)
```

4.10 Basic nonlinear manipulation

Aside from querying of final results, the system supports also algorithms that move tag values through the coordinator in various oblivious ways in order to provide tools for manipulation of the tag values. This is important because (by the nature of the semi-homomorphic infrastructure we are using) natively it is only possible to compute on tags linear functions, such as $M_G v$.

The full gamut of nonlinear operations supported will be described in later sections, as these are more involved and are not needed in order to construct the code for the basic FinTracer pairwise run. In this section, however, we will nevertheless introduce the basic scripts over which these later capabilities will be developed.

The reason for the inclusion of this code here is that even though we want, in this case, to transfer tag values to the coordinator as intermediate results, and not as final results, we will nevertheless wish to employ the mechanisms of `TransmitTagValues` to implement the actual transfers. This makes the code in this section a natural fit for the FinTracer query module, allowing us to use `TransmitTagValues` without exposing it in any way outside the module.

4.10.1 Result normalisation

We implement two uses: normalising tag values into $\{0, 1\}$ values and negating tag values. Only one of these is actually needed, because the other can easily be computed from it, but we will present both functionalities, gaining by this a marginal saving in execution time.

Both require communication through the coordinator, and in both cases this has a cost in differential privacy budget.

```
def NormaliseTag(ftquery, tag, dp_policy, epsilon, delta):
  """
    Usage:
      NormaliseTag makes the values in tag be "1" if they are nonzero.
    Inputs:
      ftquery -- The FinTracer Query module proxy object. (Unused.)
      dp_policy -- The differential privacy policy.
      epsilon, delta -- How much differential privacy budget to expend.
    Modified:
      tag -- The tag being normalised.
    Outputs:
      None.
  """

  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(dp_policy, "Differential Privacy Policy{Signature}")

  TransmitTagValues(tag, dp_policy, epsilon, delta,
                    _accounts, _perm, _plaintext, _noise_exp)

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))
```

```
  _ciphertext = transmitter<memblock<ElGamalCipher>>()
  on(CoordinatorID):
    _nomatch = Ed25519(0)
    for _n in _peer_ids:
      _ciphertext[_n] = (_plaintext[_n][@] == _nomatch) ?
                          ElGamalCipher(tag.keymgr.zero : tag.keymger.one)
      tag.keymgr.ElGamalRefresh(_ciphertext[_n])

  transmit(_ciphertext)

  modifying(tag):
    on(_peer_ids):
      tag.map[_accounts[@]] = {_ciphertext[CoordinatorID][_perm[@]]}


def NegateTag(ftquery, tag, dp_policy, epsilon, delta, new_tag):
  """
    Usage:
      NegateTag creates a new FinTracer tag, "new_tag", which is "1" wherever
        tag is zero, and "0" wherever tag is nonzero. This is done in
        communication with the coordinator and has a price in differential
        privacy.
    Inputs:
      ftquery -- The FinTracer Query module proxy object.
      tag -- The tag being negated.
      dp_policy -- The differential privacy policy.
      epsilon, delta -- How much differential privacy budget to use.
    Modified:
      new_tag -- The newly-created tag value.
    Outputs:
      None.
  """

  confirm_label(ftquery, "FinTracer Query Module{Signature}")
  confirm_label(tag, "FinTracer Tag{Signature}")
  confirm_label(dp_policy, "Differential Privacy Policy{Signature}")

  TransmitTagValues(tag, dp_policy, epsilon, delta,
                    _accounts, _perm, _plaintext, _noise_exp)

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  _ciphertext = transmitter<memblock<ElGamalCipher>>()
  on(CoordinatorID):
    _nomatch = Ed25519(0)
    for _n in _peer_ids:
      _ciphertext[_n] = (_plaintext[_n][@] == _nomatch) ?
                          ElGamalCipher(tag.keymgr.one : tag.keymger.zero)
      tag.keymgr.ElGamalRefresh(_ciphertext[_n])

  transmit(_ciphertext)

  # We begin by cloning the original tag. This gives us its sources and
```

```
  # differential privacy expenditures. We unnecessarily also copy its map,
  # which we immediately override, but this does not involve data copying so
  # is negligible.
  ftquery.ftcore.CloneToNewTag(tag, new_tag)

  on(_peer_ids):
    modifying(new_tag):
      new_tag.map[_accounts[@]] = {_ciphertext[CoordinatorID][_perm[@]]}
```

4.10.2 Tag sets

One doesn't have to normalise and negate tags individually. If one has an entire catalogue of tags, one can do it jointly, with less information revealed to the coordinator and also slightly less overhead of added noise.

Ultimately, we will develop a script "`TransmitTagSetValues`" which will be an extension of "`TransmitTagValues`" that works on entire sets (and therefore still naturally fitting within the FinTracer query module), but for this we must first develop the basic machinery for handling tag sets.

We begin this here by defining tag-set-managing `objects`.

The tag-set-managing `objects`, to be labelled "FinTracer Tag Set", are defined as follows. These are `objects` that manage a set of FinTracer tags, all held inside a `collection`. These `collections` are the unique managers of the tags within them, in the sense that these tags have no other references. Each set must contain at least one tag, and all tags in a set must share the same key manager `object`.

Tag sets support a specific interface. It includes the member attributes `members` as the `catalogue` listing all tag members in the set, `keymgr` as the key manager shared among all tags in the collection, and `ftcore` as a reference to the core FinTracer module. It include the method "`.Add`", which adds a tag to the tag set.

```
def TagSetAddTag(tag_set, tag):
  """
    Usage:
      TagSetAddTag adds "tag" to the tag set "tag_set".
    Inputs:
      tag -- The tag to be added to the set.
    Modified:
      tag_set -- The tag set being modified.
    Outputs:
      None.
  """

  confirm_label(tag_set, "FinTracer Tag Set{Signature}")
  confirm_label(tag, "FinTracer Tag{Signature}")
  assert(tag.keymgr == tag_set.keymgr)

  modifying(tag_set.members):
    # Note: In FTIL one is only allowed to send variables as user script
    # and user-method parameters, but they can still use indirect addressing.
    tag_set.ftcore.CloneToNewTag(tag, tag_set.members[tag.metadata.id])


def NewEmptyTagSet(ftquery, keymgr, tag_set):
  """
    Usage:
```

```
      NewEmptyTagSet is a helper script that constructs an empty tag set from a
        key manager object. However, it does not label the tag set, because we
        require tag sets to never be empty.
    Inputs:
      ftquery -- The FinTracer query module proxy object.
      keymgr -- The key manager to be the common key manager for all future
                  tags in the tag set.
    Modified:
      None.
    Outputs:
      tag_set -- The tag set being constructed, to be labelled
                 "FinTracer Tag Set".
  """

  confirm_label(ftquery, "FinTracer Query Module{Signature}")
  confirm_label(keymgr, "FinTracer Key{Signature}")

  _tag_set = object()

  _tag_set.members = catalogue()

  label(_tag_set.members, "FinTracer Tag Set Members{Signature}")

  _tag_set.keymgr = keymgr
  _tag_set.ftcore = ftquery.ftcore

  _tag_set.Add = ::TagSetAddTag

  tag_set = _tag_set
  # del _tag_set

  # Importantly, we do not, at this point, proceed to perform a
  # label(tag_set, "FinTracer Tag Set{Signature}")
  # This is because tag sets are not allowed to be empty.
  # Note that labelling the tag set would not have been allowed even in a
  # "NOCERT" script, because any calling script could have potentially been
  # aborted immediately after returning from this script, leading to an
  # erroneously labelled object.


def NewTagSetFromTag(ftquery, tag, tag_set):
  """
    Usage:
      NewTagSetFromTag constructs a tag set from a tag.
    Inputs:
      ftquery -- The FinTracer query module proxy object.
      tag -- The tag to be made into a set.
    Modified:
      None.
    Outputs:
      tag_set -- The tag set being constructed, to be labelled
                 "FinTracer Tag Set".
  """
```

```
  confirm_label(ftquery, "FinTracer Query Module{Signature}")
  confirm_label(tag, "FinTracer Tag{Signature}")

  ftquery.NewEmptyTagSet(tag.keymgr, _tag_set)

  modifying(_tag_set.members):
    # Note: In FTIL one is only allowed to send variables as user script
    # and user-method parameters, but they can still use indirect addressing.
    ftquery.ftcore.CloneToNewTag(tag, _tag_set.members[tag.metadata.id])

  tag_set = _tag_set
  del _tag_set

  label(tag_set, "FinTracer Tag Set{Signature}")


def NewTagSetFromPair(ftquery, tag1, tag2, tag_set):
  """
    Usage:
      NewTagSetFromPair constructs a tag set from two tags.
        It is a convenience function.
    Inputs:
      ftquery -- The FinTracer query module proxy object.
      tag1, tag2 -- The tags to be made into a set.
    Modified:
      None.
    Outputs:
      tag_set -- The tag set being constructed.
  """

  confirm_label(ftquery, "FinTracer Query Module{Signature}")
  ftquery.NewTagSetFromTag(tag1, _tag_set)

  _tag_set.Add(tag2)

  tag_set = _tag_set

  confirm_label(tag_set, "FinTracer Tag Set{Signature}")


def NewTagSetFromCatalogue(ftquery, tag_cat, tag_set):
  """
    Usage:
      NewTagSetFromCatalogue constructs a tag set from a catalogue of tags.
        It is a convenience function.
    Inputs:
      ftquery -- The FinTracer query module proxy object.
      tag_cat -- The input tag catalogue.
    Modified:
      None.
    Outputs:
      tag_set -- The tag set being constructed.
  """
```

```
  # We could have done at this point
  #   _keys = tag_cat.keys()
  # and set ".keys()" into a temporary variable, but that would only work
  # if the scope in which this script is executed includes the coordinator.
  # Otherwise, for defining the range of the loop we need to only use values
  # that are accessible to the Compute Manager. For this reason, we use
  # "tag_cat.keys()", which is metadata, directly, rather than "_keys", which
  # would have been data.

  assert(tag_cat.size() > 0)

  confirm_label(ftquery, "FinTracer Query Module{Signature}")
  ftquery.NewEmptyTagSet(tag_cat[tag_cat.keys()[0]].keymgr, _tag_set)

  for _i in tag_cat.keys():
    _tag_set.Add(tag_cat[_i])

  tag_set = _tag_set

  label(tag_set, "FinTracer Tag Set{Signature}")
```

4.10.3 Manipulating tag sets

We now turn to implementing the script `TransmitTagSetValues`.

If used on a tag set of size 1, this script behaves in ways comparable to `TransmitTagValues`. When multiple tags are queried simultaneously, however, the script considers the sources of the tags involved and tailors a noise distribution that best utilises the budget allocated for the procedure. See Section 5.4 for a full analysis.

In terms of the amounts of noise (extra tags) generated, this will always be smaller, in expectation, than the noise of querying each tag separately, and even more so when the tags are independent in terms of their sources.

Normally, however, tags queried together would share some of their sources, so noise levels need to be adjusted accordingly.

For example, if all tags queried together have been reduced to the same target account set, querying all together is, given the same amount of noise, more revealing than querying them separately, in terms of whether a particular account appears in the target set or not. The code below measures this effect and computes the appropriate noise level.

```
def TransmitTagSetValues(tag_set, dp_policy, epsilon, delta,
                         accounts_cat, perm_cat, plaintext, noise_exp):
  """
    Usage:
      TransmitTagSetValues is a generalisation over TransmitTagValues,
        the difference being that this script handles multiple tags jointly.
    Inputs:
      tag_set -- The tags being queried (in a tag set object).
      dp_policy -- The differential privacy policy.
      epsilon, delta -- How much differential privacy budget to use.
    Modified:
      None.
    Outputs:
      accounts_cat -- The keys of each tag, in an arbitrary sorting order,
                      in a catalogue.
      perm_cat -- The random permutations used, as a catalogue.
```

```
                (All catalogues share the same keys as tag_set.members.)
    plaintext -- The padded plaintext tag values, in the order of
                   "accounts_cat".
    noise_exp -- Expected amount of noise tags generated.
"""

confirm_label(tag_set, "FinTracer Tag Set{Signature}")
confirm_label(dp_policy, "Differential Privacy Policy{Signature}")

_peer_ids = setminus(scope(), nodeset(CoordinatorID))

on(_peer_ids):
  # We count for each source how many times it is reused.
  repetitions = distribute(dict<int, int>())

  for _i in tag_set.members.keys():
    for _s in tag_set.members[_i].sources.keys():
      repetitions[_s] = repetitions.lookup(_s, 0) + 1

  max_repetition = max{repetitions[i]} over i = repetitions.iter

  _epsilon_portion = epsilon / max_repetition
  _delta_portion = delta / max_repetition

modifying(tag_set):
  # We now try to use the available budget and abort if it is not enough.
  for _i in tag_set.members.keys():
    tag_set.members[_i].ExpendBudget(epsilon_portion, delta_portion)

  for _i in tag_set.members.keys():
    tag_set.members[_i].UsedBudget(_used_epsilon, _used_delta)
    on(_peer_ids):
      assert(_used_epsilon <= dp_policy.epsilon)
      assert(_used_delta <= dp_policy.delta)

dp_policy.NewDistribution(_epsilon_portion, _delta_portion, _dist)

on(_peer_ids):
  _dist.Expectation(noise_exp)
  _dist.NoisePair(_z0, _z1)

  accounts_cat = catalogue()
  _total_size = 0

  for _i in tag_set.members.keys():
    # Saving as a memblock rather than as a set imposes on "accounts"
    # an arbitrary order. (Note: return value from .keys() is already a list.)
    accounts_cat[_i] = object()
    accounts_cat[_i].accounts =
      memblock<AccountID>(tag_set.members[_i].map.keys())
    _total_size += accounts_cat[_i].accounts.size()

  _total_perm = RandomPerm(_total_size + _z0 + _z1)
```

```
    _total_size = 0
    perm_cat = catalogue()
    for _i in tag_set.members.keys():
      perm_cat[_i] = object()
      perm_cat[_i].perm =
        _total_perm[_total_size :
                    _total_size + accounts_cat[_i].accounts.size()]
      _total_size += accounts_cat[_i].accounts.size()

  _tagvalues = transmitter<memblock<ElGamalCipher>>()
  on(_peer_ids):
    _tagvalues[CoordinatorID] =
      memblock<ElGamalCipher>(_total_perm.size(), tag_set.keymgr.zero)

    for _i in tag_set.members.keys():
      _tagvalues[CoordinatorID][perm_cat[_i].perm[j]] =
        {tag.map[accounts_cat[_i].accounts[j]]}
          over j = accounts_cat[_i].accounts.iter

    _tagvalues[CoordinatorID][_total_perm[i + _total_size]] =
       {tag_set.keymgr.one} over i = arrayiter(_z1)

    tag_set.keymgr.ElGamalRefresh(_tagvalues[CoordinatorID])
    Sanitise(_tagvalues[CoordinatorID])

  transmit(_tagvalues)

  on(CoordinatorID):
    plaintext = transmitter<memblock<Ed25519>>()
    for _n in _peer_ids:
      plaintext[_n] = ElGamalDecrypt(_tagvalues[_n], tag_set.keymgr.priv_key)
```

We now implement the set versions of normalisation and negation.

```
def NormaliseTagSet(ftquery, tag_set, dp_policy, epsilon, delta):
  """
    Usage:
      NormaliseTagSet is identical to NormaliseTag, but works on sets of tags.
    Inputs:
      ftquery -- The FinTracer Query module proxy object. (Unused.)
      dp_policy -- The differential privacy policy.
      epsilon, delta -- How much differential privacy budget to use.
    Modified:
      tag_set -- The set of tags being normalised.
    Outputs:
      None.
  """

  confirm_label(tag_set, "FinTracer Tag Set{Signature}")
  confirm_label(dp_policy, "Differential Privacy Policy{Signature}")

  TransmitTagSetValues(tag_set, dp_policy, epsilon, delta,
                       _accounts_cat, _perm_cat, _plaintext, _noise_exp)
```

```
  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  _ciphertext = transmitter<memblock<ElGamalCipher>>()
  on(CoordinatorID):
    _nomatch = Ed25519(0)
    for _n in _peer_ids:
      _ciphertext[_n] = (_plaintext[_n][@] == _nomatch) ?
                    ElGamalCipher(tag_set.keymgr.zero : tag_set.keymgr.one)
      tag_set.keymgr.ElGamalRefresh(_ciphertext[_n])

  transmit(_ciphertext)

  for _i in tag_set.members.keys():
    modifying(tag_set.members[_i]):
      on(_peer_ids):
        tag_set.members[_i].map[_accounts_cat[_i].accounts[@]] =
                        {_ciphertext[CoordinatorID][_perm_cat[_i].perm[@]]}


def NegateTagSet(ftquery, tag_set, dp_policy, epsilon, delta, new_tag_set):
  """
    Usage:
      NegateTagSet is a generalisation over NegateTag, that works on tag sets.
    Inputs:
      ftquery -- The FinTracer Query module proxy object.
      tag_set -- The set of tags being negated.
      dp_policy -- The differential privacy policy.
      epsilon, delta -- How much differential privacy budget to use.
    Modified:
      None.
    Outputs:
      new_tag_set -- The newly-created result tag set.
  """

  confirm_label(ftquery, "FinTracer Query Module{Signature}")
  confirm_label(tag_set, "FinTracer Tag Set{Signature}")
  confirm_label(dp_policy, "Differential Privacy Policy{Signature}")

  TransmitTagSetValues(tag_set, dp_policy, epsilon, delta,
                    _accounts_cat, _perm_cat, _plaintext, _noise_exp)

  _peer_ids = setminus(scope(), nodeset(CoordinatorID))

  _ciphertext = transmitter<memblock<ElGamalCipher>>()
  on(CoordinatorID):
    _nomatch = Ed25519(0)
    for _n in _peer_ids:
      _ciphertext[_n] = (_plaintext[_n][@] == _nomatch) ?
                     ElGamalCipher(tag_set.keymgr.one : tag_set.keymger.zero)
      tag_set.keymgr.ElGamalRefresh(_ciphertext[_n])

  transmit(_ciphertext)

  # We begin by cloning the original tags. This gives us their sources and
```

```
  # differential privacy expenditures. We unnecessarily also copy their maps,
  # which we immediately override, but this does not involve data copying so
  # is negligible.
  ftquery.NewTagSetFromCatalogue(tag_set.members, new_tag_set)


  # Though we are not actively modifying "new_tag_set" itself, only its
  # members, during this operation new_tag_set is not fit-for-purpose: some of
  # its members may be negated while others are not.
  modifying(new_tag_set):
    for _i in new_tag_set.members.keys():
      modifying(new_tag_set.members[_i]):
        on(_peer_ids):
          new_tag_set.members[_i].map[_accounts_cat[_i].accounts[@]] =
                          {_ciphertext[CoordinatorID][_perm_cat[_i].perm[@]]}
```

4.10.4 The query module

We conclude our present exploration of FinTracer querying by listing the remaining portions of the `ftquery.ftil` module file.

```
FTIL1.0
module ftquery
using ftcore

def __init(ftquery, modules):
  """
    Usage:
      __init script for the FinTracer query module.
    Inputs:
      modules -- The modules object. We use modules.ftcore
    Modified:
      None.
    Outputs:
      ftquery -- The new module proxy object generated, to be labelled
                 "FinTracer Query Module".
  """

  confirm_label(modules.ftcore, "FinTracer Core Module{Signature}")

  ftquery = object()

  ftquery.ftcore = modules.ftcore

  ftquery.NewDPPolicy = ::NewDPPolicy
  ftquery.NewRetrievalPolicy = ::NewRetrievalPolicy
  ftquery.NewNonzeroesQuery = ::NewNonzeroesQuery
  ftquery.NewZeroesQuery = ::NewZeroesQuery
  ftquery.RunQuery = ::RunQuery
  ftquery.RunQueryAndStore = ::RunQueryAndStore
  ftquery.NormaliseTag = ::NormaliseTag
  ftquery.NegateTag = ::NegateTag
  ftquery.NewTagSetFromTag = ::NewTagSetFromTag
  ftquery.NewTagSetFromPair = ::NewTagSetFromPair
```

```
ftquery.NewTagSetFromCatalogue = ::NewTagSetFromCatalogue
ftquery.NormaliseTagSet = ::NormaliseTagSet
ftquery.NegateTagSet = ::NegateTagSet

label(ftquery, "FinTracer Query Module{Signature}")
```

4.11 Pairwise run

We have shown how to set up a tag, how to propagate it and how to read the results. To pull this all together, here is the code for a no-frills version of the *FinTracer pairwise run*, the simplest version of running the FinTracer machinery.

In this version, the AUSTRAC user defines via SQL descriptions the following elements:

1. A set of *source accounts*, A,
2. A set of *destination accounts*, B,
3. A date range, $[start\_date, end\_date]$,
4. A set of two-sided-viewable *connections* based on direct transaction information within the date range, G_2,
5. A set of one-sided-viewable *qualified connections* that is a subset of the two-sided-viewable set of connections (e.g., excluding from the connections any whitelisted accounts), G_1, and
6. A *maximum distance*, k.

The object here is to find the subset of the destination accounts that can be reached from the source account using at most k hops along the G_1 graph, while maintaining all previously-discussed privacy requirements. Because this is assumed to be a standalone query, and no partial results are kept for any future use, we will use the full available differential privacy budget.

The scripts performing all required actions have already been defined in previous sections. Here, we demonstrate an end-to-end run of the algorithm.

First, let us define a script to perform the pairwise run. This is not necessary to do, but demonstrates that the system can work at our choice of level of abstraction. The code has been annotated with more comments than our other examples, to help the reader navigate through it.

We present it here as its own standalone module, which we call `ftpairwise_e2e`. Below is the full module-file text.

The only part of the process that is assumed to occur outside the script is the generation of a new key manager. This is because key managers are likely to be used again and again for many queries, due to their stockpiling of encrypted zeroes.

Note 4.11.1. The code of `ftpairwise_e2e` begins with the helper script `Operator1DirFromSQL`, which defines an operator from SQL instructions. This code is essentially a copy of code that has already appeared in Section 4.5.7. The only difference is that here it is packaged as a module script.

```
FTIL1.0
module ftpairwise_e2e
using ftcore, ftop1dir, ftquery

def Operator1DirFromSQL(ftpairwise_e2e, two_sided_cond, start_date, end_date,
                        one_sided_cond, op):
  """
    Usage:
      Operator1DirFromSQL is a convenience function for creating a
```

```
        FinTracer one-directional operator from SQL queries.
    Inputs:
      ftpairwise_e2e -- The module proxy object.
      two_sided_cond -- The SQL conditional defining the two-sided query.
      start_date -- Earliest date to match the two-sided query.
      end_date -- Latest date to match the two-sided query.
      one_sided_cond -- SQL conditional defining one-sided query,
                          not time-bounded.
    Modified:
      None.
    Outputs:
      op -- The new FinTracer operator, to be labelled "FinTracer 1Dir".
  """

  confirm_label(ftpairwise_e2e, "FinTracer Pairwise Run Module{Signature}")

  _ftcore = ftpairwise_e2e.ftcore

  _ftcore.NewTwoSidedGraph(two_sided_cond, start_date, end_date, _tsv_graph)
  _ftcore.OneSidedTransactions(one_sided_cond, _osv_in, _osv_out)
  _ftcore.NewPropagationGraph(_tsv_graph, _osv_in, _osv_out, _graph)
  _ftcore.Operator1DirFromGraph(_graph, op)


def FinTracerPairwiseRun(ftpairwise_e2e, keymgr, query_for_A, query_for_B,
      query_for_G_2, start_date, end_date, query_for_G_1, k, found_accounts):
  """
    Usage:
      This script performs a full, end-to-end FinTracer pairwise run, good
        if what you're looking for is precisely the default behaviour for a
        pairwise run and nothing needs tweaking. Receives descriptions for
        source and target account sets, as well as for a graph (including both
        two-sided-viewable and one-sided-viewable components) and returns the
        subset of the target accounts that are at most k hops away from any
        source account on the graph.
    Inputs:
      ftpairwise_e2e -- The module proxy object.
      keymgr -- A FinTracer key manager.
      query_for_A -- SQL description of the source account set.
      query_for_B -- SQL description of the target account set.
      query_for_G_2 -- A query describing the two-sided-viewable graph.
      start_date -- Starting date to run the query on, on the Transaction Store.
      end_date -- Ending date to run the query on, on the Transaction Store.
      query_for_G_1 -- SQL description of the one-sided-viewable graph that
                       will actually be used for tag propagation.
      k -- Paths will be searched that are between length 1 and k.
    Modified:
      None.
    Outputs:
      found_accounts -- The result set, as transmitter<set<AccountID>>.
  """

  confirm_label(ftpairwise_e2e, "FinTracer Pairwise Run Module{Signature}")
  confirm_label(keymgr, "FinTracer Key{Signature}")
```

```
  # We define the FinTracer propagation graph, and create an operator that
  # walks a single hop along the graph.
  ftpairwise_e2e.Operator1DirFromSQL(
    query_for_G_2, start_date, end_date, query_for_G_1, _one_hop_op)

  # From the first operator, we compute the more sophisticated operator, namely
  # one performing at-most-k steps on the propagation graph.
  ftpairwise_e2e.ftop1dir.Operator1DirPathAtMost(
    _one_hop_op, k, _at_most_k_hop_op)

  # The REs specify the source accounts using a SQL query provided by AUSTRAC.
  ftpairwise_e2e.ftcore.AccountsFromSQL(query_for_A, _source_accounts)

  # The source set is then converted to a FinTracer tag.
  ftpairwise_e2e.ftcore.TagFromAccounts(keymgr, _source_accounts, _start_tag)

  # We propagate the tag, using the at-most-k-steps operator.

  # NOTE: No need to
  # confirm_label(_at_most_k_hop_op, "Operator 1Dir{Signature}")
  # because it was already pre-confirmed in Operator1DirPathAtMost.

  _at_most_k_hop_op.Forward(_start_tag, _result_tag)

  # The REs specify the destination accounts using another SQL query.
  ftpairwise_e2e.ftcore.AccountsFromSQL(query_for_B, _destination_accounts)

  # We now want _result_tag to only include information about accounts in B.
  _result_tag.Reduce(_destination_accounts)

  # We set privacy policy parameters for the querying of tag values.
  ftpairwise_e2e.ftquery.NewRetrievalPolicy(_policy)

  # We now create a query object to say that we're interested in the nonzero
  # values in _result_tag.
  ftpairwise_e2e.ftquery.NewNonzeroesQuery(_query_object)

  # ... and go ahead to run the query, receiving the final results in
  # found_accounts. We send epsilon and delta values that we got from the
  # policy object, to indicate that we simply want to exhaust all available
  # differential privacy budget.
  ftpairwise_e2e.ftquery.RunQuery(_query_object, _result_tag, _policy,
    _policy.dp_policy.epsilon, _policy.dp_policy.delta, found_accounts)


def __init(ftpairwise_e2e, modules):
  """
    Usage:
      __init script for the FinTracer pairwise module.
    Inputs:
      modules -- The modules object. We use modules.ftcore, modules.ftop1dir
                 and modules.ftquery.
    Modified:
```

```
      None.
    Outputs:
      ftpairwise_e2e -- The new module proxy object generated, to be labelled
                        "FinTracer Pairwise Run Module".
  """

  confirm_label(modules.ftcore, "FinTracer Core Module{Signature}")
  confirm_label(modules.ftop1dir, "Operator 1Dir Module{Signature}")
  confirm_label(modules.ftquery, "FinTracer Query Module{Signature}")

  ftpairwise_e2e = object()

  ftpairwise_e2e.ftcore = modules.ftcore
  ftpairwise_e2e.ftop1dir = modules.ftop1dir
  ftpairwise_e2e.ftquery = modules.ftquery

  ftpairwise_e2e.Operator1DirFromSQL = ::Operator1DirFromSQL
  ftpairwise_e2e.PairwiseRun = ::FinTracerPairwiseRun

  label(ftpairwise_e2e, "FinTracer Pairwise Run Module{Signature}")
```

For the analyst specifically after this kind of a pairwise run, the only lines of code required outside the running of the method `PairwiseRun` are the setting up of a new FinTracer key and the generation of a stockpile of zeroes. These are, per design, to be performed offline from the main code, and not necessarily once per execution of a pairwise run. Here is the code to be run by the intel analyst for an end-to-end run, from a "cold start":

```
import "file://ftkeys.ftil"
ftkeys.NewMgr(keymgr)
keymgr.AddZeroes()

import "file://ftcore.ftil"
import "file://ftop1dir.ftil"
import "file://ftquery.ftil"
import "file://ftpairwise_e2e.ftil"

ftpairwise_e2e.PairwiseRun(keymgr, "query for A", "query for B",
  "query for G_2", start_date, end_date, "query for G_1", k, found_accounts)
```

Alternatively, the FTIL user may take our advice from Section 11.2.1 and may want to put all their objects under `investigation_a`, and specifically all their modules under the sub-object `investigation_a.modules`. The code above, modified for this type of artefact arrangement, looks like this:

```
investigation_a = object()
investigation_a.modules = object()

import "file://ftkeys.ftil" to investigation_a.modules
investigation_a.modules.ftkeys.NewMgr(investigation_a.keymgr)
investigation_a.keymgr.AddZeroes()

import "file://ftcore.ftil" to investigation_a.modules
import "file://ftop1dir.ftil" to investigation_a.modules
import "file://ftquery.ftil" to investigation_a.modules
import "file://ftpairwise_e2e.ftil" to investigation_a.modules
```

```
investigation_a.modules.ftpairwise_e2e.PairwiseRun(
  investigation_a.keymgr, "query for A", "query for B",
  "query for G_2", start_date, end_date, "query for G_1", k,
  investigation_a.found_accounts)
```

This is essentially a one-line execution, placing the result in the new variable `found_accounts` which is visible by the coordinator node only.

At the same time, an analyst wanting to generate account sets in more sophisticated ways, manipulate tags in more sophisticated ways, or use the querying mechanism in order to resolve FinTracer queries that are more involved than just an A-to-B pairwise run, still can.

4.12 FinTracer DARK

4.12.1 setup

4.12.2 DARK iterations

4.12.3 DARK link alternatives

4.12.4 Passage through unmonitored accounts

4.12.5 unsigned DARK

4.13 Oblivious queries

4.13.1 Setting the source accounts

4.13.2 Reading tags

Chapter 5

Cryptography

This chapter details the basic cryptographic / privacy techniques underlying the construction described in the previous chapters.

5.1 ElGamal semi-homomorphic encryption

5.2 Digital signatures

5.3 Hashing

5.4 Differential privacy

Note 5.4.1. This section discusses the general concept of differential privacy, and the specific differential privacy algorithms in use within the FinTracer suite.

It should be noted that differential privacy is never an air-tight solution. Readers wanting to understand the limitations of this technology and how to manage them should refer to Section 10.1.

In Section 4.8, we describe an algorithm for querying the values of a tag, whose account domain is some set S, unknown to AUSTRAC. This may be a final query, after which some account IDs in S are revealed to AUSTRAC, or not (e.g., the values may simply be given to AUSTRAC for the purpose of normalisation).

Ultimately, the RE needs to communicate to AUSTRAC whether each element of S is tag-positive or not. This is $|S|$ bits of communication, so no protocol can work unless its expected communication size to AUSTRAC is $\Omega(|S|)$. Completely hiding the size of S therefore requires us to always communicate an amount of bits that would be an upper bound for $|S|$. A simple way to do this would have been to just send a tag value for more accounts than any RE would possibly have. The problem is that for small S, this is an unreasonable inflation in communication requirements.

To demonstrate: we expect an upper bound for the number of accounts in Australia to be on the order of 2^{27}. This means that communicating one tag value for each totals roughly 16GB. Even after compressing the tags, this is still 4GB. The theoretical minimum is 128MB, amounting to only a single bit value per account, but approaching this value is not possible because the tag values are encrypted values whose key is not known to the REs, so any ability to compress below 4GB would imply a cryptographic weakness, because it would show that the REs have more information about the tag values than by rights they should have.

Assuming that this amount of communication would be excessive just for the common task of querying moderate-sized sets, the protection any "reasonable" algorithm can afford is limited. This being the case, it is important to specify exactly what types of attacks we are protecting against, and what guarantees we provide against such attacks.

Specifically, we will be protecting against revealing whether or not a specific account is a member of any specific set that has been used as an information source in the creation of the tag. The information source could have been used to define the account domain or in order to define the tag-positive accounts.

Against this, we provide provable (ϵ, δ)-differential privacy guarantees. A probabilistic algorithm, $\mathcal{A}$ is said to be (ϵ, δ)-*differentially private* if for any input X (described as a set), with probability at least $1 - \delta$, the output of $\mathcal{A}$ on X, v, is such that for any input Y that is different to X by only a single element,

$$\exp(-\epsilon)\text{Prob}[A(Y) = v] \leq \text{Prob}[A(X) = v] \leq \exp(\epsilon)\text{Prob}[A(Y) = v].$$

Note that this definition is closed to composition:

- If we run two algorithms, one providing (ϵ_1, δ_1)-differential privacy and the other providing (ϵ_2, δ_2)-differential privacy, the overall algorithm provides $(\epsilon_1+\epsilon_2, \delta_1+\delta_2)$-differential privacy.
- If an algorithm provides (ϵ, δ)-differential privacy against a particular type of change, it also provides $(k\epsilon, k\delta)$-differential privacy against k such changes.

In the definition of (ϵ, δ)-differential privacy, larger ϵ and δ correspond to less privacy.

5.4.1 Algorithm high-level overview

The basic idea behind all our querying algorithms is that an RE needing to get AUSTRAC input on the values of a given tag, t, sends the values of the tag to AUSTRAC without the associated account ID information. Depending on the specific situation, AUSTRAC may reply with decrypted plaintext tag values, with normalised ciphertexts, or with other variations, but at least at this point in the process the account IDs themselves are not revealed to AUSTRAC in any way.

To prevent AUSTRAC from receiving more than just the zero-or-nonzero information of the account values, the REs sanitise and refresh the values before sending them to AUSTRAC. To prevent the order of the values sent from carrying any information, these are first permuted by a random permutation. (Critically, neither the random values used in sanitisation/refreshing nor those used in permutation generation are ever reused. They are allotted independently each time.)

Thus, the only information revealed to AUSTRAC even without applying any differential privacy measures, is solely the number of accounts in S that have zero tag values, s_0, and the number of accounts in S that have nonzero tag values, s_1.

Differential privacy protects these values by inserting a random number z_0 of zero-valued elements and a random number z_1 of nonzero-valued elements.

Thus, what AUSTRAC sees is $s_0 + z_0$ and $s_1 + z_1$.

Consider, now, what z_0 and z_1 must satisfy, in order to provide (ϵ, δ)-differential privacy.

We initially create tags by use of multiple information sources, each characterisable as a set of accounts, S_0. For each such S_0, we want our algorithm's behaviour to be indistinguishable, in the (ϵ, δ)-differential privacy sense, from its behaviour had the value of S_0 been S_0', where the symmetric difference between S_0 and S_0' is exactly one element.

Because of the way we construct tags (excluding the approximation made regarding tag propagation, which was introduced in Section 4.5.6 and will be discussed in greater detail in Section 10.1.1), such a change in S_0 can change the value of s_0 by at most 1 and the value of s_1 by at most 1. Because AUSTRAC can observe s_0 and s_1 separately, this counts as two observations of S_0, and, accordingly, we set z_0 and z_1 so as to provide $(\epsilon/2, \delta/2)$-differential privacy for each observation.

Recall, also, that multiple tags can be queried simultaneously. In this case, if a set S_0 is a source for k of the queried tags, its impact is observed not just twice, but $2k$ times. We must therefore set our parameters so as to provide $(\frac{\epsilon}{2k}, \frac{\delta}{2k})$-differential privacy for each observation. The actual parameters chosen are based on the set that is utilised—and therefore also measured—the largest number of times. Typically, there will be one set that will be used in all tags queried

simultaneously, because when two tags are queried as part of a common investigation there would normally be one base set of accounts that both will have been reduced to. However, the program does not assume this.[1]

Because of the compositional properties of (ϵ, δ)-differential privacy, we can measure for each source the amount of (ϵ, δ)-differential privacy afforded to it. Every time a source is observed, we add the ϵ and δ values of the specific observation into the record of expended ϵ and δ budgets for the source. This is done independently for all sources.

The program makes sure that the overall privacy afforded to any source is never below the thresholds set by the program's differential privacy policy.

5.4.2 Semi-truncated Laplace distributions

When adding z noise elements to obscure the size of a set, the condition of (ϵ, δ)-differential privacy boils down to the following: for all possible values of z, except for values with a total probability of at most δ, the probability assigned to z must be within a multiplicative factor of $\exp(\epsilon)$ from the probabilities assigned to both $z-1$ and $z+1$.

Normally, differential privacy algorithms generate their noise, z, from a Laplace distribution, i.e. a distribution where the probability to obtain any specific value for z is proportional to $\exp(-\epsilon(|z-y|))$, for some constant integer y. This provides a stronger privacy condition, namely ϵ-differential privacy, which is (ϵ, δ)-differential privacy with $\delta = 0$.

In our case, however, it is not possible to use a Laplace distribution, because we can only add noise elements, not subtract them. This requires us to use a distribution truncated at zero, which will necessarily have $\delta > 0$ because at least the minimal positive-probability value of z will not meet the criterion of having the same probability as $z-1$ up to a multiplicative factor.

Let us call z values for which the probability assigned to z is within a multiplicative factor of $\exp(\epsilon)$ from the probabilities assigned to both $z-1$ and $z+1$ as z values meeting the *ϵ-condition*.

While we cannot meet the ϵ-condition for every value of z, we have opted for the "next best thing", which is to meet the condition for every value of z except zero.

Among distributions that meet this stronger criterion, the one with smallest expectation (which is a desirable property in order to limit the amount of noise handled by the system) is the distribution described as $D_{\epsilon,\delta}$ in Section 4.8.1.

Proving this fact is relatively straightforward: consider any distribution P over the nonnegative integers such that for every $\boldsymbol{z} \sim P$, $\mathbf{Prob}[\boldsymbol{z} = z] \leq e^{\epsilon}\mathbf{Prob}[\boldsymbol{z} = z+1]$. Such a distribution is necessarily a weighted average of shifted exponential distributions, E_y, where the probability for $\boldsymbol{z} \sim E_y$ to be any value $z \geq y$ is proportional to $\exp(-\epsilon(z-y))$ and zero otherwise.

Consider any distribution created by such a weighted average that also meets all of our other criteria. The weight α_0 associated with E_0 cannot be greater than what would give $\mathbf{Prob}[\boldsymbol{z} = 0] = \delta$, and for all $y > 0$ the weight α_y associated with any E_y cannot be greater than what would give for $\boldsymbol{z_y} \sim E_y$

$$\mathbf{Prob}[\boldsymbol{z_y} = y] = \delta \exp(\epsilon(y-1))(\exp(\epsilon) - \exp(-\epsilon)).$$

This is because $\delta \exp(\epsilon(y-1))$ is the greatest possible value for $\mathbf{Prob}[\boldsymbol{z} = y-1]$ and $\exp(\epsilon)-\exp(-\epsilon)$ is the greatest possible value for $\mathbf{Prob}[\boldsymbol{z_y} = y]/\mathbf{Prob}[\boldsymbol{z} = y-1]$.

If there exists an (ϵ, δ)-differential-privacy-preserving distribution D described by a weighted average α_y over all E_y for which it is true that there exists a Y such that for all $y < Y$, α_y attains its maximum possible value, and for all $y > Y$, α_y equals zero, by necessity this distribution is the unique global minimiser of the expectation among all such distributions.

The distribution $D_{\epsilon,\delta}$ meets these criteria, for which reason it is the unique optimum.

Note 5.4.2. It is important to note that this optimum is not among all (ϵ, δ)-differential-privacy-preserving distributions, but rather over all such distributions that also satisfy the additional criterion that they meet the ϵ-condition for every positive z.

[1] Note that one addition of $(\frac{\epsilon}{2k}, \frac{\delta}{2k})$ noise is still, in expectation, less added entries than k additions of $(\epsilon/2, \delta/2)$ noise. Querying tags in aggregate *is* cheaper, albeit probably not by a meaningful margin, against the background of millions of tag values that need to be transmitted regardless of noise.

The global optimum among all (ϵ, δ)-differential-privacy-preserving distributions is harder to compute, and may involve a Laplace-like distribution that is truncated on both sides, and not just towards the negatives.

Part II

System requirements

Chapter 6

Auxiliary interfaces

The FinTracer system has 4 core users: an AUSTRAC-side user, an RE-side user, an AUSTRAC-side administrator and an RE-side administrator. Most of this document deals with the interface provided to an AUSTRAC-side user. In this chapter we will deal with the other three interfaces.

All interfaces to the system are, as always, assumed to be remote interfaces.

6.1 Interface to the RE-side user

The reason an RE-side user is needed at all is in order to handle manual RE-side input to the system. The basic question that needs to be addressed is how to implement the FTIL command

```
target_variable = Read<variable_type>(description_string)
```

and what process is needed to support it.

As a reminder, this "`Read()`" command is what reads RE-side user input based on a description string provided by the AUSTRAC-side operator via an FTIL command. These commands can be executed (in parallel) on any subset of the Segment Managers (including the one on the AUSTRAC side, which serves here as an interesting special case).

In detail: AUSTRAC is providing here a description of the requested variable and the type of variable to be read, and as a result of action from the RE-side operator(s), this value is read and stored in the variable named "`target_variable`", which is a distributed variable. For each tracer in the command's scope, `target_variable` receives its value based on the input provided by the RE-side operator local to its node.

The key to implementing the interface to support this on the RE side is to make the file name (or, more generally, the location) in the Input Store where the RE-side user places the input data determined not by the RE-side user, but rather by AUSTRAC. Furthermore, it should be determined and communicated to the RE-side user not when the FTIL executor reaches the "`Read<>()`" command, but rather much earlier, at the time in which the RE receives the associated notice (or, otherwise, receives communication from AUSTRAC regarding what will be required for a given FTIL run).

The recommended solution here is to use in the "`description_string`" a URN-like identifier that can be used to communicate

- Which notice the "`Read<>()`" command refers to,
- Which object in the notice the command refers to,
- Where to place the data in the User Input area, and
- Any additional information available to the FTIL script that was not available at the time of notice writing, that the RE-side user needs.

Such a description string may read, for example, like so: "*NoticeID*:*ObjectName*#*Comment*", where the "NoticeID" identifies the relevant notice, the "ObjectName" refers to the name of the object as it appears in the notice, which will double as the name of the file where the user should place the information in the User Input area, and "Comment" is any additional information to be communicated.

The full solution flow now works like this:

1. AUSTRAC sends to the relevant REs a notice. In the notice, one or more objects to be input by the RE-side operator are described, including their required values, types, encoding and the name by which this object is to be referred to, which serves also as a file name.

2. The RE user creates the requested files. This can be done in any order or in parallel. When the files are created, they are uploaded (e.g., via "FTP put") to their designated place (i.e, with the appropriate file name) in the User Input area of the FinTracer system.

3. In parallel, AUSTRAC runs the FTIL script related to the notice.

4. When the FTIL script reaches a "`Read<>()`" command, it checks whether a file with the appropriate name exists in the User Input area.

5. If so, the FTIL executor reads the file, parses the data, and stores the information in the appropriate FTIL data type, in the designated FTIL target variable, which is physically stored in the Variable Store. If all this completes successfully, the FTIL executor deletes the user file from the User Input area and sends back to the Coordinator that the "`Read<>()`" was completed successfully. Once all relevant tracers report this, the coordinator determines that the read was successful and moves on to execute the next command.

6. If the file is not there, the system will log this event and will wait for the file to be created. This can be done either by polling or by "`inotifywait`"-type solutions. In this particular case, RE-side users will probably want the log message to trigger an immediate alert, e.g. by sending an e-mail message. Such alerts are, however, beyond the scope of the FinTracer system, and should be handled using the organisation's standard logging and alerting/event-management tools.

7. If a file by the appropriate name exists but its format is incorrect, probably the best way to handle it is for the system to rename the file, e.g. by adding to it a suffix that indicates its defective status. This will allow the system to continue waiting for the creation of a correct file, as per the previous case discussed. Again, such an event should be logged (and the logging is likely to be configured so as to trigger an immediate alert).

8. It is possible for the system to also provide an interactive RE-side interface, where an RE user can redirect the system to find the desired file in a different place, and possibly a different name, to what the FTIL programmer originally specified. Such an interface is Priority 3 at best, and is not expected for an MVP release.

9. Logging messages regarding successful data reads, as well as those regarding failed read attempts due to badly formatted input, should also be propagated to the AUSTRAC-side, where they will also be logged, and possibly also trigger alerts. This will help AUSTRAC troubleshoot problems, e.g. if a particular RE fails to upload data. The messages propagated to the coordinator node should note when a file is read successfully and when a file is read unsuccessfully, but should not reveal to the coordinator any information regarding the contents of the file in either case. For example, AUSTRAC should not be privy to any specific error messages resulting from a failed read.

This solution was chosen based on, among others, the following considerations.

- Elegance of the solution,

- Avoidance of unnecessary duplication of information between the notice and the FTIL script (with the potential of problems due to mismatches),
- Avoidance of unnecessarily adding multiple identifiers to the same object, and
- Avoidance of a restriction that may introduce unnecessary technical constraints.

Notes regarding this solution:

1. The insertion of a "comment" portion to the description string allows the FTIL script to receive input even of a kind regarding which the notice cannot provide full details. An example of this is the old and now-superseded path retrieval algorithm, where the script discovers the path by following it, account by account. (This was later superseded by the algorithm of [3].)
2. There is no set of circumstances where the method described above keeps the RE-side user busy waiting: if the user responds to the request before the FTIL query arrives, the process is completely non-interactive, and if the user responds after, the system is the party doing the waiting. Moreover, requests for data arrive in every case as soon as it is possible to request them, giving the RE-side user as much time as possible to respond to them.
3. In FTIL scripts, it does not matter where the "`Read<>()`" commands are. They may be at the start of the script or in the middle of the script. They may depend on each other or be independent. In all cases, the solution avoids the problems discussed above. As a result, the FTIL script programmer has as much freedom as possible in designing the script.

At certain times in the past, a potential second, and potentially unrelated RE-side user interface requirement has been raised, namely that REs may want to explicitly authorise particular scripts when they run. In other words, instead of just authorising a script for use in the system (as is done by script certification), REs will be able, under this requirement, to also authorise each execution of a script individually.

The easiest way to implement this is for the (certified) script to start with a command that asks the user to manually input a yes/no authorisation via a "`Read()`" command. This allows such authorisation to be given per execution of a script, but only for scripts where this is required, and with individual authorisation only given by REs that want to provide one (by running this "`Read()`" command under an appropriate scoping), and all this without any new feature required of the FinTracer system.

Because this requirement does not require any additions to the software of the FinTracer system, the question of whether or not such a requirement is supported and/or enabled can safely be left for late-stage policy discussions, and has no impact on the technical specifications that this document is concerned with.

6.2 Interfaces to the AUSTRAC-side and RE-side administrators

Note 6.2.1. Disclaimer: This section was written based on notes from a conversation with Alex Omilian. In its present form it is a stub.

It is imperative that administrators, both on the AUSTRAC side and on the RE side, will be able to administer their respective parts of the Purgles Platform. We expect this to be done through API calls. Some of these APIs will be in the form of Remote Procedure Calls (RPC), i.e. they are initiated by the administrator and a reply is received from the system. Others will be "push" notifications, where the system proactively notifies an external service of alert situations or status changes.

We expect administrators running FinTracer nodes, particularly on the RE side, to already have their own run environments, connected to their own tools for monitoring, automation, and

general command-and-control. The idea here is that the Purgles Platform will not provide its own specific tools for command-and-control, but will rather use standard APIs to plug into such existing tools.

For example, FTIL contains an "`assert()`" command, which may be triggered when an underlying algorithm encounters an alarm condition. Such an "`assert()`", if triggered, will trigger a corresponding push API. It is now up to the node administrator to configure their local monitoring tools to determine which asserts should be treated in what way. Some may be safely ignored, for example, while others will require an e-mail to be sent to the responsible person for further action. There is no intention, at the moment, for the Purgles Platform to connect directly to an SMTP service for the purpose of sending this e-mail directly. Rather, such actions will be delegated to the downstream monitoring system. This will allow an administrator to decide, for example, whether to use e-mail for such notifications or another system, such as Slack.

Elsewhere, e.g. for the purpose of creating monitoring dashboards for the system's operation, it makes more sense for the system's status to be polled by the relevant external system. There, the RPC-style API would be more appropriate.

For many types of information, both push- and pull-style queries should be supported. For example, for monitoring, as discussed above, one would want to be able to poll the system regarding whether it is currently running an FTIL job or not. This is an RPC-style API. On the other hand, one would also need a push API that will be used to indicate that the system has just completed an FTIL job and is now idle. Having such a push API allows an external system that wants to push automated jobs into the FTIL pipeline (e.g., for transaction monitoring) to manage its own queue of automated jobs, releasing a new job whenever the system becomes available.

The following are services which we believe administrators will need, and for which suitable APIs need to be developed. (Defining the exact APIs, and specifically which task requires which type or types of API, is presently TBD.)

1. Start and stop the system,
2. Pause and resume an FTIL execution,
3. Cancel job,
4. Do not accept further queries after the current one (in preparation for a planned shutdown),
5. Retrieve status info (Is there a job running? And if so, what is the script name, what was the query, and when was it started?),
6. Ability to deal with updates and new releases. (This should probably include also an ability to start the system in a standalone mode, for pre-connection testing.)
7. Set logging verbosity level, and ability to set log categories so that any log entry related to a particular, chosen FTIL query can be easily isolated,
8. Step-through mode,
9. Setting of logging thresholds (See **Section 6.4: Logs**).
10. Running of all or selected parts of a Built-in Self Test (BIST), which will include, among other things, testing connectivity between all local parts of the system and connectivity from and towards remote parts of the system.

In addition to these APIs that should be accessible to the administrator of each node, some APIs should only be available to the administrator of the coordinator node as they affect the entire system.

1. Query reset,
2. Do not accept further queries after the current one (Same as the corresponding RE-side API, but impacting all nodes),

3. Possible API: Support of a "test mode". This is a global setting, affecting all nodes in the system, that blocks nodes from accessing their Transaction Stores and Auxiliary DBs, and swaps these with synthetically-created test data.

4. On the assumption that Variable Stores are persisted: An API to clear all variables.[1]

Note 6.2.2. We also considered whether to allow a user to manually trigger a change to the seed of a node's random number generator. Such an API may be conducive, but under no circumstances should a user have access to or control of any random seed used by the system, as this would present a major hole in the system's cryptographic security.

6.3 Interface for script certification

In FTIL, commands like `transmit`, which transfer data between nodes, are examples of privileged commands and can only run as part of privileged mode execution.

How one gets into or out of privileged mode execution is described in Section 3.15. Important for this section is only the fact that determination of what to run in which execution mode is done on a per-node basis, and the per-node information available for the decision are the *certificates* that a module script may or may not possess.

In this section, we describe these certificates.

6.3.1 Delegates and certificates

Let us begin by reviewing the idea of *digital signatures*. A digital signature function is a function that can be applied on a digital object (or, often, its hash) together with the private part of a public/private key pair, to generate a keyed hash known as a signature. This differs from a standard keyed hash only in the fact that anyone with the public key can verify that the signature matches the original object. Signature functions are designed in such a way that it is cryptographically hard to create a new signature for a desired object without knowing the private key, and it is cryptographically hard to create a new object with the same signature, even having the private key.

In addition to its keyed-hash part, a digital signature also includes a plaintext part. This typically contains a cryptographically strong hash of the element being signed plus additional plaintext elements, such as the identity of the signer and the date range signifying the signature's validity. It is this entire plaintext part that is signed in the keyed-hash part. As such, the digital signature not only proves that the object itself was untampered, but also provides any additional context information required in order to interpret the signature (such as what was being signed, who signed it and when).

Note 6.3.1. By "cryptographically strong hash" we mean a hash function, h, for which, given an object o, it is difficult to find another object, o', such that $h(o) = h(o')$.

The FinTracer system does not store any private keys for the purposes of certification. However, each Segment Manager stores in its Auxiliary DB a set of public keys. Typically, one will be a *master* key and the others will be *delegates*. These are the public keys of authorities that the organisation running the node with the Segment Manager has entrusted to certify that particular scripts are safe to run. Collectively, they are known as the node's *certificate authority keys*.

The way that we envision it is that the master key will be some constant of the node that is very difficult to alter, whereas delegate keys can be added or removed by a simple administrator command. Specifically, in order to add a new delegate, it must, itself, be digitally signed (i.e., certified) by one of the keys already in the system. Thus, the RE owning the node will have its master key and will be able to use this master key to certify delegates, with each delegate able to further certify other delegates, recursively.

[1] A FinTracer user can perform this by a "`rm`" command. This is merely an API that triggers the same command.

As is standard practice for digital signature certificates, our certificates will not just identify the public keys that they are certifying, but also a date range for the validity of the certification. In this way, certification can be cryptographically voided if no longer relevant, and not just removed by an administrator.

6.3.2 Script certification

Separate to the certification of delegates is the certification of scripts.

Any module script (and only module scripts) can be certified (i.e., digitally signed) by one or more certificate authority keys. The result, referred to as a *certificate*, can be uploaded into a special table in the Auxiliary DB.

This can be done by an administrator command. We do envision, however, that certificate authorities will in many cases be external to the RE organisation, so the REs may want an automatic process to allow certificate authorities to independently upload certificates into their nodes.

A certificate certifies one script inside one module. It does so in the form of a digital signature, using a certificate authority key. The plaintext portion of the signature includes

1. A cryptographically strong hash (e.g., a SHA3) of the module file,
2. The name of the script inside the file,
3. The identity of the signer, and
4. The validity date range for the certification.

It indicates that this particular certification authority deems this particular script, of this particular module, safe to use, without requiring additional controls on how it is invoked. Technically, invoking the script will automatically elevate execution to privileged mode.

We refer to the hash of the module file as the module's *signature*. Where relevant, we take the signature to be represented as a lowercase, zero-padded, hexadecimal string.

Note 6.3.2. Eagle-eyed readers will note that the term "signature" is a bit overloaded here. Let us clarify our use of it. The general cryptographic technology known as "digital signature", explained in Section 6.3.1, is used in FinTracer for *certification* (a process of guaranteeing fit for purpose). This is similar to its use in "signing a document", where one ascertains one's approval of the contents of the document. By contrast, the cryptographic technology known as "hashing" (or specifically, in our case, "cryptographically-strong hashing") is used in FinTracer for *signing* (a process of guaranteeing that something is untampered), for example in the name "signed scripts". A *signature* (like in its usage in the term "biometric signature") merely means "something that uniquely identifies an object", and serves in verifying that the object hasn't been tampered with. There is no direct relationship between FinTracer signing and cryptographic digital signing.

At module load time (See Section 3.14), all certificates relating to the module are loaded from the Auxiliary DB (based on the module file's signature) and are checked for their validity. This is done separately at each peer node that is in scope.

Any script in the module for which a valid certificate is found by a recognised certification authority is at this point marked by the peer node as a *certified script*.[2]

Note that different peer nodes may have different certificates and different certification authorities, so the same module may have different scripts marked as certified scripts in each peer node.

[2]This is one way, and perhaps the most logical way, to implement this certificate loading. However, there is no functional requirement to implement it this way. Checking whether a script is certified can equally be performed when it is invoked.

6.4 Logs

The system will log meaningful events, including the details of any FTIL query execution.

Log entries will be categorised by verbosity level. Each log entry will have a time-stamp.

It will also be possible for an administrator to label specific log entries as pertaining to the execution of a particular FTIL query or to particular FTIL code.

The logs will include any status changes and information regarding connections or connection attempts with other nodes.

Administrators will be able to set thresholds regarding latency, dropped connections, so as to enact a meaningful policy of when these should be logged and when they should trigger various types of alerts.

Errors that are of a "named" nature will receive their own error codes.

Such named errors may also require thresholds and policy decisions. For example, thresholds may be set regarding how the system should react if, in the synchronisation of the propagation graph between nodes (See **Section 4.5.1: Constructing a two-sided graph**) an unexpectedly high proportion of graph edges is dropped.

While not every piece of data in the system needs to be logged, in some cases it is important to log FTIL variable data. Examples are:

1. Manually input data, and
2. Private data that is reported by an RE to AUSTRAC (i.e., query results).

We recommend in such cases to keep the actual data in the logs, but to keep it in an encrypted form, so as not to cause logs to become unnecessarily sensitive.

6.5 APIs

TBD

This section will provide the full list of lower-level APIs required to support the functionality of the system, including both the operator-centric functionality described in this chapter and the service-centric functionality described in Chapter 7.

Chapter 7

Additional services

Note 7.0.1. Disclaimer: The material in this chapter is a set of bullet-point headings that were mostly introduced by Alex Omilian.

The following are additional services the FinTracer system should connect with, to be fully functional.

1. Logging and auditing, including infrastructure to ship logs off of the FinTracer system instance,
2. System and job monitoring services,
3. Key management (for inter-node authentication and communication),
4. Configuration management (i.e., the ability to create container images and apply configuration properties). A sub-category of this is sequence management. Essentially, these services provide an API environment within which to run the APIs of the FinTracer system.
5. Identity management,
6. Identity authentication,
7. Services to persist variables, the Variable Store/Registry and/or the System Store,

Note also that the system will require a significant amount of random bits, but at the moment we do not see any requirement for an external random bit generator. Modern CPUs are equipped with true random bit generators that appear to satisfy our requirements (See **Section 9.5: Entropy requirements for the system**), so this generation is presently planned to be on-chip, and there is no plan to use any external service for this purpose.

If the system is to support periodic runs, such as for transaction monitoring, the expectation is for such runs to be triggered by an external process. Such a process may require its own set of services beyond what is listed here. Specifically, it may need to connect to a job scheduler, or may require some deployment management infrastructure.

Chapter 8

Nonfunctional requirements

Note 8.0.1. Disclaimer: The contents of this chapter were adapted from material originally authored by Alex Omilian.

Fully specifying the nonfunctional requirements (NFRs) of the Purgles Platform and of the FinTracer system is beyond the scope of this document. NFRs will be specified in discussions with the REs, and will probably need additional road-testing before they can be finalised.

In this chapter, however, we will discuss what will need to be included in the system's NFRs, and what will need to be considered in setting such NFRs specifically.

The non-functional requirements for the Purgles Platform cover a number of considerations.

- Minimum performance and availability expectations for Peer Nodes. Included to ensure a reliable and performant service is offered to participants.
- Maximum traffic expectations for peer nodes. Included to ensure there is a ceiling for the amount of traffic that a peer node operator is expected to service.
- Requirements for data latency and quality. Included to give a clear indication to the depth and recency of the data available to the platform.
- Limitations on the number of calls that can be made to a single peer node. Included to ensure there is a ceiling for the amount of load that a peer node operator is expected to support.

8.1 Availability requirements

We will ultimately wish to specify the availability requirements for peer nodes as "x% per month", where unavailability is defined as any period of time where any of the API end-points defined for each node is unable to reliably provide a successful response to an appropriately constructed request.

This availability requirement does not include planned outages. Planned outages should be:

- Conducted on non-business days, and
- Published to platform participants with at least one business day lead time.

Alternatively, planned outages may also occur without notification and on business days if the change is to resolve a critical service or security issue.

The rationale behind this availability requirements specification is that the system should allow "24×5" operation, which is a specification that balances the system's needs for long, uninterrupted up-times for long query executions on the one hand, and the node operator's needs for maintenance windows on the other. The "24×5" requirement effectively synchronises maintenance windows between all participants.

We do not see this system as requiring "high availability", and the figure of "$x\%$" should be chosen to reflect that.

8.2 Performance objectives

We will ultimately wish to specify the performance objectives for peer nodes as "$x\%$ of API requests in a given hour should be responded to, from receipt of request to delivery of response, within y milliseconds."

Here, API requests refers to low-level API requests, such as HTTP requests, and may be significantly lower level than individual FTIL commands. For example, an FTIL command that requires user manual input will not be penalised for not completing within a given number of milliseconds.

Having said this, API requests are not all equal, and will be divided into categories, where for each category a separate y will be defined. Some administrative calls may have higher priority and will require a lower y. Some calls may require a large data payload and will require a higher y. And some calls may require an extra-large data payload, beyond some predetermined threshold size z, and may be exempt from completion response times objectives altogether.

Note 8.2.1. Further work is required to fully map out all system functionality to API calls, and then to determine a classification for these calls.

8.3 Traffic thresholds

We will define traffic thresholds in the form of "x API calls per day" and/or "y FTIL queries per day". A node receiving calls in excess of these thresholds will be free to throttle or reject them without this impacting their performance or availability requirements.

8.4 Data latency

Each table to be populated in the Auxiliary DB, and the one table to be populated in the Transaction Store, will have its own data latency requirements. These will come in the form of "T+x days" from the relevant base date. For example, Transaction Store data latency will be measured relative to the business day of transaction settlement.

8.5 Data quality

Data holders will be required to take reasonable steps to ensure that the Purgles Platform data is accurate and up to date, to the extent possible, appropriate and relevant, given the purpose for which this data is used.

8.6 Disaster recovery requirements

The disaster recovery requirements of the system will be worded as "recovery within x weeks", where x will be chosen to reflect that we do not view this system as a high availability system and do not expect REs to support Active/Active architectures to support it.

A Peer Node is considered to be have been restored when normal service has resumed. Normal service requires:

- All API end-points are available and meeting performance objectives,
- The Transaction Store and Auxiliary DB have been restored to service, including the data to meet data latency and data quality service objectives,

- The Transaction Store and Auxiliary DB include the full schema and scope of data defined for each, including any data which was accumulated over time to meet all rolling window date ranges, and
- All application access and audit logs are restored, including all data which accumulated over time to meet agreed rolling window date range.

Note that there is no requirement for peer nodes to recover the "scratch pad" discussed in Section 2.5.2. The scratch pad will be, for all intents and purposes, owned and managed by AUSTRAC.[1]

8.7 Exemptions to protect service

In the event of the following extreme circumstances, peer node operators will be able to obtain relief from non-functional requirements:

- Periods of time when the peer node reporting entity are the target for a distributed denial of service or equivalent form of attack, and
- Periods of a significant increase in traffic volumes originating from another one of the FinTracer nodes, which may be misbehaving.

[1] Scratch pad contents are intended to essentially be materialised views, so can always be reconstructed given new Auxiliary DB data.

Part III

Additional discussion

Chapter 9

Infrastructure specifications

This chapter is about sizing the infrastructure required for the FinTracer system. At this time, this sizing is not yet complete. For now, we merely enumerate the dimensions in which the infrastructure needs to be sized. The data in each section is for now left as "To Be Decided" (TBD).

9.1 Compute requirements for the system

TBD

9.2 Storage requirements for the system

TBD

9.3 Communication requirements for the system

TBD

9.4 Utilisation requirements for the system

TBD

9.5 Entropy requirements for the system

TBD

Here we will specify our needs regarding total random bit throughput and total entropy throughput.

9.6 Benchmarking data

TBD

This section will contain representative examples that will map out indicative API end-point performance.

Chapter 10

Cryptanalysis

10.1 Differential privacy limitations

Note 10.1.1. This section discusses the limitations of differential privacy technology and of our use of it, as well as how to manage these limitations.

Refer to Section 5.4 for an explanation of the general concept of differential privacy, and the specific differential privacy algorithms in use within the FinTracer suite.

Whereas other privacy preservation mechanisms employed in the FinTracer algorithm suite rely on protocols that guarantee cryptographic-level protection to private data, differential privacy does not. It is a layer of noise added to returned results that is meant to obfuscate particular details of these results from prying eyes.

Such protection is not absolute. It is highly reliant on

- The nature of the data being protected,
- The nature of the protection we would like to provide, and
- The assumed usage of the system, including for malicious users.

Regarding the nature of the data, the underlying probabilistic assumption is that the information associated with a particular account that is based on any combination of sources is

1. Independent of the information of other accounts using the same sources (The *account independence* assumption), and
2. Independent of any information associated with any disjoint combination of sources (The *source independence* assumption).

Regarding the nature of the protection we would like to provide, differential privacy is meant to protect the privacy of individual people from being associated with particular groups. If S is a set of accounts defined by any criterion, it should not be possible to ascertain whether a particular person's account is in S or not.

This, however, is far from the only data-privacy concern. If, for example, S is the list of accounts associated with home loans, the set's sheer size is commercially sensitive information that each RE would love to find out about its competitors. Differential privacy injects noise into this measurement but, as discussed in Section 5.4, neither differential privacy nor any other reasonable measure can hide the general, approximate size of S, and in the context of counting home loan accounts for competitive intelligence the exact size, which is kept obfuscated, is immaterial.

Thus, the protection provided by differential privacy should be taken as very narrow, covering only the specific use-case of association-hiding, for the protection of personal privacy.

Finally, regarding the assumed usage of the system, we will see that the expected use of the system violates both account independence and source independence. We will, furthermore, see

that these violations are not merely theoretical, but can lead in practice to information being revealed, even unintentionally.

In Section 10.1.1, we will discuss how account independence is violated in practical use, and in Section 10.1.2, we will do the same for source independence. Finally, in Section 10.1.3, we will consider how these concerns can be managed in practice.

10.1.1 Differential privacy and propagation

Our algorithms for handling differential privacy keep meticulous track of the information sources used in order to construct the values of each tag, but not during tag propagation. At FinTracer tag propagation, the output tag does not inherit any of the original tag's source inventory, and, instead, becomes its own, brand new information source – or, in the case of summation to an existing tag, has such an information source added to its inventory.

This exception for tag propagation is inevitable: the source tracking is meant only for situations where values are independent between accounts, so becomes invalid if we try to apply it across propagation.

If a tag, x, is propagated using an operator M into a new tag, $y = Mx$, there are relationships between x and y that cross single-account boundaries, and causes a correlation between the values assigned to multiple accounts in y, contrary to our assumption of account independence.

In this section, we will show that our differential privacy protection mechanisms can break down because of such relationships in ways that can even contribute to practical exploits.

The following is an example of how tag propagation can be used to leak private information from a FinTracer tag.

Consider a tag x. There is some set of accounts, S, for which the tag values in x are nonzero. We wish to ascertain whether a particular person's account, p, is in S. Differential privacy preservation essentially means that we shouldn't be able to determine this. However, we will show that this is possible to do if one has access to the total number of positive tag values in multiple queries, even after the introduction of differential privacy noise. (As a reminder, this information is shared with AUSTRAC as part of both tag querying and non-linear tag manipulation, but the FTIL software implementation is such that the values are only used and immediately discarded, without ever being stored longer term, thus providing a software-level protection where the protocols alone cannot.)

Let us define the set $P = \{p\}$ by use of a `Read` query that pinpoints the single person's account whose membership in S we are trying to ascertain, and let the system compute the tag x' whose positive set is $S' = S \setminus P$.

Our differential privacy noise is set to prevent us from being able to determine whether there has been a change in any single element, so it should not be possible for us to tell whether S and S' are the same set or not, which is equivalent to determining whether $p \in S$.

However, let $y = Mx$ and $y' = Mx'$ be the images of x and y, respectively, as mapped by some propagation operator M.

It may be that, for example, there are 20 accounts that p transacted with that no account in S' transacted with. This can make the difference between $|y|$ and $|y'|$, if $p \in S$, as large as 20, which is a much larger size difference than our differential privacy noise was designed to conceal.

Increasing the noise to account for this, e.g. by multiplying it by the maximal outgoing degree expected from any account, does not help, either, because one can simply repeat the trick with $z = My$ and $z' = My'$ or variations thereof, accentuating the difference to any desired degree.

The same trick can also be applied when only a single tag is being queried. Consider the tag representing the set $S \cap P$. This is non-empty if and only if p is in S. If we merely propagate this tag through enough iterations, it will grow to any desired size, assuming it started off as non-empty. But if it started off empty, it will remain empty no matter how many times it is propagated, making the two cases easily distinguishable.

The above constructions are relevant not just for FinTracer propagation, but for any other algorithm that triggers account-to-account tag interaction, all of which violate the account independence assumption, and therefore erode the protections afforded by differential privacy.

10.1.2 Considerations when defining new sources

The assumption of source independence is problematic in the context of defining new sources, i.e. when AUSTRAC defines a set of accounts by use of a description (most likely, a machine-readable description in the form of a SQL query).

It is an inherent problem in all differential privacy schemes that if an attacker is given free access to query a system, by merely repeating the same query enough times they can erode the protections provided by the differential privacy noise simply due to the law of convergence to the mean.

We can protect against this on a technical level, by preventing the same source from being overly reused (which is what our "differential privacy budget" calculations provide), but if a user redefines the same set through a second, identical query, this cannot be discovered purely using software, because identical queries can be reworded in an infinite variety of ways.

A non-malicious user can create sources that are not identical but still quite similar by virtue of trying to refine a query, iteratively improving it. This is a natural process to be done in intel work, but breaks down our differential privacy guarantees.

Furthermore, while the above is (for a human) an easily-recognisable pattern of activity that can be picked up through the examination of logs, a much trickier situation, which is far more impractical to avoid, is that an FTIL user's query just happens to have significant overlap with *some* past query relating to *some* past investigation.

Consider, for example, an investigation that centres on funds being funnelled through gym operators. A user may legitimately want to start the investigation by creating a set defining all accounts related to gyms, and may do so without knowledge of a past investigation, surrounding the same sector, where another research created a set that was defined in the same way.

If these two sets are identical, they allow a person who can determine (approximate) set sizes that were revealed during the course of using both these identical sets to effectively increase their usage of differential privacy budget beyond the allowed.

If the two sets have minor differences, on the other hand (for example, because some gyms have been opened since the older set was created) then by considering the differences between them, a potential attacker can learn private properties relating to these new gyms.

In practical terms, it is impossible to maintain true source independence in a system that is used over a long period of time and has many users.

10.1.3 Managing the limitations

The problems described above are quite inherent in differential privacy mechanisms. We deal with them through a cascade of protection mechanisms.

First, it is important to understand that while differential privacy cannot provide absolute guarantees, it does reduce any potential information leakage problem to be the exception rather than the rule, and, just as importantly, narrows down the scope of what any fallback mechanisms that will need to deal with the remaining problems have to account for.

Second, we note that all leaks are only towards the AUSTRAC user, and never towards any other node. This limits the types of leaks that are concerning to us. For example, our example of commercial sensitivity of the approximate size of returned sets is less problematic when the leak is only towards AUSTRAC.

Beyond this, we take a two-tiered approach.

On the technological tier, while we cannot prevent violations of our assumptions or even monitor for them, we do construct our software in such a way that any statistics that potentially convey differential-privacy-violating information (all of which are general sum statistics regarding the total number of elements matching a query) are always utilised as needed and then immediately discarded by the system. There is no long-term storage of them. (The only exception to this is if the system aborts while the sum is still available.)

By and large, such data is never displayed to the FTIL user. (A user may be alerted that a set is too large to be retrieved, but such an alert can only be triggered when there is an actual

attempt to retrieve the set, meaning that AUSTRAC has a legitimate purpose to see its values, making any differential privacy protection for it moot.)

On a policy tier, we suggest that governance and oversight be used to make sure that the software controls of the system are never circumvented in an attempt to harvest query statistics over longer periods of time, and to discourage overly-similar repeated querying in any case. The analysis above should serve as a guide to focus the efforts of those performing such oversight: activities that are of specific concern and require special attention are

- New source creation,
- Tag propagation, and
- Aborts while performing non-linear tag operations.

Usage of the other features of the system does not require the same level of scrutiny, because up-stream security features already protect against them.

In summary, the differential privacy measures themselves are not superfluous, but they also do not provide an end-to-end solution. The entire design of the system, including its logging and auditing capabilities, consists of a cascade of measures that jointly provide the required level of privacy protection.

10.2 Refreshing and sanitisation

FTIL provides ElGamal homomorphic encryption as a built-in. The mathematics of ElGamal have already been discussed in Section 5.1, and particularly the two operations `Refresh` and `Sanitise` which are used extensively in Chapter 4 in order to erase superfluous information when sending data between parties.

Refreshing and sanitisation provide different kinds of information erasure, which are useful in different contexts. In this section we explain the kind of protection provided by each operation, when we use each and why.

Let us begin with a quick recap of ElGamal encryption.

In FTIL, two main types underlie ElGamal computation. The first is the `Ed25519Int`, which is a group of integers over a large prime, p. The second is `Ed25519`, which is an elliptic curve group. The two groups have the same size, and one can map from the former to the latter by use of an arbitrarily chosen generator element G from the `Ed25519` group. To map an `Ed25519Int` x into an `Ed25519` element X, we use the mapping $X = G^x$. We refer to this process as *encoding*.

We will continue to use lowercase letters for `Ed25519Int` integers and uppercase letters for their encoded `Ed25519` counterparts.

To encrypt a message m using a private key a, for example, we first encode it as M, then generate a random element x, known as a *nonce*. The ultimate ciphertext is (MG^{ax}, G^x).

As demonstrated in Section 5.1, this ciphertext can be computed even if one only has the public key $A = G^a$, but decrypting it (i.e., retrieving M from the ciphertext) requires the private key a itself.

This definition inherently makes the ElGamal encryption scheme reliant on the hardness of two problems:

- First, it should be cryptographically hard to compute x from G^x. This is known as the *discrete log* problem. If computing the discrete log is possible for an adversary, the private key a can immediately be derived from the public key A.
- Second, it should be cryptographically hard to compute G^{ax} from G^a and G^x. This is known a the *computational Diffie-Hellman assumption*. This assumption is needed because both G^a and G^x are available to an attacker (the former being the public key and the latter being a given part of the ciphertext), and knowledge of G^{ax} allows retrieval of M. This is, in fact, exactly how decryption works, because knowledge of a does allow one to compute G^{ax}. For

true cryptographic security, we will, in fact, require the stronger version of this assumption, known as the *decisional Diffie-Hellman assumption*, which is that this problem is hard even as a decision problem: given G^a, G^x and Y, it should be cryptographically hard to determine whether $Y = G^{ax}$.

Two notes regarding the above:

1. If an algorithm is found that solves the discrete log problem efficiently (such as is the case, for example, with Shor's algorithm for quantum computers), it can also be used to solve the second problem described, because finding ax from a and x is simple modular integer multiplication. The reverse, however, is not necessary: one can have an Oracle for computing G^{ax} from G^a and G^x without this allowing one to compute a discrete log.

2. The above description is technically *Additive ElGamal*, which is the variation we are using. In additive ElGamal, by the very hardness assumption on taking the discrete log, it is not possible to retrieve m from M. One can only rely on checking whether M comes from one of a limited number of expected m values, because one can always take any candidate m, encode it and compare with M.

For convenience, we will write ciphertext tuples in additive notation. This is to say, instead of writing the ciphertext as $(MG^{ax}, G^x) = (G^{m+ax}, G^x)$, we will write $[m + ax, x]$, where the square brackets signify that all tuple elements are encoded into the `Ed25519` group.

This, for example, allows us to conveniently write out homomorphic addition as follows: given two ciphertexts $[m_1 + ax_1, x_1]$ and $[m_2 + ax_2, x_2]$, it is possible to compute a ciphertext for $m_1 + m_2$ without knowing even the public key A simply by summing the two tuples together to form $[(m_1 + m_2) + a(x_1 + x_2), x_1 + x_2]$, which can be done by multiplying the underlying ciphertext elements: recall that a sum in this notation corresponds to multiplication of the underlying `Ed25519` group elements, which is the basic group operation.

The two homomorphic operations we will be using are summation of two ciphertexts, as described above, and the multiplication of a ciphertext by an `Ed25519Int` scalar, k. The way to perform such multiplication is by use of a double-and-add algorithm (which translates to a "square-and-multiply" on the underlying group elements), which can be performed in $O(\log k)$ time complexity.

10.2.1 Refresh

The `Refresh` operation for ElGamal is an operation that takes a ciphertext $[m + ax, x]$ and returns a new ciphertext $[m + ax', x']$. The encrypted message does not change, but the nonce is replaced by a new, independent, randomly chosen x'.

The way to perform a refresh is to generate a new ciphertext encrypting the plaintext message $m' = 0$, and then to sum this new ciphertext to the original. The new ciphertext is $[a\tilde{x}, \tilde{x}]$ and the summation result is $[m + a(x + \tilde{x}), x + \tilde{x}]$. By taking x' to be $x + \tilde{x}$, the problem is solved.

The reason refreshing is important is that it allows one to hide connections between multiple ciphertexts, each of which is visible to an adversary. For example, if Alice sends to Bob the ciphertext $[m + ax, x]$ and Bob returns the same ciphertext to Alice, an eavesdropper Eve will not be able to tell what m is, but will be able to tell that it is the same in both messages, which is an attack we want to block. The way to thwart such an attack is for Bob to refresh the ciphertext before returning it to Alice. In this new scenario, Eve observes $[m + ax, x]$ and $[m + ax', x']$, and has no way of knowing whether the messages in the two ciphertexts are both the same.

Similar situations also occur if Bob was to sum multiple ciphertexts, forwarding on only the resulting sum: unless the ciphertext is refreshed before it is forwarded, Eve will be able to reconstruct the relationship between the inputs and the output.

In the FinTracer propagation algorithm, we `Refresh` ciphertexts before any time that they are sent from one RE to another. The reason for this is that FinTracer propagation can create scenarios identical to the one described above with Alice and Bob.

For example, suppose that Alice and Bob are two REs, and that Alice manages two accounts, A and D, and Bob manages two accounts, B and C, such that in the FinTracer propagation graph A connects to B which connects to C which connects to D. For simplicity, let us assume that these accounts do not connect to any other accounts.

From Alice's perspective, the connections A to B and C to D are visible, but the connection from B to C is a secret that only Bob knows.

In a run of the FinTracer algorithm, Alice associates a tag value (which is an ElGamal ciphertext) with A and forwards this to Bob for B, Bob then transfers this tag to C and returns it back to Alice for D. If Bob does not refresh the tag before returning it to Alice, Alice will be able, from observing that the tag returned from Bob is the same as the tag Alice originally sent, to determine that the connection from B to C exists, violating our privacy guarantees.

Thus, refreshing is essential whenever a tag is transferred from one RE to another.

10.2.2 Sanitise

A completely different type of information protection is afforded by sanitisation.

To sanitise a ciphertext $[m+ax, x]$, one allots a random, uniformly chosen nonzero `Ed25519Int` scalar k, and computes the scalar product $[km + a(kx), kx]$.

The purpose of sanitisation is to erase from m all information other than whether $m = 0$ or $m \neq 0$. A zero remains a zero in the multiplication, whereas multiplying any nonzero value by k results in a new ciphertext encrypting a message that is uniformly random among all possible nonzero values.

Sanitisation is important because the actual value of m may reveal more information than is intended. In the FinTracer algorithm we seed a certain account set A with 1 values, then propagate these tags and measure their value on B. For the purposes of FinTracer, we wish to determine whether particular accounts in B are connected to accounts in A, but the actual value of m at each account in B conveys much more information than this: it provides the exact number of walks of the appropriate length in the propagation graph from A to the target account.

Knowledge of this number of walks would allow an attacker to reconstruct much of the underlying transaction graph, so it is important that AUSTRAC does not get exposed to this extra information. By sanitising all tag values before they are sent from an RE to AUSTRAC, we ensure that AUSTRAC receives no information other than the one desired bit: whether such walks exist at all or not.

Another concern that sanitisation addresses, which is not relevant for FinTracer propagation but may be relevant to other algorithms in the FinTracer suite, is that tag values can be from the get-go imbued with more information than just 0 and 1 values.

Normally, we begin with a set S of accounts that has a 1 tag value, and the rest have zero, but suppose we take three sets, S_0, S_1 and S_2, and initialise the tag as 1 if the account belongs to S_0, plus $1,000$ if it belongs to S_1, plus $1,000,000$ if it belongs to S_2. Without sanitisation, the ultimate tag value for an account s will be of the form $t_0 + 1,000t_1 + 1,000,000t_2$, which, if t_0, t_1 and t_2 are sufficiently small, can be separated into its (t_0, t_1, t_2) components by exhaustive searching. This allows running, e.g., three FinTracer queries in parallel.

Though the running of such queries in parallel is in some cases desirable, the problem with homomorphic encryption is that the parties performing the computations (i.e., the REs) are doing so obliviously, so cannot tell whether only one FinTracer query is being executed or three, and sometimes this can make a substantial difference for privacy, e.g. by skewing differential privacy calculations, or by returning more results than are allowed.

Whether or not sanitisation is appropriate needs to be decided on an algorithm-by-algorithm basis, because sometimes there are no privacy implications to knowing the true value of m, and packing more information into it can be beneficial, but at the moment all algorithms in the FinTracer suite require REs to sanitise all tags before they can be sent to AUSTRAC.

If nothing else, this is a case of erring on the side of caution.

10.2.3 Refresh with Sanitise

As described above, all tags are sanitised before they are sent to AUSTRAC. However, they should also be refreshed.

`Refresh` and `Sanitise` perform unrelated operations. From a ciphertext $[m + ax, x]$, `Refresh` produces $[m + ax', x']$, destroying the original information of x', but `Sanitise` produces $[km + a(kx), kx]$, destroying m, but retaining the original ratio of x to m, meaning that the information in x still remains, even though it is in obfuscated form.

To see why refreshing is needed even in the presence of sanitisation, consider a situation where a tag is sent from AUSTRAC to an RE, and then returned from the RE to AUSTRAC without visiting other REs (or else it would have been refreshed in the RE-to-RE transfer). For example, consider a tag that has been normalised by AUSTRAC and then sent back to its RE of origin, but later on, perhaps for another nonlinear operation, was sent from the RE back to AUSTRAC.

When AUSTRAC sends the normalised tag $[m + ax, x]$, it knows all of a, m and x. We will consider the case $m \neq 0$, as in the case $m = 0$ sanitisation and refreshing are identical.[1]

Now, AUSTRAC receives from the RE a tag $[m' + ax', x']$, which it suspects to be the product of sanitisation from the originally sent tag. To determine this, AUSTRAC needs to answer the question of whether there exists a k value such that $m' = mk$ and $x' = xk$.

The received message contains $G^{x'}$, and by decrypting with the private key a, AUSTRAC can also determine from it $G^{m'}$.

AUSTRAC can now check whether the new tag is a sanitisation of the old tag by testing whether $\left(G^{m'}\right)^{x}$ equals $\left(G^{x'}\right)^{m}$.

This process can also be used to check whether the returned tag was the sum of some specific set of tags that were sent by AUSTRAC, and AUSTRAC can enumerate over such possibilities, if their overall number is not too high.

In summary, for a party like AUSTRAC, which has the private key and can also gain knowledge of the nonce's integer value, x, sanitisation does not perform the information erasure task that refreshing does, so tags sent to it must be both sanitised and refreshed beforehand.

Note, however, that this construction is entirely reliant on the party trying to reverse the sanitisation process knowing the private key (which, in our design, is only applicable to AUSTRAC). A party that does not have the private key has in this scenario G^{a} and $G^{x'}$, and can use knowledge of m and x (if it has them) to compute

$$Y = \left(\left(G^{m'+ax'}\right)^{x}\left(G^{x'}\right)^{-m}\right)^{x^{-1}}.$$

The ability to determine whether the observed tag is the product of sanitisation from a known origin tag is therefore equivalent to the ability to observe G^{a} and $G^{x'}$, and then to answer the decision problem of whether Y equals $G^{ax'}$. This is a problem that we assumed is cryptographically hard, or else the entire ElGamal crypto-scheme collapses.

[1] In the example of tag normalisation, any message that is not $m = 0$ will necessarily be $m = 1$.

Chapter 11

FTIL practicalities

In this section, we come full circle from Chapter 3, and discuss FTIL's design principles, how it meets its goals, and what users can and should do to reap the most benefits when using the system.

11.1 Designing effective software in FTIL

For a language to be practically useful it must provide mechanisms to create reusable code, and in particular must allow modular construction of such code.

One challenge regarding this that is relevant for languages wishing to support long, interactive sessions is the question of static vs dynamic binding. In an interpreted language, binding is almost inevitably dynamic: when one references a variable or executes a function, one expects the most up-to-date definition of this variable or function to be invoked.[1] However, in a language prioritising code security, one normally prefers static typing: the precise semantics of a piece of code should be determined at definition time (or, where relevant, compile time), so that the code semantics can be fully examined and debugged at development time, rather than much later, when it is executed operationally. In such a scenario, one would want to avoid, for example, a situation where function `A` calls function `B`, but the definition of `B` may change between when `A` is defined and when `A` is executed.

FTIL aims to be a practical language that is interpreted, supporting long interactive sessions, but is also security-minded. It provides three separate mechanisms for creating reusable code, each with its own purpose, and each dealing in its own way with the static vs dynamic binding dilemma.

11.1.1 Modules

A module is imported into the FTIL session by use of

```
import "filename"
```

or

```
import "filename" to target_object
```

The content of "*filename*" defines the name of the module, via a

```
module modulename
```

[1] The term "function" is used here as a generic programming term, not in its FTIL-specific meaning. For FTIL, the corresponding argument is regarding scripts.

directive.

This triggers the creation of a module proxy `object` called either *modulename* (in the first version) or *target_object.modulename* (in the second version). This `object` is created via execution of the module's `__init` script.

Modules are an excellent way to create code that is reusable. A module script `A` calling another script, `B`, within the same module, enjoys what is effectively static binding, because both modules must remain constant as long as the module itself exists.

This allows scripts within modules to be programmed in a modular fashion.

Scripts from one module can refer to scripts from another module, but only indirectly. This is done using the same mechanism as the one used for referring to module scripts from the command line. Namely, anyone with access to a labelled `object` instance carrying methods that refer to scripts in module `M` can execute these scripts by invoking the methods.[2]

This can be thought of as a form of permission handling: by sending such `object` instances to scripts (not in `M`) as parameters, one empowers the scripts to invoke particular scripts in `M`.

More generally, when a script in `M` attaches a method to a returned `object` instance, it is by this creating an interface by which `M` can be accessed, so this interface can be managed. Scripts that want to use functionality from `M` can be provided with interfaces tailored to their specific needs.

The module's `__init` method creates the first `object` instance to be created by `M`, so its methods can be thought of as `M`'s public interface.

When a script is bound to an `object`, that binding is static: the `object`'s method will continue to refer to the same script even if the script's module is replaced by a newer version. When a script calls a method, however, that binding is dynamic: if the `object` is replaced, the code may invoke a different method.

This dynamic binding can be used in both module and non-module scripts in order to purposefully allow dynamic binding, opening the door to incorporating new functionality that may not have been programmed at the time that the calling script was programmed. (We see this in the example code in Chapter 4, with the example of FinTracer operators: code invoking FinTracer operators is, for the most part, oblivious to the question of what kind of operator is being invoked and how it was defined.)

On the other hand, module proxies can store references to other module proxies, and general `object` instances can store references to other method-equipped `object` instances. By doing so, `objects` can make sure that any functionality that they (dynamically) bind to cannot be changed later on, and the functionality invoked by them remains consistent across invocations and across different methods. (Module proxies do this to import all modules that are prerequisites for them, and the choice of prerequisite modules is fixed at `import` time. Any updates to the prerequisite modules after `import` will not change (and therefore will not disrupt) the functionality of the loaded module. If required, the module proxy itself can be deleted and re-imported.

In terms of permissions, individual peer node operators can certify individual scripts inside modules. This allows said scripts to run privileged commands. The idea here is that privileged commands are potentially unsafe, but a certification indicates that the node operator considers a particular usage context of the privileged command safe. The script provides this context.

In practice, such scripts are rarely standalone. They rely on input and output, and part of the determination that the script is safe is that the input is fit for the intended use. To determine this, `object` instances can be given a label. The label indicates what type of `object` instance this is, and what type of usage it is fit for.

Labelling itself does not have to be done inside a privileged execution context, but when an `object`'s label is confirmed while running in privileged execution mode, what is confirmed is that the label is strong, i.e. that it was created inside a privileged execution context. Thus, the labelling mechanism makes sure that the operator of the node using the `object` has attested, via certification, that the `object` really is fit for its use.

[2] In some situations, the `rm` will need to be strongly labelled for this to work.

Calling the methods of an `object` instance within a privileged execution context is also allowed, provided that the `object` instance's label has been confirmed (with the same safeguards as above).

The above allows the FTIL code designer to run modular code, both inside a single module and involving multiple modules, including the coding of multiple implementations for the same interface (indicated by a label). It also allows a code certifier to confidently certify such code.

Once code is certified to be safe, it can be executed from any context to which the user provides the appropriate interface `object` instance. So, an FTIL programmer can program any code, with any flow, using non-certified code to connect together certified code segments as building blocks. Such practice does not require any new certification.

One can divide modules in FTIL into two types, based on the type of capability they provide.

In Chapter 4, we concentrated on one type of module: modules that define new "safe" execution contexts, where privileged commands may be run.

Such modules are meant to be stable pieces of software that have undergone much vetting within AUSTRAC, are communicated to the REs, get vetted further by the REs and their delegates, and ultimately get their appropriate scripts certified, in order to allow them to run the desired privileged commands. Any change to the modules at this point will invalidate all their certificates, because it will modify the module's signature.

The other type of module is modules that are merely meant for internal AUSTRAC consumption: they are simply pieces of reusable code that needs to be shared and used by entire teams. Such modules do not need to have their code communicated with the REs at all. They are loaded like any module, but none of their scripts can be (or need to be) certified, and any labels they create are weak labels (meaning that their `object`'s methods can only be called from a non-privileged context, such as from each other or from the command line).

Thus, modules allow code reusability and code sharing even for code that is still somewhat in flux (e.g., new functionality is being added to it). This is ideal for code developed by an intel team as part of an ongoing investigation, in order to be used for the investigation itself. The module proxy, like any `object`, is retained from session to session.

Such code can evolve on a daily basis, as the project's needs evolve. Note, however, that when modifying a module script, the entire module needs to be re-imported to be reflected in the module proxy.

This supports a governance mechanism for such project code, where module files are stored in some version management system, external to the platform.

11.1.2 Non-module scripts

Non-module scripts can be written at the FTIL command line or loaded via an "`exec()`". They are almost as powerful as any non-certified script, but have the advantage that they follow a user from session to session.

Code that is programmed by a user for their own re-use is optimally placed in a non-module script. It has no "signature" that can be verified, but does not need such verification because it cannot be corrupted, intentionally or accidentally, by another user.

Nevertheless, we recommend such scripts to be placed in a file management system outside the platform, where they can be stored and versioned, and only uploaded via an "`exec()`".

Calling a non-module script from a non-module script is done via dynamic binding. However, given that these scripts were all written by the same user we hope that can be managed.

Calling module scripts from non-module scripts is done using a mediating `object` instance. By verifying the `object` instance's label, the calling script can verify that the `object` and its methods are all fit for the purpose being used, while still allowing the implementation advantages of dynamic binding.

11.1.3 Driver code

The final type of code supported by FTIL is the code that does not reside inside any script. This is code that is generated at the command line. Its main usage is as *driver code*: while the heavy

lifting will certainly happen inside scripts, FTIL scripts themselves are passive. They cannot act unless invoked. Furthermore, without sending them interface `object` instances as parameters, many scripts cannot function.

This makes FTIL only usable with significant amounts of driver code: modules need to be imported, `object` instances need to be generated from them, and these `object` instances are then sent to scripts.

While all this can be done interactively, we believe it shouldn't, and FTIL provides the command "`exec()`" as an alternative.

When invoking "`exec(URI)`", FTIL reads textual, file-like input from `URI` and executes it exactly as though it was written by the user at the command-line (but with command echoing).

This allows an FTIL programmer to write automatic driver code, for use and for re-use.

It also allows FTIL code to be viewed, vetted and approved as part of the general governance of the FinTracer system, before it is executed, and this is done in a way that prevents mistyping and similar errors that may create a discrepancy between the intended code and what is actually executed.

As before, the code itself is optimally stored off the system inside a version management framework, to allow later inspection.

11.2 Best practices for FTIL script libraries

This section introduces tips-and-tricks in the best use of FTIL.

11.2.1 Handling multiple investigations

In future, FTIL may offer support for user group permissions, which will enable isolation of permissions between different groups of users, each of which handles a different investigation.

In order to keep the FinTracer system as simple as possible for delivery of a Minimum Viable Product (MVP), however, the FTIL language currently does not support such mechanisms. Every user can see every non-script variable equally.

Up to this point in the section, we showed that FTIL naturally allows users to create and manage their own individual code, while investigations create collaborative artefacts in the form of variables. Many of these variables will inevitably store investigation data, but some may be `object` instances that also carry investigation-specific functionality. We have shown that from a user's perspective there is no ambiguity regarding what is an individual piece of code and what is a project artefact (all scripts in a user's namespace are their own, all non-script variables are shared), and both aspects can be managed well and governed well, separately.

Now, however, we ask the follow-up question of how (without the introduction of group permissions) do we keep multiple, simultaneous investigations from stepping on each other's toes.

There is no solution that will work here against malicious behaviour, but in terms of accidental toe-stepping, we suggest the following as best practice when coding in FTIL.

We suggest that each investigation, when starting off, will define a top-level `object` instance whose name clearly identifies the investigation it belongs to. Whenever the investigation needs to define new objects, instead of creating more top-level objects it should simply create the new objects as attributes of their root `object` instance.

This includes modules imported for the investigation: under the investigation's main `object` (which may be named something like "`investigation_a`") one may create a dedicated sub-object (e.g., "`investigation_a.modules`") and all investigation modules can be imported onto that. This makes sure that different investigations can keep track of the module code they use, and can even consistently use separate code versions between them without this causing clashes.

In this way, every variable name is naturally prefixed by the name of the investigation it belongs to, making objects from different investigations non-confusable, the global namespace is not unnecessarily cluttered, and when an investigation reaches its end all of its variables can be removed using a single `del` command.

For the individual user, typing of the investigation name only needs to happen when writing driver code. Within any script, module or non-module, the identity of the investigation is encapsulated by the choice of parameters sent to the script, so is invisible inside the script itself. (In fact, scripts can be easily reused from one investigation to the next, where appropriate.)

11.2.2 Safe parameter passing

As already shown in this section, when one script calls another script in FTIL this is usually done with either the advantage of static binding (the calling script knows exactly what code the called script will execute) or of late binding (the calling script can and does ascertain that the called script is vetted to be fit for this specific purpose).

These properties allow the FTIL programmer to construct safe code. However, despite these safeguards that are built into the language, it is never a bad idea to write your code defensively where you can. Specifically, suppose that your script runs the command

```
OtherScript(my_variable)
```

There is no built-in guarantee that `OtherScript` will not tamper with `my_variable`. Even if `my_variable` is immutable, `OtherScript` may simply re-assign it, potentially even changing the original variable's type.

FTIL does not have features to allow scripts to declare that they do not tamper with specific parameters, nor does it offer built-in mechanisms to verify that they don't. (Such may be added to the language or built on top of the language at a later date, but are not part of FTIL's MVP.)

Instead, here are some tips for the FTIL programmer regarding how to structure their code so as to minimise, and in some cases avoid entirely, the issue.

The main recommendation is, if you want to keep `my_variable` safe, don't pass it to a suspect script. An alternative would be to pass a copy, instead, like so:

```
_my_variable_copy = my_variable
OtherScript(_my_variable_copy)
```

Because of FTIL's semantics, such copying merely adds a reference to an existing variable and performs no data copying. If `OtherScript` refrains from tampering with `_my_variable_copy`, as it should, no data copying will take place at all, making this process add only negligible overhead.

Passing a copy ensures that `my_variable` cannot be swapped out for another variable, regardless of `my_variable`'s type. For variables `my_variable` that are of elementary type, it further ensures that their values cannot be tampered with: any attempt to do so will trigger a copy-on-write.

Strings and pseudotypes are immutable, so cannot be altered to begin with. This leaves only the referential types. Unfortunately, FTIL does not allow one to guarantee that the attributes of one's referential types will not be altered if one sends them to a suspect sub-script.

One potential way to avoid this issue is to prefer sending elementary types where possible, to scripts that are suspect: by sending (copies of) the necessary attributes rather than the parent variable, immutability is guaranteed.

This is generally possible to do unless the called script needs to validate the sent collection's label. Where this is the case, however, the sent parameter is a labelled collection, and so protected from tampering unless through a `modifying` clause.

These clauses being prominent in the code and subject to the highest scrutiny when vetting for certification or when determining their viability for execution in privileged mode, we believe that the sending of a labelled collection as a script parameter requires no additional defensive-programming mechanisms.

On the flip side, a calling script in FTIL does need to be programmed defensively against bad parameters. This is more important than defending parameters against a bad script, because, as noted, in FTIL scripts can be vetted. A script, on the other hand, has no control over what parameters are sent to it, and FTIL does not even guarantee that they are of the correct type.

(This is with the exception of module scripts that do not get externalised, which know exactly what is sent to them and do not need to take quite so defensive a posture.)

Clearly, before parameters can be relied on, they need to be inspected. Inspection my include, depending on the needs of the script, running "`initialised()`" in order to verify that the parameter is associated with an entity, testing entity types, testing entity scopes, `assert`ing regarding the parameters' values and confirming the labels of collections.

One thing that does not get tested through all of the above is whether multiple parameters to a script are different names to the same variable. FTIL does not provide direct means to test this, for which reason programmers should employ the following defensive idiom: output values should be constructed in local, temporary variables, and only copied out to their ultimate output variable destinations (with no data copying involved) after all computations are done and the script no longer relies on any input parameter values.

The reason for this is that when assigning to an output variable, one may also be overwriting input variables, should these be different names to the same variable.

Note 11.2.1. Many times such extra caution is not needed. For example, if two parameters are of distinct types or are collections with distinct labels, they are not going to be different names to the same variable, so the problem is moot.

11.2.3 Setting collections' scopes

One of the more problematic features of FTIL is the fact that collection attributes that are themselves of a collection type must have exactly the same scope as their parent collection.

This is often not the most desired functionality. Consider, for example, an `object` like the FinTracer key manager, `keymgr`, of Chapter 4. It would be nice if one such key manager could be created at the widest possible scope, for it to be able to provide key management services for all other FinTracer `objects` in the system, regardless of their definition scope.

Unfortunately, such a capability would have led FTIL into multiple issues that would have required many new features to fully harness, which we wanted to avoid for an MVP. For example, suppose that only one `object` remains that refers to the key manager, and it does so at a smaller scope—how should the key manager `object` handle such a situation? Unlike distributed elementary variables, for an `object` the original definition scope matters: its label refers to a global fit-for purpose, not just any node-specific property, so the key manager's scope cannot simply "shrink", and nodes that have no references to it must still approve any modification applied to it using a `modifying` clause.

Moreover, once the remaining reference to the key manager `object` is deleted, this should trigger deletion for the key manager `object` itself, but if `objects` don't see all references to themselves, it is very difficult to handle garbage collection properly: no single party can tell when a variable is orphaned.

The semantics of handling such situations, and the technicalities of referring to such an `object` from the code itself, made us choose not to incorporate this capability to FTIL.

Somewhat less semantic problems and somewhat more usability problems would have been caused had we allowed FTIL collection-type attributes to have a *smaller* scope than their parent collection. One main issue to keep in mind in this context is that presently in FTIL back-pointers are easy and safe to produce: if `object A` refers to `object B`, it is always allowed for `object B` to refer to `object A`. Allowing only reducing collection attribute references would have broken that invariance.

For all these and other reasons, we decided that collection attributes in FTIL must always have exactly the same scope as their parent collection. To the FTIL programmer, this means that collections of different scopes have essentially no method of interacting meaningfully.

Our recommendation to the FTIL programmer, in light of this, is to always define every collection at the widest possible scope, even if all attributes of said collection are defined at much narrower scopes. This will ensure interoperability between collections.

Note that this does not exclude entirely the ability to create collections with other scopes, if one is aware of the limitations this imposes. For example, the script `TransmitTagSetValues` in Section 4.10.3 creates a `catalogue` named `accounts_cat` whose scope is only the peer nodes. There is no problem working with this `catalogue` within its defined scope, and it is even returned to a different script, `NormaliseTagSet` for continued handling.

The reason all this is possible is because `accounts_cat` is a very transient collection: it is created, used and then discarded without having to exist at any point independently of any script. Its host scripts are aware of its scope and handle it accordingly, and it never needs to interact directly with any other collection.

11.3 Useful idioms

The previous sections in this chapter already included some useful idioms that the FTIL programmer and FTIL user can utilise to get more from FTIL.

Here are some other useful idioms, for various programming purposes.

11.3.1 Dual scoping

One of the differences between module and non-module scripts is that when running module scripts peer nodes have more visibility, and therefore also more checkability, regarding what code is being run. This is usually desirable, but not always.

Consider, for example, the now-deprecated path-finding algorithm. This algorithm visits each account on the path in turn, and receives from the peer node managing the account enough information to find the next account.

Running such a script as a module script without further precautions is risky, because each node is aware of what code is being run on all other nodes. Hence, even though node A does not know what account on node B is being queried, it does know that the account being queried is on node B, which can be enough information for it to infer the identity of the account, thereby gaining undue knowledge regarding suspicion formation at accounts that do not belong to it.

The following is a useful idiom to allow such code to run even as module code, without this private information being revealed.

```
for _n in scope():
  on(_n):
    if _n == BranchToBank(account_id.bsb_num):
      MyConfidentialScript(account_id)
    else:
      MyTimeWastingScript()
```

The code does not attempt to run a single command, in parallel, across all nodes. Rather, it loops over each node in turn, and determines for each node, separately, whether it is the node that should now be executing `MyConfidentialScript`. Each node sees that the other nodes execute their individual "`on()`" scoping clauses, but does not know the outcome of the underlying "`if`" conditional check. When the conditional is checked for some node `_n`, because the "`if`" is handled by the Compute Manager, `_n` only sees the Boolean result of the conditional, not the details of the computation leading up to it (e.g., the account's BSB).

The reason for providing an "`else`" clause is so other nodes will not be able to determine regarding a node whether it executed meaningful code or not by timing its responses. The "`else`" leads each node to execute a code segment that has no effect on the data, but is identical to "`MyConfidentialScript`" in terms of non-data observations (e.g., time to complete, energy to complete).

11.3.2 Pre-confirmation

In Chapter 4, we demonstrate the idiom

```
obj = _temp
del _temp
label(obj, "My Label")
```

This is an important idiom to know: after finishing the construction of an output collection in a temporary collection `_temp` (regarding the reasons for which, see Section 11.2.2), we want to copy the value to its final destination, the output parameter `obj`. If the output is to be labelled, the operations above should preferably be performed in the order described: deleting `_temp` before `label`ling ensures that no copy-on-write is triggered during the labelling process, and keeping the `label` as the last operation allows `obj` to be label pre-confirmed when it is returned to the calling script, saving a usually-quite-superfluous label re-confirmation operation.

If, for any reason, we do not want to pre-confirm the returned value, the way to express this is

```
label(_temp, "My Label")
obj = _temp
```

Here, too, no data copying takes place. Also, there is no need to explicitly delete the "`_temp`" variable if this is the end of the script, because at the end of each script all script-local identifiers are deallocated in any case.

11.3.3 The versatility of catalogues

A `catalogue` in FTIL is a mapping from integers to `object` instances, and it is the only way to enable iteration through an unbounded number of collections.

On the face of it, while having a `catalogue` is better than not having a `catalogue`, the power provided by `catalogue`s nevertheless seems much smaller than the versatility afforded by container types, where one may want to use `dict`s, `list`s or `set`s, each of which may map to any fixed-sized type (and `dict`s also *from* any fixed-sized type.

However, the power of the `catalogue` type is not far behind all of the above.

Its native use is as a dictionary. True, it can only map from integers, but, as discussed in Section 3.11.9, almost every FTIL fixed-sized type can be converted to and from an `int`. Even for types that do not have such a conversion, it is possible to use an auxiliary `dict` to map the desired keys, of whatever fixed-sized type, to integers in order for these to be used as `catalogue` keys.

Implementing list-like behaviour with a `catalogue` is even easier, because the integer indices to the `catalogue` can directly be used as list keys (i.e., as positions).

Most interesting is how to implement set-like behaviour with a `catalogue`. This is exemplified, e.g., in the account set source tracking catalogue "`sources`" in Section 4.3.1.

To implement this, given a `catalogue`, `cat`, and an `object`, `obj`, the way we place `obj` in `cat` is

```
cat[obj.metadata.id] = obj
```

This ensures that all `object`s are all consistently mapped to the same `catalogue` positions. (An `object` may still change its `id` when it is labelled. For the purpose of the use of a `catalogue` as a set, a pre-labelled `object` is not the same as the result after labelling it. We generally recommend to populate `catalogue`s by labelled `object`s where possible.)

On the range side, `catalogue`s only map to `object`s, not to general variables. However, to map to any other type, merely map to an `object`, `obj`, and have `obj` contain an attribute, e.g. "`obj.member`" that is a variable of the desired type.

11.3.4 Wrappers for "modifying" scripts

It is a useful practice to write wrapper scripts for every script that features a "`modifying()`" clause, as exemplified in Section 4.4.1.

This means that if one has a module script

```
def MyScript(argument1, argument2, argument3):
  ...
  modifying(obj):
    ...
  ...
```

that features a "`modifying()`" clause, as shown, one should typically also provide a wrapper script

```
def MyScriptWrapper(argument1, argument2, argument3):
  MyScript(argument1, argument2, argument3)
```

in the same module, and make sure that only the wrapper script, in our example `MyScriptWrapper`, is ever externalised or called by any other script, and never `MyScript` directly.

The reason for doing this is that it gives all certifiers the option of certifying the wrapper script `MyScriptWrapper` rather than `MyScript` itself, and by this to indicate that while they believe that `MyScript` is a safe script with no need for any further context, it is only that when it is invoked in such a way that its modification of (in the example) `obj` is done at the full scope of `obj`, ensuring that whatever modification is made is done with full visibility and consistency across that entire scope.

11.3.5 Scoping and modifying

When given the option, always prefer to write

```
modifying(obj):
  on(scope):
    ...
```

rather than

```
on(scope):
  modifying(obj):
    ...
```

While the action performed by the two is the same, the permissions may not be. The former (and recommended) version is one where changes only occur inside a narrow scope within `obj`'s definition, but the script itself and the `modifying` clause inside it may be executed at a wider scope, meaning that more nodes are aware of the process and can control it via certification.

As a result, if in the first variation the `modifying` clause is run at the full scope of `obj`, this will require less permissions: it will not require each node to have individually certified the specific module script that is being executed.

The same is not true when reversing the order of the commands.

11.3.6 Ensuring coherency with "modifying"

Consider the definition of `TagExpendBudget` in Section 4.4.1. It contains the clause

```
  modifying(tag.sources):  # Used here to ensure an all-or-nothing update.
    for _i in tag.sources.keys():
      modifying(tag.sources[_i]):
        on(_peer_ids):
          tag.sources[_i].epsilon += epsilon
          tag.sources[_i].delta += delta
```

The use of "`modifying(tag.sources)`" here is a useful idiom: no actual modification is done to `tag.sources` inside this clause. The purpose of having the outside "`modifying`" is to provide a "save point" for the system to roll back to in case there are any problems.

Without this clause, failure may cause the update of the current "`tag.sources[_i]`" to be rolled back, but other elements of `sources` could be left already updated, and others may not have been touched yet at all. The top-level `modifying` clauses forces the update to be all-or-nothing, which is what is really needed in order to ensure the semantic correctness of `tag.sources`.

The use of `tag.sources` as the argument for this top-level clause is partially justified by the above, i.e. that the clause protects the semantic correctness of this particular collection, but in terms of functionality it is also prudent to not use just some arbitrary collection as the `modifying` argument in such a case: ideally, the argument used for the outer `modifying` clause should not add more restrictions to the execution that otherwise would not have been relevant.

In the specific example above, `tag.sources` is a good choice because

- It is known to exist and has already been label-confirmed (Both it and its attributes have been label-confirmed by the same command, in fact); and
- Its scope is guaranteed to be the same scope as its attributes, thereby guaranteeing no change to the certification requirements for the script.

11.3.7 Narrow scope functionality

Prefer to write code, where possible, so that unless there is a reason to prevent it from executing at a scope narrower than the scope of variables being manipulated, it should execute properly at such a scope.

One important tool in doing so is the use of sub-assignment ("`[] =`").

If one performs

```
a = b
```

at a scope narrower than that of `a`, this causes execution to abort, but

```
a[] = b
```

updates `a` only in the nodes that are in scope, without this causing an error or affecting any of the other nodes.

A real usage example can be found in the script `TagReduce`, in Section 4.4.1.

Note also in all examples in Chapter 4 that code never assumes what scope it is executed in. Rather, scripts check "`scope()`", and, where necessary, define a narrower scope by removing specific nodes that they do not want to execute particular commands on.

11.3.8 Scope of output variables

FTIL syntax does not restrict one much in how one's scripts may treat the parameters they are sent, so FTIL users need to rely on documentation and conventions in order to understand what to expect.

A convention we have found useful in the course of writing Chapter 4 is that (unless there are good reasons to do otherwise), when a script creates new variables and returns them to a caller, those new variables should preferably be created at the same scope as the scope of the script that created them.

This is not just a handy convention, but also increases the usability of one's scripts, because it delays the choice of when the scope of the created variables is determined from script programming time to invocation time, thereby granting extra flexibility.

11.3.9 Coordinator in scope

Continuing the point of the previous section, the question of whether the coordinator node itself is kept in scope or not is one that usually requires separate consideration.

In terms of the execution itself, all commands need to run at whatever scope the privacy preserving protocols being implemented require them to run. The question considered here, however, is, if all of a script's commands need to run at a scope that does not include the coordinator node, whether to scope the coordinator out from the call invoking the script or whether to invoke the script with the coordinator still in scope but then, within the script, to remove the coordinator from the scope of all commands.

Though different situations call for different solutions, all else being equal we suggest to prefer scoping the commands rather than the script. The reasons for this are

- Scoping out the coordinator node has significant impacts to the semantics of the script, e.g. in terms of what flow control it can do. In particular, scripts that do not have the coordinator node in their scope cannot accept string parameters, and
- If we want the called script to be certified, it needs to be safe regardless of how it is called. If its function requires commands to run without the coordinator node in scope, only by the script scoping the coordinator node out itself can it ensure that this has been done properly.

11.3.10 Key seeding

Cryptographic keys are in FTIL non-serialisable types. The reason for this is that once a type is serialisable, no protection is given by the language itself regarding how it is distributed. It may still be the case that the operators of the coordinator node are careful not to invoke commands that distribute the type and that the operators of the peer nodes are careful not to certify any such commands, but having a built-in guarantee by the language provides a powerful additional layer of protection.

Consider, for example, the key used to encrypt all FinTracer tag values in Chapter 4. Had that key been serialisable, nothing technical would have stopped a coordinator-side user from "`display`"-ing it, after which its distribution would have been completely outside the control of the FinTracer system.

As another example, consider a key that is shared between two peer nodes, A and B. Had such a key been serialisable, the operators of node A would have needed to trust the operators of node B not to certify operations that distribute the key onward.

While these are all examples of the importance of built-in, technical protection of cryptographic key types from distribution, some protocols actively rely on such distribution, e.g. requiring keys to be generated at one time, but then distributed on at a later time.

To support such protocols without any unnecessary weakening of the language's security controls, FTIL provides for every cryptographic key type the ability to generate it deterministically and in an entropy-preserving way from a seed value. For example, `ElGamalKeyFromSeed` generates an ElGamal private-public key pair from an `Ed25519Int` seed value. (Possibly, the seed value is directly used as the private key.)

This allows the FTIL programmer to use non-seed-based key generation for algorithms that do not require it, allowing them the extra layer of key protection, while enabling protocols that require later key distribution to work, simply by retaining the seed and later reusing it in another node.

FTIL programmers are encouraged to use seeding only when required, and even then only in the safest ways possible, in terms of built-in language protections. For example, if a seed does not need to be stored (which is sometimes the case), it shouldn't be, and then the idiom to program this should be as follows (using ElGamal functionality as an example).

```
seed = RandomEd25519Int()
... # manipulate seed as necessary here.
```

```
obj = object()
ElGamalKeyFromSeed(seed, obj.pub_key, obj.priv_key)
del seed
label(obj, "My New Label")
```

The two main elements to notice in this idiom are

1. The seed is generated from randomness in a way that ensures that the key ultimately constructed has full entropy, and

2. The seed is deleted (along with everything else that doesn't require continued storage which may provide clues to the value of the private key `obj.priv_key`) before the key's `object` is labelled. When performed in a script, this ensures that the key is not declared "fit for purpose" before clean up of its creation process has successfully concluded.

It is in general a good idea to wrap keys by managing `objects`, as demonstrated in Section 4.1.3 with the FinTracer key manager.

11.4 How FTIL meets its design goals

We introduced FTIL as a language whose design criteria include functionality, extendability, privacy-by-design, understandability, efficiency and implementability.

Now that FTIL has been described in some detail, it is time to return to these criteria and assess how well FTIL satisfies them.

In some cases, it is either evident, or at least directly provable, that FTIL meets its design goals. The following are examples of these.

Functionality: Proving that FTIL delivers the functionality we require for the FinTracer system can be done explicitly, by providing FTIL code that implements the various algorithms presently intended for the system. We provide such code in Chapter 4.

Extendability: The basic functionality provided by FTIL includes generic input and output capabilities and processing capabilities such as mapping, reduction, joining and filtering, which are well known (e.g., from the database world) to provide universal data processing functionality. Communication patterns provided by "`transmit()`" are entirely general. And any cryptographic primitives we may wish to make use of in privacy-preserving computation are part of the standard library (or can be added to it, if need be). This makes the language future-proof. As an example, all past variations of FinTracer are equally implementable in it, as are the many new algorithms that joined our algorithm stable since the writing of this document began. What we do not think of as a closed part of FTIL's description is the list of cryptographic primitives available in its standard library. Questions of whether, e.g., BFV full-homomorphic encryption (FHE) should be implemented are ones we purposefully left open. We did not include FHE for now, because our current algorithms do not require it and because we wanted to avoid dependencies on external software such as Microsoft's SEAL library. However, it is easy to reverse course on this decision and add extra primitives to the language at any time should these become needed. Such additions are currently at Priority 3.

Efficiency: The core tight-loops that are run in FTIL, where processing power is needed, are all data parallel and implemented using the SIMD syntax introduced in Section 3.12, and, as also explained there, for maximal efficiency FTIL allows such syntax to run over primitives such as arrays and memblocks, where the memory structure is known to the executor and can be optimised. For the purposes of the present project, we don't need to optimise every

possible form of the mass operations (although that would certainly be possible in a just-in-time-compiled language), but we do know which SIMD structures are actually used inside our algorithms, and can make sure those are executed in as efficient a process as possible.[3]

This leaves three criteria regarding which it is less obvious how FTIL satisfies them. The sections below provide an analysis.

11.4.1 Understandability: Can FTIL be read?

FTIL is a fairly restricted language, and the mass operations are reminiscent more of database SELECT statements than of a standard procedural language, so a person reading Chapter 4 can legitimately reach the conclusion that FTIL isn't the most read-friendly language out there.

However, we claim that that would be a conclusion based on a false premise. Namely, it is based on the notion that the code in Chapter 4 is typical of a user's interactions with the language. In fact, the opposite is the case.

The code in Chapter 4 exemplifies the tight-loops that require the most optimisation in the FinTracer algorithm, and—as all highly-optimised code in any language—are therefore harder to read.

However, that is not the typical interaction with FTIL. Instead, these code snippets are parts of scripts. The purpose of these scripts is to generate (labelled) `object` instances, themselves containing both data and scripts, and these `object` instances will provide all the functionality the user needs in a highly encapsulated manner.

From this point on, the user's interaction is not directly with the language, but rather with the functionality provided by the `object` instances, using `object` method syntax that does not require the user to know anything about the `object` instance's contents or its underlying implementation.

For example, it is possible in a single user command to generate an `object` instance that performs a FinTracer step. In Section 4.11 we provide such a script that takes as parameters (1) a date range, (2) database queries for sets of interest and propagation rules, and (3) the radius to search in. Base on this very simple query, the command constructs an `object` instance that knows how to perform the corresponding FinTracer step, along with all the bells and whistles involved such as the definition of a two-sided viewable graph (with appropriate inter-node signalling), the definition of the background set, account whitelisting, tag propagation inhibition, transaction exclusion, etc., and is able to determine whether the execution can be performed in any of three possible compaction modes, and will automatically execute the appropriate one including any setup-phase communication for optimising later sending. It then uses this `object` to execute the FinTracer query, and ultimately retrieves the results, implementing any required differential privacy mechanism.

This is quite a lot of functionality—for which reason the implementing code, which is not long, is rather dense—but from the perspective of the FinTracer user, this is a single command, with very simple inputs and a very simple output: the set of accounts of interest.

For a user attempting to match typologies, the internals of such `object` instances doesn't matter. It is straightforward to combine them in multiple ways in order to look for a wide variety of typologies. Examples of such combinations are given in the operator arithmetic examples in Section 4.6.

In short, FTIL distinguishes between two types of primitives: the lower-level primitives such as memblocks, arrays, lists and dictionaries (within which powerful functionality can be programmed in succinct ways and executed efficiently) and higher level primitives (that encapsulate functionality in very user-friendly ways).

So, for the end-user, reading FTIL is not a problem.

There is, however, a separate question of how readable FTIL is for the power-user. Specifically: we will need FinTracer's basic functionality to be packaged as module scripts. Will the people

[3] The core bulk operations in the basic FinTracer algorithm are "`rm`", "`rm`", and the mass operation "`rm`". More advanced algorithms also heavily utilise "`rm`". If these are performed optimally over memblocks and dictionaries, the rest, while not negligible, will still succumb to the 80-20 rule.

validating the code and providing certifications be able to read it?

Here, the code in Chapter 4 is likely to be much more representative.

To answer this question, we suggest that, in order to certify our code, certifiers will need to go through three steps.

1. The algorithm will need to be validated. This will most likely be performed over a document explaining the algorithm and providing pseudocode, rather than over the code itself.

2. It will need to be validated that the code implements the algorithm exactly. If the code in Chapter 4 is representative, this should not be difficult to do. Each line of pseudocode is transformed into one line or a handful of lines of code, and the rest of the code in the examples is, once that is mapped out and stripped away, easily readable management code. We believe that FTIL is close enough to pseudocode to make our algorithms eminently readable for this part.

3. It will need to be validated that the code cannot be abused in any way. For this, the language provides multiple safety features, including labels, metadata, asserts, etc., which, together with the fact that the code runs in single-process mode, makes what it does and what it can do quite checkable. (If one assumes that the FTIL executor has already been pre-checked and is known to perform as advertised.)

All in all, we believe FTIL 'meets and exceeds' on the criterion of understandability.

11.4.2 Privacy-by-design: Governance and the FTIL security model

Part of the plan for the Purgles Platform is to get external review for the FinTracer system before it can be deployed at RE sites. While at face value such a step looks natural (and, indeed, unavoidable), a closer look leads us to the question: "What is it, that this external review should verify?"

In particular, if extendability is a core requirement of the system (and given that we still have brand new algorithms added to our stable on a weekly basis, this certainly is a requirement) it seems that we are asking such reviewers to validate not what the system presently does, but rather what it *can* do, if it is extended to its limits. Unfortunately, it is the nature of computing systems that even minimal extendability makes them, in a sense, "universal", and that is certainly the case here—no lesser extendability would have been in any way meaningful.

So, how does one give a stamp of safety approval to a system with universal-functionality?

The answer to this question is that the safety of the Purgles Platform is not just the safety of the software, but rather the safety of the entire process operating said platform. The design of FTIL makes the security model of the software mirror the governance model of its use.

Specifically, FTIL was designed with the following governance model in mind. Its initial premise is that while new algorithms can be implemented as FTIL scripts at any time, the existence of such scripts does not automatically grant them the power to perform any potentially-revealing information-exchange. Only inherently "safe" actions are initially allowed. Even so, such scripts are initially not uploaded to the system, and even those algorithms that are on the system require a per-use approval process before they can be executed.

The envisioned steps of this approval process are as follows.

- When a new functionality is programmed that implements a safe protocol through a series of steps, each of which may be individually "unsafe", the code for it, and any necessary additional documentation, is first screened, reviewed, and vetted on the AUSTRAC side. AUSTRAC reviewers consider the implementation's safety, applicability and stability. Well-written scripts should not only correctly implement a useful privacy-preserving protocol, but also provide safeguards against unintended (and, where possible, malicious) use, and incorporate actions that can trigger logging and, where necessary, downstream alerting, that can aid in surfacing any system misuse.

- Once approved on the AUSTRAC side, the implementation is provided for RE-side review. RE-side reviewers certify that the new functionality is, indeed, safe as claimed (under the assumption of judicious intel use of said functionality by AUSTRAC). Note that the Lego-like design of FTIL scripting means that certification of an implementation as "safe" implies that it can technically be used by AUSTRAC in any context, so approval needs to be considered in this wide scope.

- There is a Fintel Alliance-internal process in order to decide on the deployment of the system's capabilities for given, specific intel purposes.

- The code for the specific solution is then built, debugged, vetted and run by the system's end-users (intel plus FTIL specialist help). If such code is merely a Lego-build over existing, pre-approved scripts (which is intended to be the common scenario), no further approval is necessary from the REs. Otherwise, the full process of code approval needs to be instigated. (Note that some RE-approved code may include FTIL triggers that require RE-side operators to also provide per-execution approval.)

- All parties execute the script jointly, each able to verify that the other parties are following the script's correct steps.

- Deviations of one party from the script in a way that is invisible to all other parties cannot be used to gain undue information. This is ensured by the nature of the privacy-preserving protocols used.

- System logging allows post-hoc monitoring of the system and periodic review. Some operations may trigger alerts for immediate oversight actions.

- The users of each node are themselves vetted by their respective organisations.

In this document, we will focus on the security that is specifically afforded by FTIL, to support this structure.

FTIL's security model is as follows.

- The functionality of the system is in-principle generic, but in-practice any command that allows for substantial inter-node information exchange (especially between RE nodes) is privileged, and therefore not initially accessible. The system is therefore "safe" in its initial state, and FTIL's privilege mechanism ensures that no code can be executed unless it was properly certified and all data prerequisites have been properly verified.

- What the RE reviewers are provided for certification is FTIL scripts that encapsulate a specific functionality, using inputs that are tested by the script to be fit for the script's particular purpose and producing outputs that correctly attest regarding themselves what purposes they are fit for. (These attestations are made in FTIL by the use of collection "labels", which are strings that, in turn, direct reviewers to additional external documentation for a full discussion of the variables' intended properties.) The reviewers certify that the scripts are safe without the need for any additional context, that they implement the functionality they purport to implement, and that their outputs' declarations of fitness are all true. FTIL's certification mechanism supports this, allowing each RE to decide separately on certification.

- Not all FTIL scripts need to be thus certified, and some will not be eligible for a certificate at all because they do require additional context before they can be declared safe to use. FTIL provides a directive for AUSTRAC to flag such "unsafe" scripts to REs, so they do not accidentally become candidates for review.

- FTIL's execution model is one that ensures that all parties are synchronised regarding what is being executed. Each party can verify independently that all parties are following the intended protocol in lockstep.

- FTIL's specific commands (such as `broadcast` and `transmit` operations), as well as explicit user checks (e.g., via `assert` operations) and user notifications (e.g., via `display`) can be linked to logging, which, in turn, can trigger alerts where necessary.

11.4.3 Implementability: Can FTIL be realised on time and on budget?

FTIL, as described here, provides fairly-generic functional capabilities, which, at least at face value, appear harder to implement than a simple list of some small, fixed number of pre-canned algorithms. Is such a vision realisable, given the project's constraints?

This section was initially written before there was a team tasked with implementing FTIL. Today, the truly qualified reply to such questions should be provided by said team.

Nevertheless, here are some considerations that were made in defining FTIL whose intention was ease of implementability.

1. The language supports only a very limited number of types that are serialisable, especially if `array`s are not implemented.

2. We understand the expected load pattern of the system. The language was designed so that each of its commands may be separately heavy, but algorithms programmed in it do not rely on executing a great many such commands. For this reason, it is quite all right for us to implement this system so that there is a significant, noticeable overhead before the execution of each command, as long as the execution itself, outside this overhead, is efficient.

3. The syntax of FTIL isn't stipulated. As explained, what this document describes is the functionality to be supported, not the precise syntax. The syntax was designed to make the writing of examples easiest. In practice, we can determine the final syntax in whatever way we choose, after other architectural choices regarding the implementation are made, so as to simplify the syntax's parsing and the implementation.

11.4.4 Simplicity: Are we not over-complicating things?

Though not previously given as a design criterion for FTIL, it is still worth asking: was every complication introduced in the definition of FTIL necessary?

In short, the answer is "Yes".

Every element in the FTIL description was inserted into it in order to advance one of the language's design goals, such as verifiability and extendability.

The language description may appear a little "heavy", in two aspects.

- It restricts many things from occurring (e.g., by not allowing lists of lists), and

- It may sometimes appear to be two systems in one (e.g., in applying very different rules to elementary and non-elementary types).

The root reason underlying all such cases is that the language is meant to serve a dual-purpose, offering on the one hand a flexible, readable, user-friendly interface to its front-end user, and offering, on the other hand, highly optimised distributed mass processing in the back end.

To the extent that the definition of the language seems at any point to have a "split personality" or some unusual restrictions, this is meant to delineate these two worlds away from each other. Each is provided with tools most conducive for its task, communication between the two sides is kept simplest and using the most constrained possible set of tools, and the tasks are differentiated so that any tool necessary for one will not become a burden for the other.

At present, this is the best and most straightforward solution we could think of that actually addresses the complexities of the problem at hand.

Appendix A

Changelog

This is the 3rd revision (a.k.a. the "Purgles" edition) of the FinTracer Compendium. Past editions are

- First revision (released 27 October, 2020), and
- Second revision (released 1 July, 2021).

The following are the main elements that have changed from release to release.

A.1 From the second revision to the present release

A.1.1 Chapter 1: Introduction

1. Clarified the purpose of FTIL and its "language" status.
2. Added discussion to stress the importance of a programming interface that automatically sends FTIL instructions.

A.1.2 Chapter 2: The FinTracer executor

The FTIL execution model underwent significant changes. This edition not only presents the new version, but also goes into considerably more detail than the previous one (which purposefully left some difficult issues vague, in order to allow detailed solutions to be fleshed out later).

As part of the new presentation, the structure of the chapter has been completely redesigned. The following are some specific topics advanced.

1. Treatment of the Auxiliary DB is much more robust in this version. As part of this, the text expands significantly on the types of data that may be stored inside the Auxiliary DB, such as, for example, international transfer information and cash transaction information.
2. Previous versions mentioned caching, but largely kept it outside the scope of the document. The present version replaces this with a full treatment of the topic, now introduced as "save points", and demonstrates its interplay with the system's microservices architecture. (See Section 2.5.1.)
3. Discussion and requirements of the system's SQL environment was also moved into the scope of the document. (See Section 2.5.2.)
4. Handling of users and sessions has evolved significantly.
5. Added a discussion of how "`exec()`" and "`import`" are handled in terms of referencing external resources (via URIs).

6. Introduced a new command exit status. (See Section 2.11.1.)
7. FTIL's handling of privilege has evolved dramatically:
   (a) Privileged commands are no longer all of one type, but get individual treatment.
   (b) Certification is now individual to each RE.
   (c) FTIL now distinguishes signing from certification, and allows certified scripts to run non-certified scripts.
   (d) We separated to four clearly-defined execution modes, now fully documented.

Additionally, large portions of the chapter have been removed in order to focus only on the execution environment:

1. The exact semantics of execution-mode switching and any other FTIL-specific details were moved to Chapter 3.
2. Certificate management was moved to Chapter 6.

Additional, smaller improvements and fixes are scattered throughout.

A.1.3 Chapter 3: The FinTracer Intermediate Language

This version represents a major overhaul of FTIL, in both semantics and terminology. Much has evolved, largely because the original solutions, which side-stepped some difficult issues, ultimately didn't prove to be robust enough. The new version places FTIL on much sturdier footing.

Main changes include:

1. Clarified and evolved the semantics of type flavours, and particularly of "shared".
2. Evolution of "protection" and "immutability", now with clear semantics.
3. Minor changes in naming rules.
4. Changes to FTIL's typing system:
   (a) Types now divided into five clear categories.
   (b) Simplified handling of `arrays`, and reduced their overall priority.
   (c) Simplified the handling of cryptography: encryption and decryption are no longer privileged, ciphertexts don't need to remember where their keys are located, and many types specific to cryptographic algorithms have been retired in favour of usage of generic types.
   (d) Added support for the creation of cryptographic keys from seed values.
   (e) Extended the list of cryptographic algorithms we want to support. Specifically, added decryptable semi-homomorphic algorithms.
   (f) Introduced `catalogues`.
   (g) Introduced unified semantics for all collection types.
   (h) Re-introduced pointer-like behaviour in collections, and with this pointer arithmetic also the need for garbage collection.
   (i) Demoted `nodeset` to be an alias.
   (j) Simplified template support: `nodesets` are no longer valid template parameter types.
   (k) ElGamal ciphertexts are now structs, to enable visibility into their separate parts.
   (l) Extended the discussion on serialisable types.
   (m) Extended the discussion on how entities are initialised.
   (n) Significant evolution and clarifications regarding the semantics of `transmitters`.

(o) Introduction of pseudotypes, and particularly the new pseudotype "shared string".

5. Some constants are no longer needed, because their values can be calculated on the fly. This improves the resilience of scripts to being executed at less than maximal scope.

6. FTIL command handling has been fleshed out and clarified:

   (a) The roles of the Compute Manager and Segment Managers has been clarified and expanded upon. Shadow compute managers were introduced.

   (b) The five types of command interpretation are now separated and detailed, as well as the role of the Compute Manager and Segment Managers in each. This includes deep dives on how `catalogue`s and strings are handled. String handling is now much more robust than it was, as is general flow control.

7. This version lists exactly which information is available in the Variable Registry and which in the Variable Store, whether the information relates to entities, variables or identifiers, whether it is accessible to the Compute Manager, Segment Managers or both, and whether (and how) the information can be accessed from FTIL.

8. As part of the above, the document now explicates how introspection is handled (and the priority given to it is discussed).

9. Labelling has been completely re-invented. (See Section 3.10.) This includes:

   (a) Labelling is now only for collections.

   (b) The semantics of the `label` function changed significantly.

   (c) Fleshed out and evolved the concept of label confirmation.

   (d) Introduced a dedicated `confirm_label` command.

   (e) Introduced named lists to assist in label confirmation and in protection.

   (f) Introduced the "`modifying()`" clause.

   (g) Introduced weak labels.

   (h) Introduced suspended labels.

10. Added the modifier `NOCERT`.

11. Changes to many FTIL commands. Some examples:

    (a) Tweaks to the functionality of commands like "`dir()`", "`del`" and "`scope()`".

    (b) Additional clarification for commands like "`display()`", "`confirm()`" and "`transmit()`".

    (c) New commands introduced, like "`limit()`", "`inspect()`", "`initialised()`" and the "`distribute()`" function.

    (d) Removed commands, like "`cond()`".

    (e) Explicit listing of all arithmetic commands, string commands, membership operations, type conversion and flavour conversion options, element and entity deletion options, etc..

    (f) Introduced date arithmetic.

    (g) Additional clarifications regarding which functions are "simple" and which are "proper".

    (h) Introduced a ternary operator.

    (i) Explicitly prohibited short-circuit Boolean evaluation.

    (j) More robust handling of the Auxiliary DB: added read functions from the Auxiliary DB that return a set instead of a list, and clarified the interplay between HashRead commands and hashing that is done in-database or in FTIL.

12. Some changes to mass operations:
    (a) Tweaks to existing functions.
    (b) Changes to the handling of default values, type conversions, array iterators and more.
    (c) Introduced new types of reductions: "`min`" and "`max`".
13. Major changes to script semantics, which have completely shed now their macro-like origins:
    (a) Parameter passing is now quite fleshed out. Specific sections discuss its details, including the semantics of label confirmation, when applied to sent and returned parameters.
    (b) FTIL now has a three-tier referencing system, using the names "entity", "variable" and "identifier", as well as "elements" (previously called "sub-variables"). New sections have been added to explain this. The new "identifier" tier replaces the old macro-like parameter substitution and name-mangling rules. (Name mangling is no longer a part of FTIL.)
    (c) Scripts are now stack-based.
    (d) Script scoping rules have changed, and they now behave almost like pure functions.
    (e) As part of the above, global (non-system) variables have been abolished.
    (f) Introduced two new clean-up procedures, that are triggered depending on a script's exit status. Under normal conditions, scripts now delete their local identifiers when execution completes. If a script aborts, however, a special process is triggered, which saves the status of variables for later processing and/or debugging purposes. (See the new section, Section 3.13.6.)
    (g) Scripts protect the antecedents of their parameters. Methods also protect their calling `object`.
    (h) Differences were introduced between module and non-module scripts.
14. Major semantic changes to the handling of modules. These are now far more robust than before. As part of the above:
    (a) Major changes to the syntax and semantics of module loading.
    (b) Major evolution to the module proxy object (including a new name).
    (c) Introduction of language mechanisms to handle module dependencies.
    (d) Added (at lower priority) lazy module loading.
15. Everything in the chapter that is not user requirements has been moved to Chapter 11.
16. Additional minor fixes and improved wordings throughout.
17. Minor additional changes to priorities throughout.

A.1.4 Chapter 4: Example algorithms

In terms of the functionality implemented, this chapter covers roughly the same grounds as in the previous version, but it has been entirely rewritten from scratch, and the example code has been redesigned and reprogrammed from the ground up. Reasons for this rewrite include:

1. Bug fixes. These range from typing errors to designs that simply don't work and require language features that were not originally available to fix.
2. Changes in the FTIL language.
3. The chapter originally used code for algorithms that are no longer recommended for use in our algorithm suite as examples. These were all removed.

4. Code was refactored.
5. Some code was added specifically to demonstrate language features.
6. Extended discussion of new language features was added.
7. Names were changed for clarity and to adhere to a consistent naming scheme.
8. The commands' visual layout was improved through better use of spacing.
9. Some functionality was modified from the original code: e.g., whitelisting is now done via SQL querying, rather than manually. (This also caused a corresponding change in naming.)

Portions that underwent the most radical changes include:

1. The original key manager code was buggy, because it didn't take into account that the key manager data (the stockpile of zeroes) needs to keep updating even though the `object` holding it is labelled. The new code now makes good use of the new "`modifying`" feature. It has also been carefully designed so that one key manager can serve many processes without spawning a separate stockpile per use.
2. The original FinTracer tag was a bare-bones dictionary. It relied on certain language features (e.g., encryption and decryption being privileged, and FTIL's extensive use of specialised cryptographic types) to attain its design goals. Unfortunately, that made it inflexible, and it made the design of FTIL too purpose-built to support the single FinTracer algorithm, making it much more difficult to support with it alternatives and variations that we need. These language features have now all been removed, for a much simpler and more generic language definition, and the FinTracer tag is now an `object` type that knows much about itself (e.g., about its own key manager, and its own differential privacy properties), and has many methods that allow on it all manipulations we wish tags to support. These can now be certified—or not—individually.
3. For the same reason, account sets are now also their own, fleshed out `objects`.
4. FinTracer propagation has been simplified, because the code no longer includes support for a now-defunct path-finding algorithm.
5. FinTracer querying has been completely redesigned with new algorithms, and now supports a far more versatile, robust, and efficient paradigm for the support of differential privacy.
6. The example code now works as module code. It has been separated into six modules, namely for (a) key management, (b) tag definition, (c) tag propagation, (d) simple retrieval, (e) tag querying, and (f) end-to-end FinTracer pairwise queries.
7. The chapter also includes in-depth discussion on topics such as which scripts need certification and why.

A.1.5 Chapter 5: Cryptography

The chapter was an empty placeholder in previous versions. Now it includes a section breakdown and a discussion of our present differential privacy algorithms. Its other sections are still placeholders.

A.1.6 Chapter 6: Auxiliary interfaces

Minor touch-ups were done to Section 6.1 and Section 6.3 to fit their wording with the latest version of the FTIL language. Minor corrections were also made throughout.

These two sections are complete in this version, but the rest of the chapter is still a placeholder.

A.1.7 Chapter 8: Nonfunctional requirements

A final paragraph was appended to the end of Section 8.6, to reflect the impacts of the newly-added Section 2.5.2 on nonfunctional requirements.

The contents of this chapter remain a placeholder in this version.

A.1.8 Chapter 10: Cryptanalysis

1. The previous version's differential privacy analysis was done over a no-longer-used algorithm. This version updates to an analysis of the latest algorithm.
2. The analyses of the ElGamal Refresh and Sanitise operations are new to this version.

A.1.9 Chapter 11: FTIL practicalities

This chapter is new to this version. Section 11.1 and Section 11.4 significantly update material that was previously in Chapter 3. The rest of the chapter is brand new.

A.1.10 Other

1. A table of contents was added.
2. The Changelog has been updated and re-structured.
3. Appendix B is new in this version.
4. The bibliography was updated to include revision 2 of this document.

A.2 From the first revision to the second revision

1. This changelog added.
2. The bibliography has been updated.
3. Section 1.1 was rewritten to reflect the present aims of the document.
4. Mild phrasing changes (for correctness and clarity) and grammatical corrections throughout.
5. Mention of partner agencies and of transaction monitoring was added in Section 1.2.
6. The difference between the FinTracer algorithm suite and the FinTracer system was clarified in Section 1.2.
7. Terminology was updated throughout: "Tracer Node" is now "Peer Node", "Tracer" is now "Segment Manager", "Coordinator" is now "Compute Manager", "Globals" (or "Global Constants") are now "System Constants", and "Coordinator Variable Store" is now "Variable Registry".
8. **Chapter 6: Auxiliary interfaces** is new to this version, but its contents were previously circulated as a standalone addendum document.
9. Two additional chapters, **Chapter 7: Additional services** and **Chapter 8: Nonfunctional requirements** were added. Their contents are presently largely placeholders.
10. Chapter 9, Section 1.4 and Section 1.5 have been renamed, to better reflect their contents.
11. **Chapter 9: Infrastructure specifications** and **Chapter 10: Cryptanalysis** were separated into a new "Discussion" part of the document.

12. The examples in **Section 1.4: Interfacing with the FinTracer system** and **Section 1.5: Structure of the FinTracer system** regarding what is not in scope for this document and this system have been updated. This is for two reasons: first, FTIL has proved more powerful than originally anticipated, and second, we are experiencing some mild scope creep due to the possibility that we may need to support transaction monitoring, now or in the future.

13. Handling of a change of value to a variable defining the scope of a block while within the block has changed. It was originally implementation dependent. Now, scope is being evaluated only once, prior to the start of executing the scoped block.

14. Questions of when `cond()` conditionals are shared and when they are private have been expanded upon. Handling a change in the value of a `cond()` conditional was updated, similarly to the handling of a change in scope, described above.

15. Descriptions of scope handling have been reworded to avoid the ambiguous term "literal".

16. Everything to do with script signing and object instance labelling has evolved significantly.

17. Added peer-to-peer communication as part of checks in command/script execution, to make sure all participating segment managers are aligned.

18. Eliminated some repetitions between Chapter 2 and Chapter 3

19. An FTIL standard library has been introduced.

20. FTIL types have been simplified and clarified. This includes eliminating the `ByteArray` type, eliminating methods for the user to create new types, creating a separation between base types and "standard library" types, moving `NodeSet` (now "`nodeset`") to be a non-fixed-sized type, and simplification from four basic flavours of types to two.

21. Handling of constness and immutability has been clarified and simplified.

22. A consistent capitalisation scheme has been adopted. (Many names needed to be changed to accommodate this.)

23. Non-serialisable types have received an expanded and more detailed treatment, and now have their own section. In particular, methods are now their own types, and are not merely syntactic sugaring over scripts.

24. Modules have been introduced as their own non-serialisable type.

25. Introduction of modules changes (and in many case simplifies) script handling. In particular, modules have introduced namespaces for scripts that change script semantics.

26. The old type `class` has changed its name to `object`.

27. Operations on integers have now been detailed explicitly.

28. Memory handling in the presence of `objects` has been simplified, eliminating the need for any garbage collection at the expense of having no pointer arithmetic in the language.

29. The relationship between variables and the value objects they refer to has been clarified and simplified.

30. Some examples have been corrected/updated/clarified. See, for example, Figure 3.1 at Section 3.2.8, where the examples relating to the non-serialisable types `transmitter` and `object` have been updated.

31. Clarification of the difference between a script, a command and a function.

32. Introduced new terminology: "proper functions" and "simple functions".
33. The table `BSBtoRE` has been converted to a function called `BranchToBank`.
34. New section: **Section 11.1: Designing effective software in FTIL**.
35. Introduction of `exec` for automatic driver code.
36. Introduction of helper strings for scripts.
37. User sessions have been introduced, as part of better support for multi-tenancy.
38. Added commands for reading from the Auxiliary DB, including hashed reading.
39. Extended the functionality of "`dir()`" to include programmatic use.
40. Some mention of feature priorities was added to the document, as well as a discussion of how these priority statements should be interpreted.
41. Improvements in the handling of random number generation.
42. "`Output`" changed its name to "`export`".
43. FTIL's type conversion rules were fully defined, fleshed-out and documented.
44. The section on flavour conversion has been separated from the section on type conversion.
45. Section 3.11.20 was beefed up considerably.
46. Clarified that Auxiliary DB contains an image of the information in the Transaction Store.
47. Added description of explicit literal strings (including explicit support for multi-line strings).
48. Added description of implicit `pair` construction.
49. Added description of legal variable names.
50. Separated mass operations to its own section.
51. Clarification added regarding the mass operation reduction syntax.
52. Clarifications added on name mangling.
53. Hash types no longer carry as a template parameter the type of their pre-hashed input.
54. Section 11.4.3 was largely only needed for the first edition of this document. Though we haven't omitted it entirely, it was considerably shortened.
55. Within scripts, `for` loops can now loop on `sets` and `memblocks`.
56. Added clarifications regarding what explicit literals can be sent as script parameters.
57. Added clarifications regarding what can be used as script default values.

Appendix B

To do

The following are the items remaining to be done before this work-in-progress becomes a completed document (at least at MVP level).

1. Chapter 4 so far only covers algorithms up to and including the basic FinTracer algorithm. Many additional algorithms are critical components of the FinTracer suite. (This is true even for an MVP release of the Purgles Platform, i.e. without including advanced functionality like FinTracer DARK and oblivious querying.)

2. Chapter 5 is still mostly empty, and will need a discussion of our chosen cryptographic primitives, their parameters, their features, safety guarantees and properties (e.g., space and time requirements). Most of these primitives are for use in algorithms that are yet to be documented in Chapter 4. (Some material intended for this chapter is presently scattered elsewhere in the document.)

3. Section 6.2, Section 6.4, Section 6.5, Chapter 7 and Chapter 8 are all place-holders, and are presently assigned to Bruce Carney.

4. Chapter 9 is a place-holder, and is presently assigned to David Spencer.

5. Chapter 10 presently includes a discussion of two major cryptanalytic issues. We do not, at the moment, see an immediate need to expand on any other cryptanalytic topic, but it is likely that such topics will arise when more of Chapter 4 is written up.

Bibliography

[1] Brand, M. (2020), Efficient connection-finding with FinTracer. (Available as: `pair_findingpdf`)

[2] Brand, M. (2020), Improved differential privacy for FinTracer Boolean queries. (Available as: `boolean_queries.pdf`)

[3] Brand, M. (2020), Massively parallel FinTracer path finding. (Available as: `path_finding.pdf`)

[4] Brand, M. (2020), Oblivious setting of source accounts in FinTracer. (Available as: `oblivious_a.pdf`)

[5] Brand, M. (2021), Oblivious querying of FinTracer tag values. (Available as: `oblivious_b.pdf`)

[6] Brand, M. (2021), ElGamal key switching. (Available as: `keyswitch.pdf`)

[7] Brand, M. (2021), Extreme DARK: A more secure DARK variant. (Available as: `extreme.pdf`)

[8] Brand, M. (2021), Oblivious (and non-oblivious) discovery of sparse matches. (Available as: `oblivious_c.pdf`)

[9] Brand, M. (2021), FinTracer with side information. (Available as: `sideinfo.pdf`)