Blind Learning Environment (BLE)

Marco Benedetti, Giuseppe Galano, Andrea Gentili, Oliver Giudice, Michela Iezzi, Alessandro Maggi and Antonio Muci
Applied Research Team (ART)
IT Department
Bank of Italy

June 6, 2022

ART Project EP.AI.CS/2015/1

Revision History

Revision	Date	Author(s)	Description
0.1	18.12.2017	Marco Benedetti	created
0.2	22.12.2017	Marco Benedetti	reviewed
0.3	15.03.2018	Oliver Giudice	minor changes
0.4	18.07.2018	Michela Iezzi	added implementation details and data ingestion sections
0.5	04.10.2018	Michela Iezzi	updated after demo
0.6	23.01.2019	Michela Iezzi	added host machine architecture
0.8	01.03.2019	Andrea Gentili	updated dashboard section
0.7	18.06.2019	Michela Iezzi	updated after security assessment
0.9	20.03.2020	Andrea Gentili	updated to fix dashboard issues
1.0	06.06.2022	Michela Iezzi	added new workflow, modified data ingestion, added NER-based confidentiality check

Contents

1 Motivating Scenario **6**
- 1.1 Traditional SW development workflow . . . 6
- 1.2 Developing SW as a Data Scientist . . . 7
- 1.3 Not all Data Scientists are created equal . . . 7
- 1.4 BLE and its friends: NDA, TEE, HE, Anonymization . . . 9

2 BLE: Structure and functioning **11**
- 2.1 Target workflows . . . 11
- 2.2 The directional information boundary . . . 13
- 2.3 BLE Roles . . . 14
- 2.4 Sources of information . . . 15
- 2.5 How the bots operate . . . 16
  - 2.5.1 The DS-bot . . . 16
  - 2.5.2 The SA-bot . . . 16
  - 2.5.3 The DO-bot . . . 18
- 2.6 Mail-based channels and messages . . . 24
  - 2.6.1 DS $\rightarrow$ DS-bot messages . . . 24
  - 2.6.2 DS $\rightarrow$ DO-bot messages . . . 26
  - 2.6.3 DO-bot $\rightarrow$ DS message . . . 26
  - 2.6.4 DO-bot $\longleftrightarrow$ DO messages . . . 30

3 The BLE at runtime **33**
- 3.1 Creation and execution of jobs . . . 33
  - 3.1.1 Job identifiers . . . 33
  - 3.1.2 Job structure . . . 33
  - 3.1.3 The queue of jobs . . . 33
  - 3.1.4 Job input and output . . . 34
  - 3.1.5 Job execution . . . 34
- 3.2 Runtime checks . . . 35
  - 3.2.1 Mutex Access System - MAS . . . 35
  - 3.2.2 Integrity checks . . . 35
- 3.3 Event-driven architecture . . . 35

4 Implementation details **38**
- 4.1 Surrounding IT infrastructure . . . 38
- 4.2 The host machine . . . 38
- 4.3 Virtualization . . . 38
- 4.4 Network configuration and firewall openings . . . 38
- 4.5 Setup and configuration via Ansible . . . 39
- 4.6 Filesystem structure and permissions . . . 39
- 4.7 The Reserved Area . . . 39
- 4.8 Encrypted RAM Disk . . . 39
- 4.9 Booting the “BLE core" . . . 39
- 4.10 Instantiating bots . . . 40
- 4.11 Confining DS-bot . . . 40
- 4.12 Checking mails . . . 41
- 4.13 Exposing the state of ingestion through the dashboard . . . 42
- 4.14 Logging . . . 42
  - 4.14.1 Enterprise Security Information and Event Management (SIEM) . . . 42

4.14.2 Enterprise Technical Monitoring . . . 42
4.15 Running and manteinance . . . 42

5 Data Ingestion **43**
5.1 Structure of the external CD repository . . . 43
5.2 Triggering the Data Ingestion/Erasure process . . . 43
5.3 Light night-time ingestions . . . 43
5.4 ETL: Extraction, Transformation, Loading . . . 44
5.5 Extraction . . . 44
5.6 Transformation . . . 45
5.7 Loading . . . 45
5.8 Tagged ICD: the Named-Entity Recognition (NER) model . . . 45
5.9 Data Erasure . . . 45

6 The control dashboard **46**
6.1 Dashboard usage . . . 46
6.1.1 Menu . . . 47
6.1.2 Dashboard . . . 47
6.1.3 Completed jobs . . . 48
6.1.4 Administration panel . . . 48
6.2 Dashboard Information . . . 49
6.3 Encryption . . . 49
6.4 Web authentication . . . 49
6.5 Web authorization . . . 49
6.6 Interface between BLE and ble-ws . . . 50
6.7 BLE UI web service component, aka `ble-ws` . . . 51
6.7.1 Health endpoint . . . 52
6.8 BLE UI web client component, aka `ble-client` . . . 52
6.9 Dashboard Logging . . . 52

Appendix A. Summary of Security Features **53**
A.1. Security features meant to protect data as they enter the BLE and are stored in it . . . 53
A.2. Security features meant to ensure the integrity and confidentiality of the email-based workflow 53
A.3. Security features meant to ensure the integrity of the execution environment . . . 53
A.4. Security features meant to check the DS input script and sanitize the output of every DS job 54
A.5. Security features meant to ensure safe interaction with the Dashboard . . . 54

Appendix B. Example of Dashboard Log from startup **55**

Appendix C. List of Technical Features **56**

Appendix D. Acronyms & Abbreviations **58**

Blind Learning Environment (BLE)

Executive Summary

The Blind Learning Environment (BLE) is a computational infrastructure meant to allow a benevolent Data Scientist (DS) or External Researcher (ER) to safely apply Machine Learning, Artificial Intelligence, and Statistical Analysis algorithms on Confidential Data (CD), when the Data Owner (DO) *does not trust* DS, i.e.: CD can be accessed by DO and by the trusted System Administrator (SA), *but not by* DS/ER.

The BLE is useful when other solutions—from Non-Disclosure Agreements to Homomorphic Encryption—are not satisfactory or impractical, and is designed to prevent (unintentional) leakage of confidential data by the DSs (or leakage of personal data via queries done by ERs who access data for academic purposes). More details are given in Section 1.

BLE achieves this goal by splitting each of the aforementioned roles (DS, DO and SA) into two sub-roles (the *human*-side role and its *bot* counterpart) and by setting in place strict *information boundaries* between them. All bots live within the BLE, have access to CD, and are continuously subject to integrity checks (see Section 3.2). They act on behalf of the corresponding human agents, who, by contrast, need not have direct access to the internals of the BLE nor to CD; see Section 2 for more details. A simplified overview of the architecture is depicted in the following picture.

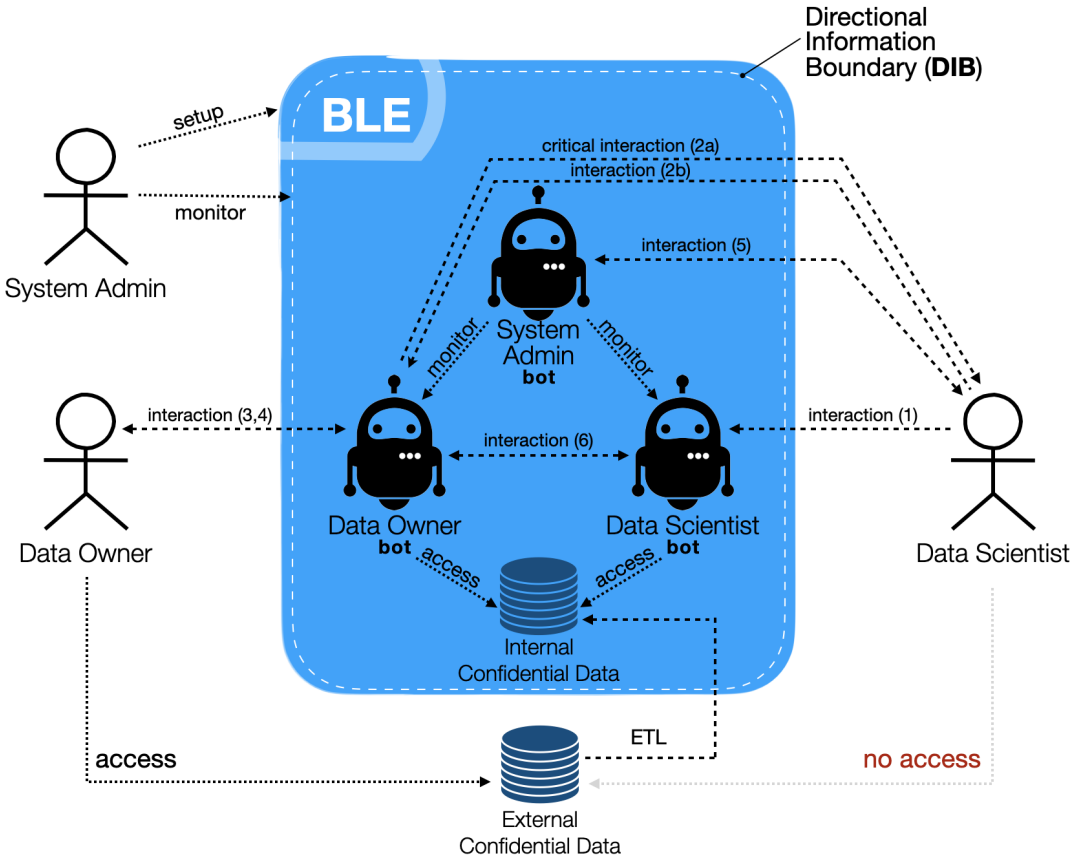

Figure 1: Architecture of BLE.

DS submits ML/AI/statistical jobs by email to DS-bot (see Section 2.6) and receives a response email with results from DO-bot, who first carefully inspects the content to decide whether it can be safely

forwarded to the DS; in most common cases, DO-bot can perform conclusive checks autonomously; more complex cases are decided by asking clearance to the human Data Owner (see Section 2.5.3.2). Details about the management of DS jobs and about the runtime model are given in Section 3.1 and Section 3.3, respectively. Every interaction of the DS with CD are tracked since DS submits ML/AI/statistical jobs by email and receives results by email.

The original, authoritative repository of Confidential Data (*External Confidential Data* database) lives somewhere outside the BLE. When necessary, such data is copied (fully or in part, over an encrypted, secure channel) into the *Internal Confidential Data* (ICD) - where it is stored in an volatile support (in-memory DB), using an *Encrypted RAM Disk* (see Section 4.8). ICD are managed on behalf of the DS by two processes (see Section 5): (i) the *Data Ingestion* process (a.k.a. ETL), that is in charge of extracting, transforming and loading (ETL) the portion of Confidential Data that is of interest for the ML job (see Section 5.4), and (ii) the *Data Erasure* process, which is in charge of ensuring that ICD doesn't persist inside the BLE any longer than necessary (see Section 5.9).

Finally, a secure communication channel between the BLE and the human counterparties is offered by the **control dashboard** (see Section 6); it shows the state of the computational environment (machine load, free RAM, etc.), the state of the Internal Confidential Data, of the bots, of submitted/completed DS jobs, etc.

The underlying hardware and network infrastructures are carefully set up and hardened, as described in Section 4.

Appendix A summarizes all the security features the BLE employs to protect confidential data.

Note: This document is an excerpt of the full (restricted) technical report describing the BLE Environment. This is the reason why the reader will find several "omissis" words, sentences, sections: The ones which contain Bank-of-Italy-specific implementation details. Overall, 17 pages of data and text have been redacted.

1 Motivating Scenario

It is often the case that software developers (**DEV**) are asked to develop a business application (**APP**) or information system (**IS**) that will need, once production-ready, to deal with **confidential data** (**CD**), i.e., data that can be freely accessed by some Data Owner (**DO**) and by no one else; in particular, *DEV must NOT gain access to CD at any time*.

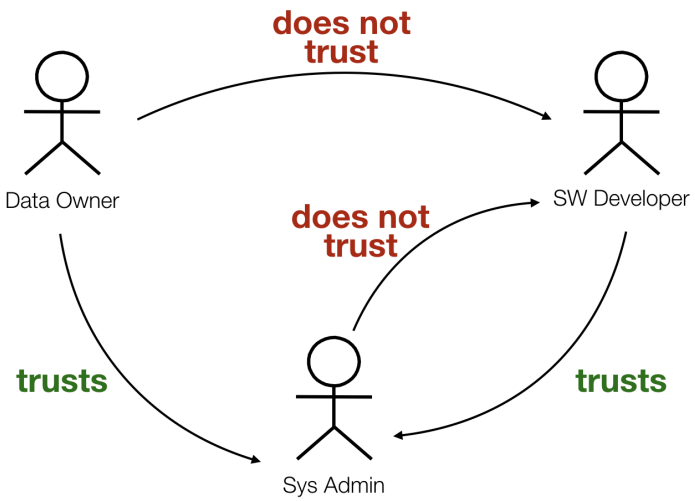

Figure 2: Motivating Scenario.

This situation is typically dealt with as follows.

1.1 Traditional SW development workflow

With help from a trusted third party—a system administrator (**SA**)—we have that:

1. SA sets up a **development environment** for DEV to operate in, and—in cooperation with DO—provides DEV with **test data** (TD) that reproduces any interesting feature of CD while being devoid of any confidentiality issue;
2. DEV develops some (version of) APP and ensures all functional and non-functional requirements are met, using TD to check APP;
3. Once (2) is completed, SA deploys the app to a **production environment**, where the real CD resides; only DO has access to production environment, together with SA itself for maintenance, upgrades, and system administration.

Whenever a new or updated/fixed version of APP is required by the DO, this cycle is repeated.

As a typical example, consider a case where CD is a (large) database containing sensitive personal and/or financial information about customers/citizens; APP is a custom application that allows DO to perform queries, correlation, searches, and visualisations over CD; TD is a database of dummy customers, filled with fabricated personal and financial information.

This traditional workflow works under a couple of assumptions:

1. DEV **needs not see CD** to develop APP, as APP is fully independent of the specific data being processed;
2. The $1 \rightarrow 2 \rightarrow 3$ cycle is "**slow**" (compared to the usage the DO makes of APP); e.g., APP is used tens of times a day by DO, while updates to APP by DEV are deployed every few months.

So, this particular workflow works even if no one ever entrusts DEV with manipulating real data.

1.2 Developing SW as a Data Scientist

It so happens that the employment of certain Machine Learning (ML) and Artificial Intelligence (AI) techniques by DEV breaks both assumptions; namely:

1. APP is **not** independent of the specific data being processed; absent any special arrangement, DEV **needs to see CD** to develop a functioning APP, because TD is not enough to guarantee that APP will work properly on CD;
2. The $1 \rightarrow 2 \rightarrow 3$ cycle is "**fast**"; i.e., the development cycle of APP may need to be repeated against CD several times a day (continuous development).

In this context, let us call Data Scientist (**DS**) a particular kind of DEV who, by virtue of using ML or AI techniques, needs to closely interact with real—and possibly confidential—data in order to develop APP.

When DS is not trusted by either SA or DO, the traditional development workflow from Section 1.1 doesn't work. A typical situations in which DS is not trusted by DO is when the DS role is external to the company. However, even if the DS works inside the company as a "trusted" employee, he may still be restricted from accessing CD for several reasons: Maybe he has not the same security clearance as the DO, or the experimental environment and SW infrastructure DS needs to use is not deemed safe enough for handling classified data, etc.

1.3 Not all Data Scientists are created equal

We want to give the DS the opportunity to test his techniques and algorithms on confidential data, while containing the risk that such confidential data are exposed to *anyone* else, other than DO and SA.

To better define the scope of our work, and the required features of the environment we are going to build, we present more specific notions of the behaviours we want to protect from, of the probabilities that such behaviours are put in practice, and of the potential resulting damage.

First of all, we restate the different "status" a DS may have in the eye of the DO:

- A **trusted data scientist** has been granted unrestricted access to CD but he is not allowed to show it to anyone else (neither in the original nor in derived form); in addition, he is not allowed to handle CD via any computational or communication infrastructure that is less safe than the ones the DO himself uses to handle CD;
- An **untrusted data scientist** has no access to CD, except perhaps in a few specific cases and subject to case-by-case approval by the DO. In addition to the constraints under which the trusted data scientist works, his untrusted cousin has to exercise his techniques "*blindly*".

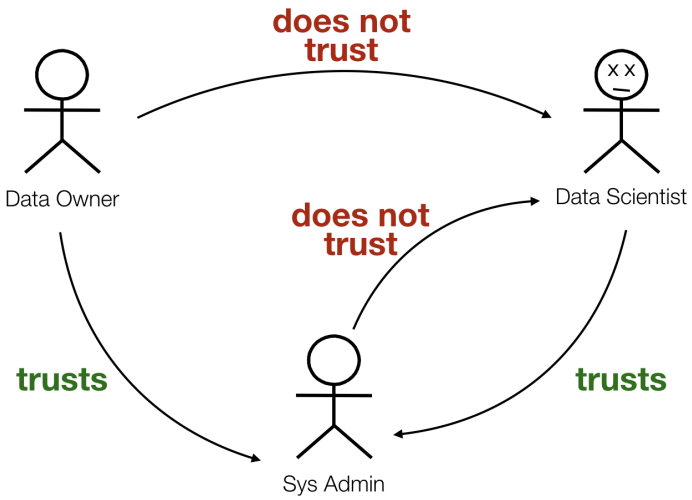

Figure 3: Business Case.

W.r.t. *intent*, we distinguish between two types of data scientists:

- A **malicious data scientist**. His real goal may be to leak CD, or to trick DO into allowing full access to CD in case access is restricted. He is motivated to attack any information system protecting CD, even by indirect means such as encoding or obfuscating confidential data into seemingly innocuous statistical reports or images;
- A **benevolent data scientist**. He is motivated to obtain the best possible performances from his algorithms in full cooperation with the DO. He is diligent and trustworthy, but of course software bugs or data mishandling on his part may still result in unacceptable information leaks. He himself needs and is willing to be protected from the consequences of causing accidental data leaks. What's more, the "DS" actor is in fact no single individual but a set of real people: It is sufficient that any one of these people leaks information for the damage to happen.

In this work, we focus exclusively on a **benevolent untrusted data scientist**. We are after an environment in which to let him run complex DS activities in such a way that:

1. The probability of unintentional leakage of confidential assets (to the benevolent DS himself or to anyone else) is **minimised**, whereas
2. the ability of the DS to operate is **unhindered** (as much as possible).

1.4 BLE and its friends: NDA, TEE, HE, Anonymization

The problem we consider here is not a new one. However, it is becoming more impactful by the day because of the growing number of AI techniques in use in business environments. Several solutions to this or similar problems—both technical and legal in nature—are already in use. We briefly discuss the main ones here, and consider how they relate to our case.

- **TEE** (Trusted Execution Environment); this is a (secure area of) a processor that provides strong confidentiality and integrity to data loaded inside it; it uses a combination of hardware and software (and cryptographic) features to provide strong guarantees within the boundary of an isolated environment that runs in parallel with the main operating system. Typical use cases are to provide an environment in which to execute secure financial transaction or biometric authentication (on a mobile phone). In our case, the *origin of trust* is in the computational infrastructure itself; so we don't require the BLE to be a TEE, though this is technically possible.

- **NDA** (Non-Disclosure Agreement); this would be a legally-binding contract that forces the DS not to divulge CD should he get to know (some of) its content; in our context, a similar agreement is already in effect as a consequence of DS having signed an employment contract with his employer (shared by DS, DO, SA). This is a purely legal tool, orthogonal to any technological security provision; by its nature, it can only be enforced after-the-fact and is an effective deterrent against certain behaviours of the benevolent DS; less so for malicious data scientists or against unintentional leakages, which can do more damage to the organisation than any sanction to the infringing DS can repay.

- **HE** (Homomorphic Encryption); this is a form of encryption that allows computation on encrypted data. In particular, computations are performed on cypher-texts and generate encrypted results which, when decrypted, match the results of the operations, as if they had been performed on the plaintext. The decryption keys are not accessible by and not necessary to those who perform the computations. The idea is to allow for (external, as-a-service) providers of computational power that can be trusted with operating on (internal, confidential) data for the simple reason they never get to see such data in clear format (think of cloud IAAS/PAAS/SAAS providers which operate on confidential assets). In our context, the BLE could be a HE platform operated by the SA, where encrypted data from the DO are processed by the DS. However, we would be solving a substantially different problem: Namely, *how to proceed when the SA is not trusted by the DO*. The assumption that DO trusts SA removes the need for an HE environment. Furthermore, even if we had a BLE with HE, the problem of who and when decrypts the HE results (ML outcomes which may contain confidential information) and let them known to the DS would still need a solution. Overall, HE and BLE are orthogonal mechanisms. Finally, it is not to be forgotten that HE introduces substantial overhead in the execution of any algorithms, while ML approaches require in some cases a massive amount of hardware-accelerated computational power (e.g., Deep Learning - DL), which would be difficult to express via pure software, let alone with the added indirection of an HE layer. So the two mechanisms may prove at present incompatible on a performance basis.

- **Anonymization** (or pseudonymization) techniques were initially considered as an alternative to the BLE, but ultimately discarded for the reasons we explain next. First, anonymization techniques, whereby personally identifiable information (PII) in the dataset are removed and irreversibly replaced by a unique identifier (such as the result of a one-way hash of the value) before handing the data to the DS are too strong for our use case, because the presence of the same individual in different contexts and records may be an important information to retain (too much information is lost). So, *pseudonymization* should be used instead: In this technique, PIIs are replaced by the same artificial identifier, or pseudonym, every time they appear. A table of correspondences between pseudonyms and actual PIIs is constructed in the process, which in certain cases is to be stored separately to make de-anonymization possible at a later time, whereas in other cases (like ours) is to be discarded immediately. There are 3 main problems with applying pseudonymization to our use case:

(i) A large pseudonymized dataset can be sometimes (partially) de-anonimized by either "inference attacks" (that detect small patterns unique to some individual to infer its identity) or "correlation attacks" (that join the data with other public datasets on the same subjects to identify them). Several cases of supposedly anonymous datasets that were deanonymized by these techniques are known in the literature. This would be quite problematic in our case, in case of a massive leakage of the pseudonymized dataset;

(ii) Our dataset contains much unstructured data (text) that may contain PII fields, possibly with spelling errors, plus sentences that make people indirectly recognizable through facts, dates, events. Unstructured data is very hard to pseudonymize but cannot be removed because it contains useful information;

(iii) Even if a pseudonymization approach were attempted, it would most probably be a trial-and-error process, wherein subsequent iterations of the procedure show better and better effectiveness on real data. Such procedures should be designed and executed by the DS, but only the DO could see the results of possibly imperfectly anonymized data at each iteration; so, the DS would need. . . . a BLE-like environment to develop an acceptable pseudonymization algorithm!

2 BLE: Structure and functioning

We present a computational setup meant to provide an answer—with caveats—to the following question (see the end of Section 1.3):

> *How do we allow a **benevolent untrusted** DS to safely apply **unrestricted ML & AI** techniques **without** granting him (direct) access to CD, under the assumption that a **SA trusted by both DS and DO** exists*?

There are two main ingredients to the solution we devise here:

- A dedicated computational infrastructure—called **Blind Learning Environment** (BLE)—is established by SA; it features *certified information boundaries* whereby any outgoing flow of information is monitored, filtered, and heavily restricted, in a DO-sanctioned way, employing a series of network-level, OS-level, and application-level checkpoints;
- The three relevant roles (DS, DO, SA) are **split into two sub-roles each**, assigned to human actors (DS, DO, SA) and to software robots (DS-bot, DO-bot, SA-bot) respectively. The bots live within the BLE and act on behalf of the corresponding human agents, who, by contrast, need not have direct access to the internals of the BLE nor to CD in normal conditions. Software actors are immutable and are subject to strict BLE-enforced rules, ultimately meant to bound their ability to talk to the outside.

A first simplified overview of the architecture is in Figure 4.

The key point here is the split of the DS into a human actor outside the BLE who designs, implements, and tests complex ML/AI algorithms but has no access whatsoever to confidential data (either *Internal* or *External* to the BLE), and a DS-bot inside the BLE that has full access to (a copy of) CD—the so-called Internal CD—and executes the ML/AI algorithms on them. The joint efforts of these two actors should ideally be sufficient to achieve any result a trusted human DS with full access to CD would have obtained. Furthermore, we stress that every interaction of the DS with ICD are tracked and logged since jobs and results are forwarded by email.

The External CD is copied inside the BLE and becomes "Internal CD" via a carefully tuned and secured process of *Extraction - Transformation - Loading* (ETL)—See Section 5.4 for details. At the end of ETL process, External CD is no longer accessed: The DS works on a (volatile, encrypted) copy instead, residing in RAM, which is erased when either (i) the DS declares her/his job is temporarily over, or (ii) a certain amount of time passes since the last interaction with the DS. See Section 5.9 for details.

2.1 Target workflows

Before detailing how each element of this construction works, let us characterize the main workflows we expect this architecture to be able to support.

The entire BLE construction is ultimately meant to enable and facilitate the ordinary life of an untrusted benevolent DS without requiring too much human intervention by DO; that is, DO-bot should be trusted with settling autonomously and correctly most common cases, without taxing the attention of DO nor slowing down the trial-and-error process (and progress) typical of any AI-intensive DS activity.

Three typical workflows, with their expected frequency and way of handling within the BLE, are the following.

1. **Param tuning**. Via some properly crafted processing script, DS asks repeatedly DS-bot to:
   - Perform the **learning phase** of some ML technique that needs access to full text confidential data on a certain training set;
   - Evaluate some **quality metric** (e.g., accuracy) of the resulting model on a specific test set;

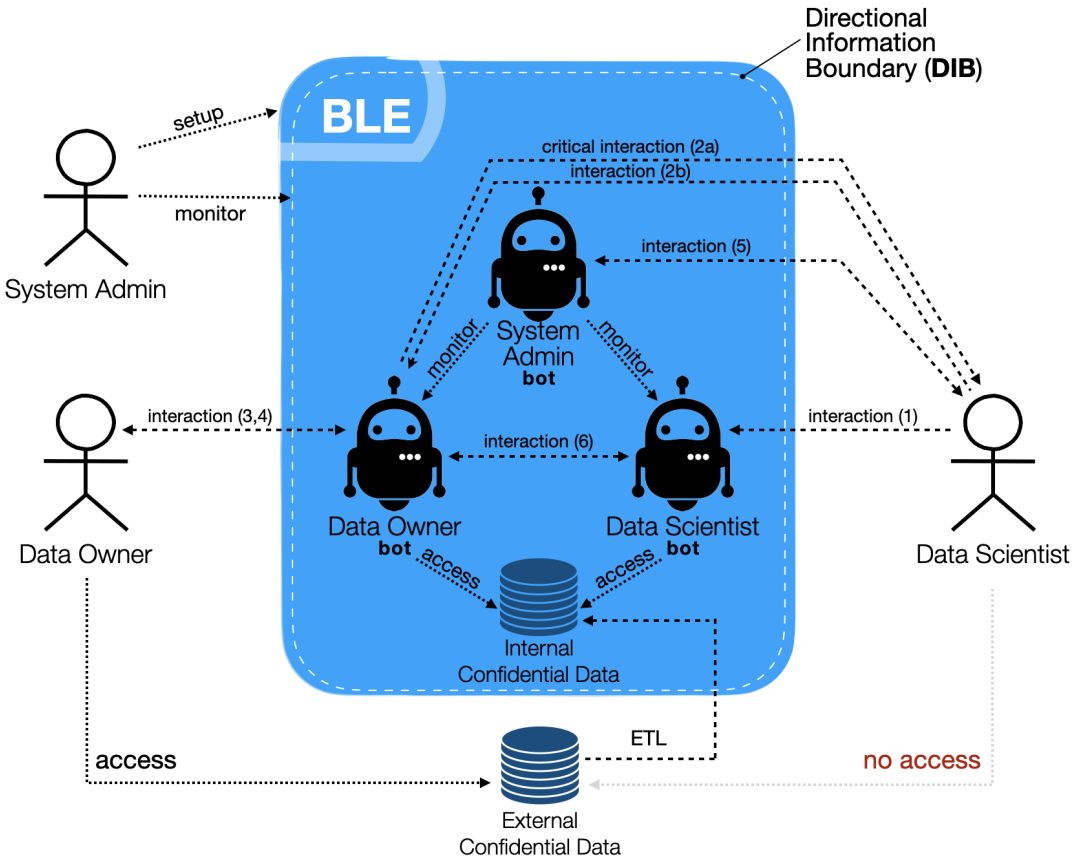

Figure 4: Simplified architecture overview.

- Send the **quality result** (e.g., numerical accuracy) back to DS. In this scenario, what is likely happening is that the DS is tuning parameters or structural features of the learning model at hand, and is repeatedly sending slightly patched processing scripts to DS-bot, one for each round.

How we expect the BLE to support this workflow: The DO-bot recognises this as a legit use case with no information leakage and allows the cycle to be repeated, even several times per hour, without blocking it nor asking the DO for his opinion; still, the DS-bot has full access to confidential data at each round.

2. **Library update**. DS realises that DS-bot needs to do things it cannot be instructed to do via a simple processing script; for example, a bug fix to a ML algorithm needs to be applied, or an entirely new library needs to be installed. The problem DS faces here can be solved—in principle—without gaining access to any confidential data.

   How we expect the BLE to support this workflow: The BLE lets DS in, after having rendered *unaccessible* any confidential asset or derived information; the DS is free to patch and upgrade the DS-bot as required, even several times a week, without having to ask for any blocking permission (from either the SA or the DO). No human intervention (other than by DS) is required.

3. **Complex debugging**. DS-bot is behaving unexpectedly, and there is some complex script debugging DS has to do, which requires gaining access to intermediate computation results, system logs, data

tables, etc. DS asks the DS-bot to send back all this information, which may or may not contain confidential data (by design or by accident). These complex situations may arise a few times a week.

How we expect the BLE to support this workflow: DO-bot—asked by DS-bot to forward DS a large amount of information—may be incapable of deciding autonomously whether any confidentiality breach would ensue should DS access it. So it asks DO for decision support. This injects a possibly slow human step in the workflow, but safeguards confidentiality and is acceptable for DS, assuming the situation is infrequent. An interaction path facilitates debugging. The DS can request the release of all the task output to the DO (therefore essentially requesting a "human" evaluation even in the face of an automatically rejected job from BLE). In addition, the logs of any errors produced during the execution of the scripts will be treated separately: the content of the "standard error" shell is stored in BLE but not automatically disclosed to the DS. This will be provided to the DS only after a "clearance" of the output by the DO, or as part of the standard authorization process (i.e., when the output of a job cannot be automatically blocked or released by BLE but human control is required), or at the request of the DS via the interaction path above.

2.2 The directional information boundary

The Directional Information Boundary (**DIB**) is one key element of the BLE: It is a **virtual perimeter** that can be traversed quite easily by *inbound information* (data, source code), but only under precise conditions by *outbound information*. The overarching idea is that if we can be sure no sensible information ever leaves this boundary to reach some untrusted actor, then we can be quite *liberal* about what information enters it (including CD) and which code is executed within its perimeter (anything the DS wants). Every interaction of the DS with CD are tracked since DS submits ML/AI/statistical jobs by email and receives results by email, as detailed in Section 2.5.

So, for example, the human DS can submit any instructions he wants to the software DS (DS-bot) for execution; a copy of CD can freely enter the BLE and be processed there; library and software updates coming from outside the BLE can be applied to DS-bot. Conversely, any outgoing flow of information is preliminary assessed and strictly regulated, with the goal of preventing confidentiality issues.

In the picture, dashed arrows represent interactions that imply a flow of information along the direction of the arrow. In particular, flows that originate within the BLE and are meant for some external actor or vice-versa are the following:

#	Path	Direction	Purpose
1	DS $\rightarrow$ DS-bot	`inbound`	submit scripts
2a	DO-bot $\rightarrow$ DS	`outbound`	output results and provide more information
2b	DS $\rightarrow$ DO-bot	`inbound`	request more information
3	DO-bot $\rightarrow$ DO	`outbound`	ask for clearance
4	DO $\rightarrow$ DO-bot	`inbound`	grant/deny clearance for output and more information
5	DS $\longleftrightarrow$ SA-bot	`bidirectional`	expose the control interface

Each of these flows will be studied in details; what matters here is that not all outbound channels are equally sensible; their potential for causing leaks depends on a combination of at least the following factors: How trusted the agent at the receiving end is; how secure the communication channel is; whether the information being transferred is structured or unstructured; how much information is traveling per unit of time; etc.

Let us define "**critical information path**" any *outgoing unbounded* flow of information that reaches an *untrusted* agent. The idea is that non-critical paths can be assessed at deployment time by SA and DO (e.g., during the Pre-Production Security Assessment - PPSA process) to rule out potential leaks (within reasonable certainty), so they need not be inspected at runtime, while **critical paths require runtime surveillance**. In particular, we assume any path that reaches trusted actors only or that reaches untrusted actors with a flow of structured and bounded information (with pre-determined structure and content) is *not critical*.

As we'll see, the BLE is built in such a way that there is only one critical information path: Path number 2a. It reaches a non-trusted actor (the human DS) with possibly confidential content. The entire DIB is then essentially about ensuring that **no confidential information travels across link 2a**.

The DIB is implemented and preserved by the cooperation of three actors, working at different times (some of them at runtime, some at deployment time):

- **SA**: The (human) *System Administrator* works at the **OS and network levels** at deployment time (e.g., he places the BLE in a Network Security Zone that forbids all IP connections but those approved by the DO and SA); several IPC primitives are forbidden; the OS user executing the DS-bot is constrained to write data in a few managed locations; etc., more on this in Section 4; the resulting arrangement is meant to ensure that there is only one critical path in need for "intelligent'" runtime monitoring (the aforementioned path 2);
- **DO-bot**: The *Data Owner bot* is entrusted with **inspecting communications** flowing over such link at runtime and filtering out any undue content; conceptually, this operation is similar to the deep packet inspection filtering that certain network firewalls operate: The software Data Owner reads and (to some extent) interprets the content of the messages to decide whether they can be *declassified*. Such filtering is done online, within the BLE, any time it is necessary. Section 2.5.3 gives more details about how this is implemented;
- **SA-bot**: The *System Administrator bot* continuously **checks the structural integrity** of the BLE at runtime, from *within* the environment. It also ensures that all security-related invariants and configurations remain in the state the System Administrator put them in at deployment time; more on this in Section 3.2.

2.3 BLE Roles

The five roles relevant to the BLE have the following permissions and features.

1. The *Data Scientist Bot* (**DS-bot**), embodied by a software agent, lives within the BLE; it has full access to the encrypted volatile copy of the CD (*Internal CD*) and is able to execute any number of ML/AI algorithms on behalf of DS, even those requiring extensive clear-text access to CD; it receives instructions on what to do from DS, in the form of a *processing script*; DS-bot *cannot* talk back to DS directly. Rather, any outcome or intermediate result of any computation executed by DS-bot (numbers, tables, text, graphs, etc.) is handed to DO-bot for further processing; the algorithms and workflows executed by DS-bot **cannot be trusted** by SA or DO, because over the lifetime of BLE they can be changed arbitrarily many times by DS, without any supervision nor any approval process.
2. The *(human) Data Scientist* (**DS**), embodied by a human actor, works both outside (often) and inside (infrequently) the BLE; in either case, he is granted **no access to (any copy of) CD** (see Section 2.4), nor to any data derived from it. The goal of DS depends on where he is operating:
    1. When he operates within the BLE, his objective is to update, upgrade, fix, extend, or otherwise change the DS-bot agent; he has access to nothing but DS-bot itself;
    2. When he operates outside the BLE, he designs tests and computations, represents them as processing scripts, and sends them to DS-bot for execution.

3. The *Data Owner Bot* (**DO-bot**), embodied by a software agent, is the only actor inside the BLE allowed to communicate with the outside world over a critical path; he receives computation outcomes from DS-bot, checks them, and decides—on behalf of DO yet fully autonomously—whether forwarding such outcomes to DS breaches data confidentiality. There are three cases here:

    1. The outcome of the computation poses **no confidentiality issue** (according to some formal notion of "confidentiality issue" agreed upon beforehand by DO and DS, see next); in this case, the information is forwarded to DS (and to DO for auditing purposes);
    2. The outcome of the computation poses **clear confidentiality issues** (again, according to some formalized notion of the term); in this case, the information is withheld from DS;
    3. The DO-bot is **unable to decide autonomously** whether forwarding the results at hand poses any confidentiality issue; in this case, he seeks help from DO to settle the case; in particular, a message with the possibly offending content is sent to DO over a secure channel, and a protocol implying human intervention is followed to decide whether DS can get to know the results of the computation.

The DO-bot, along with the computation outcomes, always forwards the respective processing scripts to its corresponding human counterpart; furthermore, encryption is always applied to its output messages. As opposed to DS-bot, the DO-bot agent is **trusted by both SA and DO**, because over the lifetime of BLE its structure and functioning don't deviate from its approved and security-checked initial installment; the DS has no bearing on it and cannot affect its operations (see point 5 below).

4. The *(human) Data Owner* (**DO**), embodied by a human actor, monitors the exchanges between DO-bot and DS; his oversight is non-blocking from the point of view of DS; that is, no human validation is required during "typical DS workflows" (see Section 2.1). Conversely, DO intermediates the communication between DO-bot and DS whenever DO-bot is unable to tell confidential information from forwardable one. DO is the authoritative source when it comes to decide whether a piece of DS-bot-generated information may be safely disclosed to DS. DO is trusted by definition.

5. The *System Administrator Bot* (**SA-bot**), embodied by a software agent, works inside the BLE on behalf of SA. It has four main responsibilities:

- [**Integrity**] to guarantee that the environment **stays** as **trusted** as initially deployed. In particular, it checks continuously that permissions of the various users and daemons and directories used to implement the BLE stay unaltered, and that the code being executed by himself and by DO-bot is unmodified (it is important to remark that the code that builds the environment is scrutinized and verified at deployment time by the human SA; so, the integrity checks performed by SA-bot are additional integrity controls on top of the human controls);
- [**Confidentiality**] to guarantee that before the human DS enters the BLE to work on DS-bot, every direct or derived CD is rendered effectively unaccessible (from memory as well as from local/network file systems);
- [**Workflow**] to manage the **data structures** shared among the three bots inside the BLE, such as the queue of jobs (see Section 3.1.3), and the working space for DS-bot, which may contain (derived) confidential data, enabling the workflow to proceed correctly;
- [**Control**] to expose proper **user interfaces** to both DO and DS through which they can safely monitor and control what is happening within the BLE.

2.4 Sources of information

Conceptually, four different areas for storing information and executable code are available from within the BLE:

1. The **Internal CD repository** contains a non-persistent copy of *External CD*. The *Internal CD* is built by a secured ETL process (part of DO-bot), which is *the only piece of machinery within the BLE enabled to read from the external DB*; inside the BLE, the *Internal CD* is securely stored on an Encrypted RAM Disk (ERD), as described in Section 4.8.

2. The **system configuration storage** contains any system software and application data, software, and configuration required for the BLE itself and the DO-bot and SA-bot agents to function. No one can write to this repository other than the trusted SA; confidential data are never placed here; this is typically a local disk on which the BLE operates.

3. The **ML configuration area** contains any system software and application data, software, and configuration required for the DS-bot agent to work. In particular, this repository contains all ML/AI libraries in use by the DS-bot, and the implementation of the DS-bot itself; **permanent permission is granted to DS to update/change such libraries and source code without any supervision**. This repository, by construction (see below), never contains confidential data; at implementation level, this is a local disk attached to the machine on which the BLE operates.

4. The **Workspace** contains any data produced by the DS-bot during its operations; this the only space to which DS-bot has unlimited (read and) **write** access, and may contain confidential information leaked from the CD repository; it is kept always unaccessible to DS. This is implemented by a directory placed in an Encrypted RAM Disk created into the machine on which the BLE operates.

The relevant access permissions granted to software and human DS agents in the context of these BLE repositories are as follows.

2.5 How the bots operate

2.5.1 The DS-bot

The DS-bot does the actual ML work submitted by DS. Each piece of work (job) being executed has full and exclusive access to the resources of the BLE while executing. The time a job takes to complete may be quite significant, up to hours or days. For example, the learning stage of present-day deep neural network approaches are particularly taxing for the underlying computational infrastructure, and may require days to converge.

The input scripts submitted by the DS are inspected for clearance by the DO-bot using a Named-Entity Recognition (NER) model; NER is executed right after the phase of data ingestion, and more details are given in Section 5.8. The DO-bot checks the raw strings in the script against the named entities in the ICD. Once the DO-bot greenlights the script submitted by DS, the task is executed.

The final output produced by each job is defined by the script itself and has **no fixed format**; that is: The BLE restricts where these results can be stored (namely, within the Workspace) and who has access to them (within the information boundary), but poses no a-priori limitation on their size, structure, or content. This means that—once and if the DO-bot/DO filter greenlights the transmission of the output(s)—an archive that packages the entirety of these outputs is created and sent back to DS.

2.5.2 The SA-bot

SA-bot maintains a **queue of jobs** to execute; each job is meant to perform a processing script sent by DS. The script may be as simple as a pure text file containing a sequence of commands understood by DS-bot, or as complex as an executable package containing embeddings, dictionaries, commands, datasets, tables of values, etc. In any case, the script may reference the entire content of the ICD, which is made accessible from within the script at a known "location" and in a predefined format (see Section 4).

Jobs run **sequentially**, in a FIFO order.

Given the potential duration and complexity of each script execution, it is not satisfactory for DS to ear no feedback at all from DS-bot—for hours or days—as the latter is executing.

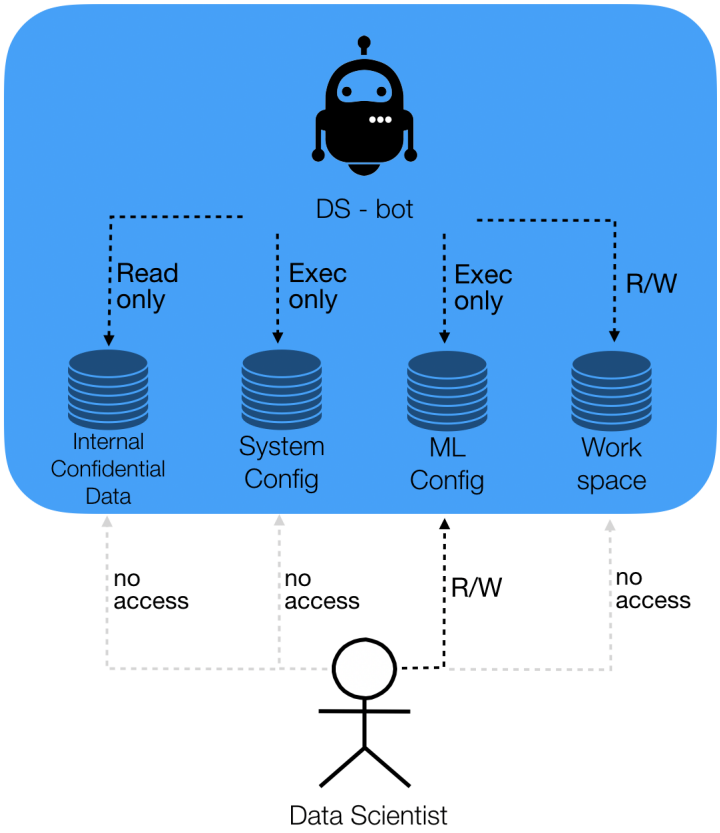

Figure 5: Access permissions.

To ease the issue, a dedicated control interface is exposed to DS by SA-bot (see Section 6); it provides ways and means to **monitor** and **control** what the DS-bot is doing, without exposing confidential information. This interface offers:

1. A **System statistics report**, i.e., a **Dashboard**; the current state of the BLE and of the underlying machine (average and instantaneous CPU and GPU occupation, free disk space on every relevant repository, number of files and bytes produced by the current and previous scripts, state of each process and daemon working in the BLE, mounted volumes and devices, number of online DS, etc.) is presented in a dashboard;
2. A **Queue management interface**; the current job may need to be killed prematurely, for example because DS realizes its output is going to be useless or unsatisfactory; or, jobs that have not yet been started may need to be removed or reordered; job queue may also need to be paused. All these operations can be done by exhibiting minimal metadata to DS (in addition to the script defining the jobs themselves); furthermore, a progress bar for running job is provided to the DS;
3. An **Administration interface**; this interface is mostly concerned with exposing controls to schedule the loading of the ECD into the ICD, to monitor the progress of such operations (which can takes hours), and to securely erase the ICD.

The existence of these three monitoring interfaces may theoretically pose a challenge to the confiden-

tiality of CD; i.e., they might in principle be exploited to leak (portions of) CD to DS. The BLE prevents such leaks as follows:

- All three interfaces are exposed by the trusted **SA-bot**; this means the code that exposes the interfaces cannot be changed by DS and has been audited and verified by both SA and DO at deployment time;
- Interfaces (1) and (2) are **non-critical** (cf. Section 2.1), because they are based on the transmission of a small amount of structured information that is known not to contain confidential data (e.g., average CPU occupation during the past hour); it would take a malicious and highly motivated DS and a lot of time to encode any sensible information into this channel; protecting against this malicious behavior is beyond the scope of the BLE;
- Interface (3) could be critical; for this reason, it will be forced to follow **the same workflow as a normal script output**: It will go through the automated checks of DO-bot and—if required—submitted to DO for clearance before reaching DS. Normally, these intermediate results will just contain some progress metrics and will pose no challenge to either DO-bot or DO.

2.5.3 The DO-bot

DO-bot is an event-triggered agent that—when presented by DS-bot with an input script or data package `MSG=<IN,OUT>` meant for DS—inspects it to decide whether it can be safely forwarded.

The first control is applied on the script submitted by DS: DO-bot checks the raws strings in the script submitted by DS against the named entities in the ICD; see Section 5.8 for more details about NER. If the DO-bot greenlights the execution of the script, a second level of control is applied on the results of the task.

`MSG` is a data package that includes the entire input `(IN)`—i.e., the input scripts and other files sent by the DS—and the full output `(OUT)`—i.e., the entirety of the output of the job. The full output `(OUT)` is splitted into `standard error`, that is stored internally and not delivered by default to the DS, and `standard output`, as in Figure 6. The `standard output` can be though of as a folder, containing text files, images, etc. DO-bot applies both deterministic checks and heuristics validations to tell how to proceed. If needed, DS can request for more information about `standard error` to the the DO-bot, which asks for clearance to the DO.

The non-numerical execution output `(OUT)` is checked by DO-bot against the named entities in the ICD, as in Figure 7. Then, a decision tree is applied. Before diving into the decision tree of DO-bot, let us introduce the key ideas supporting the choices made by DS-bot.

2.5.3.1 Information entropy and leaks

We eschew any sophisticated analysis of the content of `MSG` in favor of an information theoretic one, based on the hypothesis that the set of messages useful to the activities of a benevolent DS align more gracefully with non-leaking messages than with problematic (leaking) messages. In addition, we assume these two classes can be separated somehow effectively with a simple threshold on the entropy of the message.

In Figure 8, the distribution of any possible message generated by the DS-bot is shown, including any message leaking information (red) plus any possible non-leaking response message DS-bot may generate during its operation (blue).

DO is assumed capable of separating red messages from blue ones perfectly, no matter their position along the x axis; conversely, DO-bot can only reason approximately. Suppose DO-bot applies a binary separator at the optimal threshold, whereby messages to the left of the threshold are forwarded immediately to DS, whereas messages to the right of the threshold are sent to DO for verification first. We have:

- **False negatives**: Cases DO-bot forwards immediately to DS-bot, even though on close inspection DO would have deemed the message confidential; this set is supposed to be small if DS-bot is built by a benevolent DS;
- **False positives**: Cases where DO-bot suspends the process waiting for a final word by DO, that will happen to be positive (it is ok to forward); this is where the existence of the BLE workflows slow

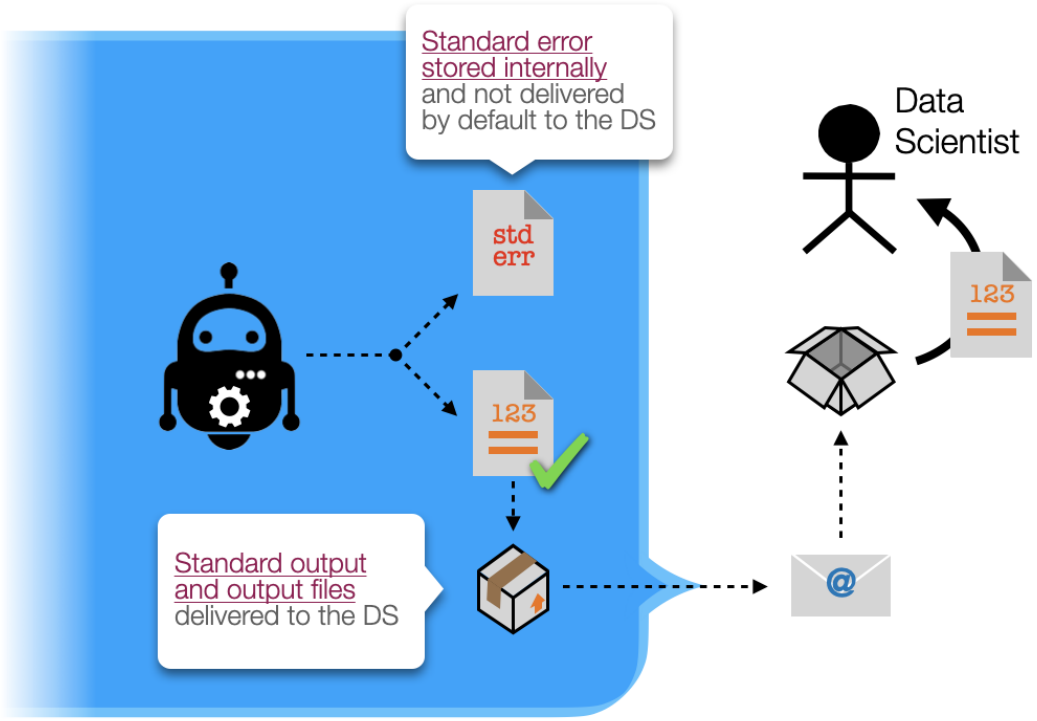

Figure 6: Output message.

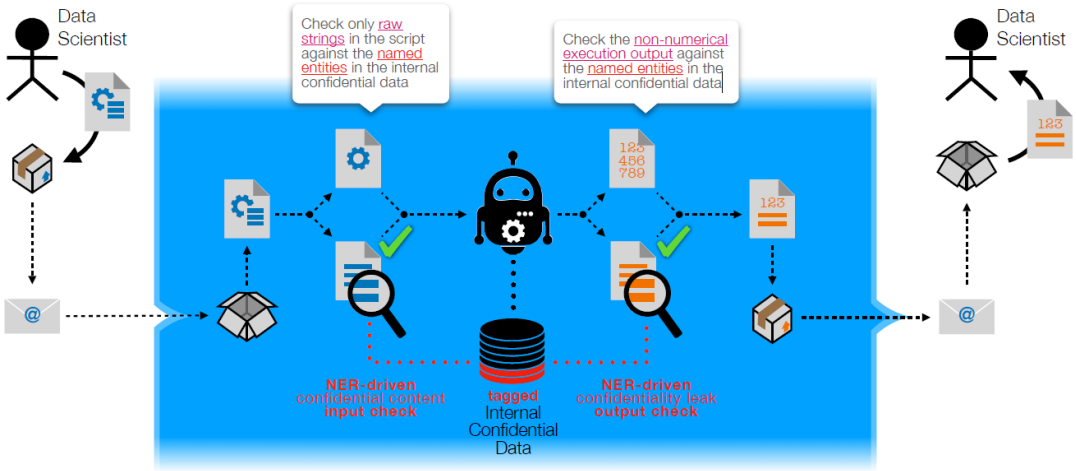

Figure 7: NER-based confidentiality check on output.

down the operations of DS compared to working outside the BLE; however, the quality of the final results by DS are unchanged;

- **Useful true positives** are messages potentially useful to the DS (the information they contain is helpful in improving the classifier or whatever ML task the DS is at) that won't pass the DO-bot/DO

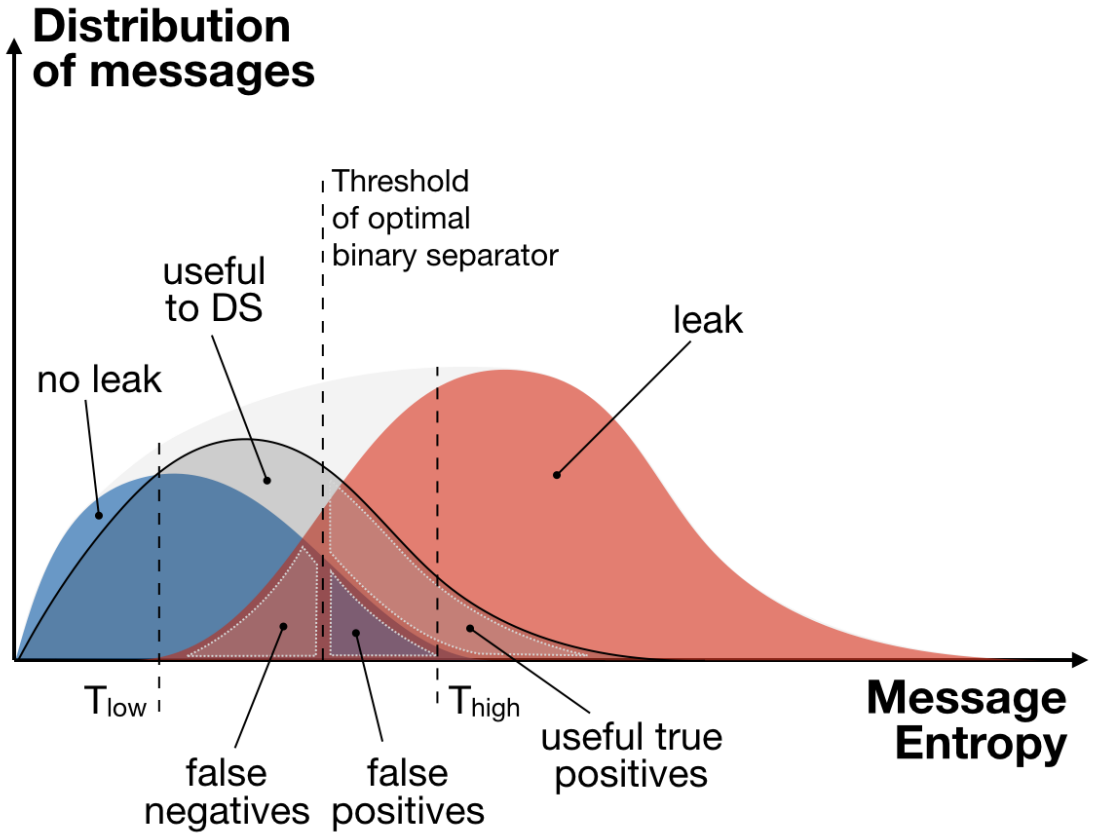

Figure 8: Distribution of messages vs Message Entropy.

filter (they do contain some confidential information); this set represents the cost DS necessarily pays for working with data *blindly*; the quality of the products delivered by DS may be reduced if this set is large.

To further improve the situation in terms of workflow speed and accuracy, i.e., to reduce the number of false negatives and false positives, we may split the single threshold of the optimal separator into two thresholds:

- T_{low}, below which the number of false negatives is assumed to be negligible and the message can be forwarded automatically;
- T_{high}, above which the number of false positives is assumed to be negligible and the message can be stopped automatically;
- in between T_{low} and T_{high}, help from DO must be summoned.

We remark that T_{low} and T_{high} are treated as configuration parameters that are decided by DO. As we will see, the DS-bot uses a proxy for the information entropy, that is, it measures the **size of a lossless encoding** of the message. In addition, any threshold-based reasoning will be applied not to individual messages, but averaged over a moving window that encompasses messages exchanged in a given period of time (hours).

2.5.3.2 Decision tree of DO-bot

At a high-level of abstraction, DO-bot may be seen as implementing the following pseudocode.

Algorithm 1 def check_and_forward_msg(MSG)

```
<IN,OUT> = MSG

 # Case 1: A sufficient condition for confidentiality violation is recognized
if definitely_ confidential(IN ∪ OUT) then
    tell DS: "A message from DS-bot has been blocked, sorry; some metadata attached"
    tell DO: "MSG has been blocked"

# Case 2: DO-bot is "quite sure" (heuristics) the message is not leaking any confidential information
else if heuristically_nonconfidential(OUT) then
    forward MSG to DS

# Case 3: DO-bot judges the message to be quite possibly confidential
else
    tell DS: "A message for you is in; waiting for clearance by DO"
    tell DO: "MSG is waiting for clearance"
    wait for an answer by DO
    once answer arrives do
    if answer is NOT_CONFIDENTIAL then
        forward MSG to DS
    else
        tell DS: "The message did not receive clearance, sorry"
```

There are three functions in this decision tree:

- definitely_confidential(...)
- heuristically_nonconfidential(...)
- heuristically_confidential(...)

upon which the exact behavior of DO-bot dependes. Let us look at them one by one.

definitely_confidential(...)

This function is meant to execute one or more fast and *deterministic* tests on the content of MSG, that are sufficient (though not necessary) to infer the presence of confidential information in it.

For example, if CD consists of records and records have fields known to contain personally identifiable information (PII), or sensitive personal information (SPI), then the presence in MSG of any such data (in clear format) is a sufficient condition to prevent any forwarding (check Section 5.5). An instance of this case is the presence in MSG of the IBAN or tax code or full name of an individual in CD. Of course, a simple clear-text check like this offers *no protection against a malicious DS* (this is not our target though; see Section 1.3); this preliminary automated check is only meant to save DO time by filtering out common cases of "technical" unintended leakage caused by a benevolent DS indirectly through DS-bot, such as the dump of entire CD records into a crash log produced by DS-bot and meant for DS.

The pseudocode of this function is as follows.

Algorithm 2 def definitely_confidential(ARG)

```
# A message is made up of several files in general
MSG_files = array(files_in(ARG))
# We search for sensitive contents inside any file, versus fields in the CD known to be confidential
for each file in MSG_files do
    for each word in file do
        for each confidential_field in Confidential_fields do
            if word in Internal_CD[confidential_field] then
                Return True
Return False
```

Note that this function is passed both the output of the job and the *input script*. This is done to avoid that the DS deduces confidential information from an apparently harmless job output by attempting to insert confidential data in the input script. For example, asking about the existence in the DB of a person with a given name or identifying code produces a simple one-bit answer which would pass all filters but ultimately deliver confidential information to the DS.

As usual, this naive clear-text check does not work (and is not meant to work) against a malicious DS, who can obfuscate the input code/name in arbitrarily sofisticated ways, but is an additional security check for a benevolent DS.

heuristically_non_confidential(...)

There are two scenarios in which this function deems, heuristically, that the content of MSG is *not* confidential:

1. (A proxy of the) entropy of the message content lays to the left of T_{low} (see Section 2.5.3.1); in other words, the message contains so little information that it is unlikely to be leaking confidential data *if produced by a benevolent DS*;
2. The message is made up of a single text file that is formatted according to one of a set of well defined templates.

The pseudocode of this function is as follows.

Algorithm 3 def definitely_confidential(ARG)

```
ZIPPED_MSG= zip(ARG)
size = size_in_bytes(ZIPPED_MSG)
if size < T_low then
    Return True
else if size < T_high then
    if path_to_job_output/result.txt is the only output then
        output_text = content_of_file(path_to_job_output/result.txt)
        for each template in SafeMLTemplates do
            if matches(output_text, template) then
                Return True
Return False
```

Each template recognizes the typical ouput of a ML technique, and is implemented via Regular Expressions (regex).

For example, the following is the typical output received by DS to evalute the quality of a classifier: Classical metrics that a DS may use for a blind evaluation of a job.

```
Confusion Matrix (normalized)

[[ 0.72932331  0.19548872  0.          0.05263158  0.0075188   0.
   0.0075188   0.          0.0075188   0.        ]
 [ 0.22727273  0.6969697   0.          0.02272727  0.00757576  0.01515152
   0.          0.          0.02272727  0.00757576]
 [ 0.          0.01290323  0.83225806  0.01935484  0.01935484  0.03225806
   0.00645161  0.          0.06451613  0.01290323]
 [ 0.06756757  0.03378378  0.          0.7027027   0.08108108  0.01351351
   0.02702703  0.          0.0472973   0.02702703]
 [ 0.          0.00847458  0.00847458  0.10169492  0.56779661  0.04237288
   0.04237288  0.00847458  0.12711864  0.09322034]
 [ 0.          0.02919708  0.          0.02189781  0.12408759  0.70072993
   0.          0.02919708  0.05839416  0.03649635]
 [ 0.          0.          0.          0.00775194  0.09302326  0.
   0.86821705  0.          0.02325581  0.00775194]
 [ 0.          0.00719424  0.          0.          0.02877698  0.07913669
   0.          0.83453237  0.04316547  0.00719424]
 [ 0.          0.          0.          0.02884615  0.03846154  0.04807692
   0.01923077  0.          0.83653846  0.02884615]
 [ 0.01449275  0.          0.00724638  0.01449275  0.0942029   0.01449275
   0.00724638  0.00724638  0.04347826  0.79710145]]

Report              precision    recall  f1-score   support

 class1                    1.00      1.00      1.00      2515
 class2                    0.99      0.98      0.98      2537
 class3                    0.97      0.99      0.98      2562
 class4                    1.00      1.00      1.00      2542

avg / total       0.99      0.99      0.99     10156

Accuracy 0.991433635289
Logloss: -0.493
  AUC: 0.824
```

Another example is the following: When training a Neural Network (NN), it is necessary for DS to analyse not only the final quality metrics, but also the information about the training process for each epoch or batch. The following example is a typical NN training result:

```
Epoch: 0000, loss=133.752563477, accuracy_on_test=0.0985,accuracy_on_batch=0.11
Epoch: 0001, loss=72.769439697, accuracy_on_test=0.1717,accuracy_on_batch=0.19
Epoch: 0002, loss=42.784042358, accuracy_on_test=0.2824,accuracy_on_batch=0.29
Epoch: 0003, loss=34.390235901, accuracy_on_test=0.3902,accuracy_on_batch=0.44
```

Again, this can be easily recognized via regex.

heuristically_confidential(...)

This function is meant to capture cases where there is reason to suspect `MSG` cannot be forwarded, because it matches no template and is too big to just contain reasonable DS output file.

Algorithm 4 def heuristically_confidential(ARG)

```
ZIPPED_MSG = zip(ARG)
size = size_n_bytes(ZIPPED_MSG)
if size> T_high then
    Return True
Return False
```

2.6 Mail-based channels and messages

DIB-crossing communications paths 1...4 (cf. Section 2.2) are implemented via **exchanges of standard email messages** handled by the Enterprise Trusted Mail Server and possibly encrypted using PKI certificates.

To this end, each human uses his company mailbox and each involved software actor has access to a dedicated BLE mailbox; the addresses associated with these mailboxes—for the purpose of this report—are as follows:

- (**a**) `<NAME>.<SURNAME>@ble.bankit.art` is associated with DS; it has rights to send mails to DS-bot and receive mails from DO-bot;
- (**b**) `ds.bot@ble.bankit.art` is associated with DS-bot; it can only receive mails from DS;
- (**c**) `do@ble.bankit.art` is associated with DO; it can send/receive mails to/from DO and send mails to DS;
- (**d**) `do.bot@ble.bankit.art` is associated with DO-bot; it can only send/receive emails to/from DO and send mails to DS and to DO.

Mailbox (a) is the normal company mailbox of DS. Mailbox (c) is a functional (multi-user) mailbox (FMB) to which the SA will associate the personal mailboxes of those allowed to act as DO. Mailboxes (b) and (d) are application mailboxes to which the BLE process/user associated with DS-bot and DO-bot is granted access (via PKI). Forwarding rules are implemented by DS-bot and DO-bot and are discussed in the next sections.

In the following sections we specify which conventions regulate how mail messages exchanged along channels (1)...(4) are to be interpreted by software and human agents. We emphasize that channels 2,3 and 4 involve encrypted communications.

2.6.1 DS → DS-bot messages

Any message sent by `name.surname@ble.bankit.art` to `ds.bot@ble.bankit.art`—independently of the message subject and content—is interpreted as a request to enqueue (and execute) a new job.

The **subject line** of the message is a free-form string that can be used by DS to briefly describe what the job is meant to do; it should be short and descriptive, and perferrably unique. It will be used in all subsequent communications by BLE bots to refer to and identify this particular job and to optionally manage its life cycle via the control dashboard.

The processing script this new job is meant to execute can be specified in different ways:

1. A mail with **a pure-text body and no attachments** is interpreted as a no-dependency script to execute; the commands are written by DS in the body of the mail, which is supposed to be correctly formatted according to the syntax of the scripting language in use and to mention/import only libraries DS-bot was previously configured to use;
2. A mail with **pure-text content and some attachment(s)** is interpreted as a script with dependencies (e.g., it needs to access data that are not already inside the BLE to function properly); all such dependencies (datasets in `.csv`, pre-trained embeddings, etc.) are included as a list of unencrypted (but possibly individually compressed as `.gz` or `.zip`) attachments to the mail; when the script is

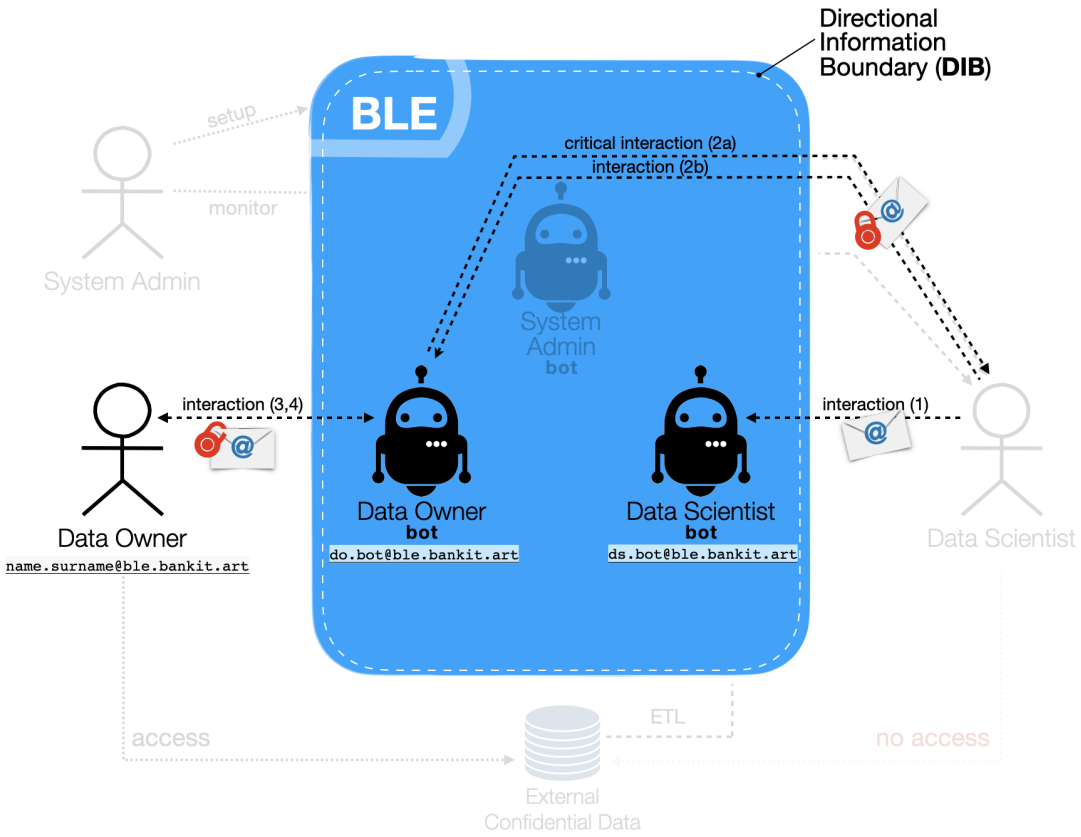

Figure 9: Mail-based interactions.

executed, all such attachments are made available to DS-bot in the local execution directory of the file system (compressed attachments will have been decompressed);

3. A mail with **one line of content and one compressed archive as its unique attachment** is interpreted as a script with dependencies to execute; the script together with all its dependencies is sent as a (compressed) archive (`.tar` or `.tar.gz` or `.zip`), which is expanded in a temporary working location; the entry point of the script in the archive is given as a single line of text in the body of the mail.

The dimension of attachments could vary between a few hundreds Kb to the maximum allowed by the enterprise Mail Server; each DS could send from 10 up to 100 attachments per day. A simple example of a mail structured to submit a job according to case (2) is the following.

```
=====================================================================
FROM: name.surname@ble.bankit.art
TO: ds.bot@ble.bankit.art
SUBJECT: TensorFlow, 100 epochs, third attempt
ATTACHMENTS: wiki.en.vec (14.4MB)
ENCRYPTED: NO
=====================================================================

#!/usr/bin/python3
```

```
# Setup of input & output paths (roots are opaque, provided by BLE)
from BLE import PATHS

# dataset previously stored here in ML-friendly format:
dataset_filepath   = join(PATHS["cache"], "confidential.csv")
# see the attachment to the present message:
embedding_filepath = join(PATHS["input"], "wiki.en.vec")
# the content of this will be sent back to DS (hopefully):
result_filepath    = join(PATHS["output"], "result.txt")

# Reading confidential data
data_frame = pd.read_csv(dataset_filename,sep='\t', encoding='utf-8')

# Reading non-confidential job-specific input
embedding = KeyedVectors.load_word2vec_format(embedding_filepath)

# Here, some complex ML algorithm is applied to data_frame.
# The embedding at aux_input_filepath is used to extract features from text.
# Both the training phase and the test phase are executed.
# After a couple of hours, we obtain a value for the following variables:
(accuracy, confusion_matrix) = do_the_real_work(data_frame,embedding)

# Save the results of the job
with open(result_filepath, "w", newline="\r\n") as result_file:
        result_file.write("Overall Accuracy: "+ str(accuracy))
        result_file.write("Confusion matrix: ")
    for line in confusion_matrix:
            for value in line:
                result_file.write("{:10.4f}".format(value))
            result_file.write("\n")
```

Note that DS-bot will never respond to this message directly; the answer will come from DO-bot.

2.6.2 DS → DO-bot messages

The DS can request more information to the DO-bot about a submitted job. A message sent by the DS to `do.bot@ble.bankit.art` is interpreted as a request for more information about a given task, specifying its identifier. The DS can request the access to `standard error` both if the task passed successfully or if it has been rejected by DO or DO-bot. Both the cases are represented in Figures 10 and 11.

2.6.3 DO-bot → DS message

Any message sent by DO-bot mailbox to DS mailbox—independently of the message subject and content—is interpreted as a notification that a job in execution by DS-bot is terminated and has produced certain outputs, which have been greenlighted by the DO-bot/DO filter and can now be inspected by DS.

The mail will make clear—using the subject and the body—whether the output of the job:

1. is being **forwarded** to DS (in the body/attachments of the email):
   - (**a**) by autonomous decision of DO-bot;
   - (**b**) by DO-bot after consultation with DO;
2. will **not be forwarded** to DS (the rest of the email is empty, no attachments), because:

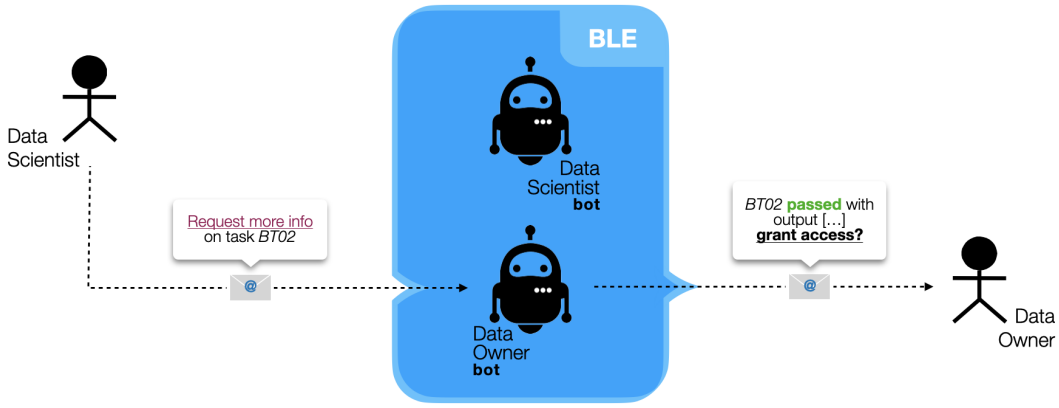

Figure 10: Request more information interaction: access to standard error if task passed.

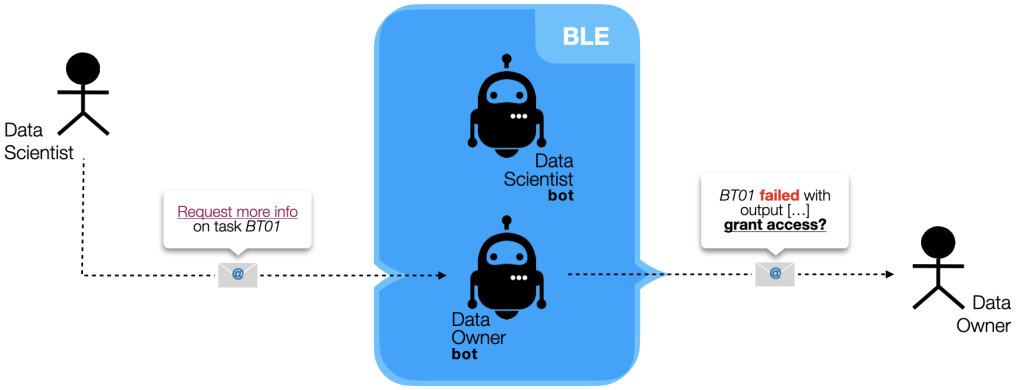

Figure 11: Request more information interaction: access to standard error if task failed.

- (**a**) DS-bot automatically detected confidentiality violations;
- (**a**) DO, called into question by DO-bot, deliberated that the mail content was confidential.

The **subject line** of the email is:

- `Output of <ID> is ready` in cases (1a) and (1b);
- `Problems with <ID>`, in cases (2a) and (2b).

In both cases, `ID` is replaced by the subject line DS wrote in the originating DS→DS-bot request email. The **receivers** of the email will always be:

- `TO` : `name.surname@ble.bankit.art`, i.e., the DS requesting the job;
- `CC` : `do@ble.bankit.art`, i.e., the functional mailbox of DO.

That is: The DO mailbox is always kept as secondary recipient of every communication that occurs between DO-bot and DS.

The **body of the message** contains an initial section with information about the job just completed, who submitted it, how long it took to complete, etc. Crucially, this header section always specifies what was the assessment of DO-bot and DO on the result of the job.

Then (only if the output was greenlighted by DO-bot/DO, so in cases 1a/1b):

- If the execution of the job has produced as output just **one single, pure-text file**, the content of such file is appended to the mail body;
- If the output produced by the job is a **set of files** stored at the root level of the directory reserved for the output of the job within the BLE, then each such file is an attachment to the email; large files may be attached in compressed `.gz` or `.zip` format;
- If the output produced by the job is **anything more complex** (i.e., a directory tree), then such tree is attached to the mail as a compressed `.tar.gz` or `.zip` archive.

It is worth mentioning that the input script and the attachments sent by the DS to the DS-bot is always appended to the mail body too; furthermore, the entire message (input and attachments by DS and output) will always be encrypted. An example of a mail structured to report about the outcome of the job submitted by the sample message at the end of the previous section is the following.

```
========================================================================
FROM: do.bot@ble.bankit.art
TO: name.surname@ble.bankit.art
CC: do@ble.bankit.art
SUBJECT: Output of 'TensorFlow, 100 epochs, third attempt' is ready
ENCRYPTED: NO
========================================================================

This is an automated email from the DO-bot bot.
The results of a job submitted by DS are in: See below.
Please don't reply to this email.

Sincerely,
your trustworthy DO-bot.

------------
- OVERVIEW -
------------

  Job title: TensorFlow, 100 epochs, third attempt
  Submitted by: name.surname@ble.bankit.art
  Submitted on: 2019-05-10 09:02:01.40 +01:00
  Completed at: 2019-05-10 11:10:30.21 +01:00
  Duration: 2h08m29s
  BOT assessment: PASS
  DO assessment: NOT required

----------
- OUTPUT -
----------

Overall accuracy: 0.965545
Confusion matrix:
    0.9133    0.0000    0.0000    0.0766
```

```
    0.0000    1.0000    0.0000    0.0000
    0.0000    0.0000    1.0000    0.0000
    0.0867    0.0000    0.0000    0.9234
```

Had the script output leaked confidential information (caught by DO after a request by DO-bot), this mail would have looked as follows.

```
========================================================================
FROM: do.bot@ble.bankit.art
TO: name.surname@ble.bankit.art
CC: do@ble.bankit.art
SUBJECT: Problems with 'TensorFlow, 100 epochs, third attempt'
ENCRYPTED: NO
========================================================================

This is an automated email from the DO-bot bot.
The results of a job you submitted are in but cannot be forwarded to you.
Please do not reply to this email.

Sincerely,
your trustworthy DO-bot.

------------
- OVERVIEW -
------------

  Job title: TensorFlow, 100 epochs, third attempt
  Submitted by: name.surname@ble.bankit.art
  Submitted on: 2019-05-10 09:02:01.40 +01:00
  Completed at: 2019-05-10 11:10:30.21 +01:00
  Duration: 2h08m29s
  BOT assessment: UNSURE, I ask DO
  DO assessment: CONFIDENTIAL

--------------------
- COMMENT from DO -
--------------------

Hi guys; you screwed up for real this time!
Half my confidential dataset was dumped into
what looks like a crash log or something...
Perhaps this is just a bug in your code?

Cheers,
Ric

----------
- OUTPUT -
----------

You don't have permission to access the output.
```

The DO-bot can forward more information, i.e. `standard error`, to DS, if required, after having

received the DO clearance, as in Figure 12.

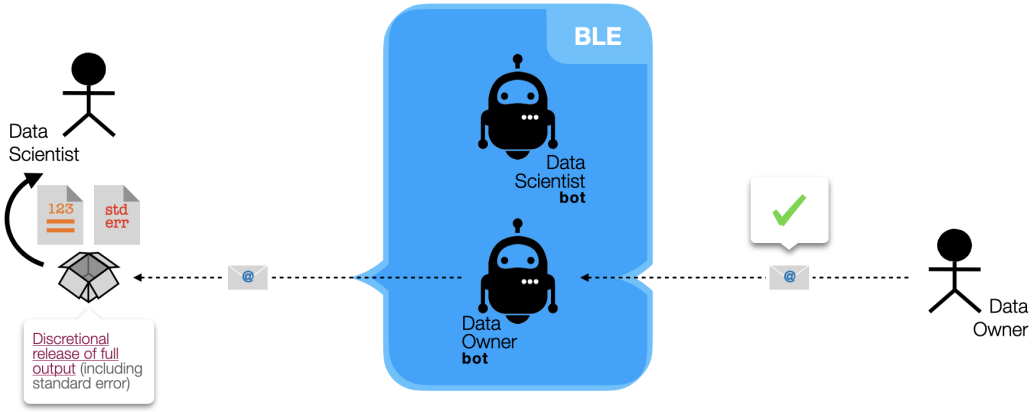

Figure 12: Request more information interaction: DO Clearance.

2.6.4 DO-bot ⟷ DO messages

If DO-bot requires the intervention of DO to screen some output for confidential content, then `do.bot-@ble.bankit.art` sends an email to `do@ble.bankit.art`; this message is signed and encrypted. The structure of the email is exactly as the messages described in Section 2.6.3, with the exception of the **subject line**, which changes to: `Please check the output of <ID>`. Again, `<ID>` is replaced by the subject line the DS wrote when he submitted the script.

Once DO has inspected the mail content (and possibly all of its attachments) and has assessed whether confidentiality would be breached by forwarding it to DS-bot, then he hits the button corresponding to his decision:

- `OK` if the content of the mail poses no confidentiality challenge;
- `NO` if the content of the mail looks suspicious or confidential.

The subject of the DO→DO-bot email will be automatically composed by the mail client and must not be modified by DO; it will be like this: `accept:<ID>` in case of `OK`; `reject:<ID>` otherwise, where `<ID>` in this case is instrumental and a unique id for the task assigned by BLE.

DO can also write, in the first non-empty line of the body of the mail, a brief free-text comment which will be forwarded to DS verbatim, in both cases (i.e., whether the answer is `OK` or `NO`). These comments will be typically used—in the `NO` case—to explain DS why the results cold not be made available to him.

For example, suppose DO receives this email.

```
========================================================================
FROM: do.bot@ble.bankit.art
TO: do@ble.bankit.art
SUBJECT: Please check the output of 'TensorFlow, 100 epochs, third attempt'
ENCRYPTED: YES
========================================================================

This is an automated email from the DO-bot bot.
The results of a job DS (name.surname@ble.bankit.art) submitted are ready
but they look suspicious to me.
```

```
Please select an action:
- "OK" if you think DS can see the results below;
- "NO" if you think the results below should remain confidential.

You can also include a message (optional) as feedback for DS.

Sincerely,
your trustworthy DO-bot.

------------
- OVERVIEW -
------------

  Job title: TensorFlow, 100 epochs, third attempt
  Submitted by: name.surname@ble.bankit.art
  Submitted on: 2019-05-10 09:02:01.40 +01:00
  Completed at: 2019-05-10 10:03:01.50 +01:00
  Duration: 1h01m01s
  BOT assessment: UNSURE, I ask DO
  DO assessment: PENDING

----------
- OUTPUT -
----------

Traceback (most recent call last):
File "ML_feature_ext.py", line 127, in <ML_module>
KeyError: 'code'

Key 'code' not found in dictionary:
{"ID" = "LF01.2c",
"list_of_CC_frauders" = [
{"name" : "Proserpio Mazzi", "born" : "January 20, 1963", "CI": "RA88673845", "CC":
 "4716 9454 7898 1569", "type" : "visa", "n_frauds": 2},
{"name" : "Severino Nucci", "born" : "May 18, 1942", "CI": "DW17587500", "CC":
"5484 0333 1972 9542", "type" : "visa", "n_frauds": 1},
{"name" : "Giovanna Panicucci", "born" : "November 15, 1970", "CI": "IQ87739818",
"CC": "5262 1089 0182 4618", "type" : "mastercard", "n_frauds": 15},
{"name" : "Ernesto De Luca", "born" : "January 26, 1938", "CI": "KD77125421", "CC":
"5120 8226 1865 8463", "type" : "mastercard", "n_frauds": 1},
{"name" : "Nazario Giuseppina", "born" : "September 14, 1975", "CI": "IT96884716",
"CC": "4916 2426 4516 9601", "n_frauds": 1},

[... thousands more records here]

]}
```

DO recognizes DO-bot was right to be suspicious, and also sees the leak is accidental. He hits "NO" button and composes an email such as the following.

```
========================================================================
FROM: do@ble.bankit.art
```

```
TO: do.bot@ble.bankit.art
SUBJECT: reject:'TensorFlow, 100 epochs, third attempt'
ENCRYPTED: YES
=====================================================================
Hi guys; you screwed up for real this time!
Half my confidential dataset was dumped into
what looks like a crash log or something...
Perhaps this is just a bug in your code?

Cheers,
Ric
```

The same inspection and forward process applies if the DS asks for more information, i.e. `standard error`, to DO-bot.

3 The BLE at runtime

3.1 Creation and execution of jobs

3.1.1 Job identifiers

Inside the BLE, any "job", i.e., script submitted to DS-bot for execution, is represented—whichever its execution state—by a directory whose name follows this convention:

<JOB_ID> := RECEPTION_TIMESTAMP-DS_SUBMITTER_EMAIL

For example, a directory named:

20190510132311-name.surname@ble.bankit.it

represents and uniquely identifies the job submitted for execution by the DS with email address `name.surname@ble.bankit.it` to the DS-bot at 13:23:11 of May 10, 2019.

3.1.2 Job structure

Inside any job directory, at the first level, there is always an "entry point" named `script.py` from which the computation starts.

This entry point is the only required component of any job. Optionally, there may be arbitrarily many files and (possibly nested) subdirectories with assets and further scripts required to execute the job.

All this executable material is submitted by the DS to the BLE in one of several possible formats—see Section 2.6.1.

The content of a job directory and its name are *immutable* (with the exception of `index.yml`, see below): They won't ever change for the entire lifetime of the job. The output of the job and the (possibly confidential) inputs to it are placed elsewhere—see Section 4.6.

So, by construction, no job directory will ever contain any confidential information. Also, an `index.yml` file containing metadata about the job is automatically created and updated by DS-bot througout all job execution phases. A sample `index.yml` file is as follows:

```
---
todo:
  timestamp: '20190510132311'
  sender: name.surname@ble.bankit.art
  subject: Machine Learning Test with SVM
```

This `index.yml` file is placed in the job root (see Sections 3.1.3 and 3.1.4).

3.1.3 The queue of jobs

Jobs are executed in a first-come-first-served basis using a queue.

As jobs move from one state to another ("enqueued for execution", "being executed", "completed") the position of the directory representing the job changes. In particular, the `/non-confidential/` directory (see Section 4.6) contains 3 subdirectories used to record the state of execution of all submitted jobs:

- `/non-confidential/todo/`: Contains jobs DS asked DS-bot to perform which DS-bot hasn't executed yet (if any); this directory implements the queue of jobs to be executed; the enqueuing order is given by the creation time of the job directory (and, redundantly but more SA-friendly, by the first portion of the directory name);
- `/non-confidential/doing/`: Contains the (single) job DS-bot is currently executing (if any);

- `/non-confidential/done/`: Contains all the jobs DS-bot has completed (if any).

At any moment, the `todo` and `done` folders thus contains [0...n] job subfolders ([0..1] for `doing`). The state of the job is updated (by the SA-bot) using the `mv` and `cp` system commands in such a way that there will never be two job directories inside `/non-confidential/` with the same name.

3.1.4 Job input and output

Jobs being executed can place the output of their work in two specific locations inside `/confidential/`:

- `/confidential/<JOB_ID>/results/`, a dedicated per-job directory where all the outputs of the current script meant for DS go; this output may or may not contain confidential information: It is meant for the DS, but it has to be analyzed and declassified first;
- `/confidential/cache/`, a directory shared by all jobs, where jobs can place the result of long or complex computations (e.g., BD queries, embeddings, etc.) that will be useful to subsequent jobs (to avoid re-doing work); this directory usually contains confidential data; however, it is not meant to ever exit the BLE. Also, jobs cannot assume they will find any specific intermediate result here; they must be capable of recomputing the missing pieces of information if they don't find them cached here.

The structure of the file system and the positions of all these information containers are opaque to the DS-bot scripts; scripts are passed the paths where to find information as environment variables. In particular:

- `PATH["input"]` is the path where the script being executed finds its own source code and all the accompanying inputs that came with the message from the DS (see Section 2.6.1); the script, while executing, has **read-only** access to this folder and cannot navigate out of it; this resolves to:
  - `/non-confidential/doing/<JOB_ID>`
- `PATH["output"]` is the path where the script being executed can place the results of its computations that are meant for the DS; the script, while executing, has **read-write** access to this folder and cannot navigate out of it; this resolves to:
  - `/confidential/<JOB_ID>/output/`
- `PATH["cache"]` is the path scripts being executed can place intermediate results into or read pre-computed results from; the script, while executing, has **read-write** access to this folder and cannot navigate out of it; this resolves to:
  - `/confidential/cache/`

3.1.5 Job execution

When a job with id `<JOB_ID>` is first submitted for execution, a properly named, dedicated directory (see Section 3.1.2) is created in the `todo` area (see Section 3.1.3); inside this directory are placed the submitted script and all its associated assets (trasmitted by the DS as explained in Section 2.6.1). In particular, there will always be (at the root level) a `script.py` file which is the entry point of execution.

When this job is the next one in the queue of execution (see Section 3.1.3) and DS-bot has finished all previous work, then the job is moved into `doing`; when the presence of a new job into `doing` is detected, the system does what follows:

1. Creates a new directory `\confidential\<JOB_ID>` in the Working area; the *ble_worker* user—the user associated to the service responsible for the instantiation of the ML job on ICD; see Section 4.10— is granted `read+write` access to it;

2. A trampoline code (C++/Python) is executed that spawns a fresh process inside an *isolated execution* environment;
3. Paths to the input and output directories the new process must use are stored into the `PATH` dictionary (see Section 3.1.4);
4. A Python interpreter is run against the entrypoint `\confidential\doing\script.py` and is passed the environment variables in `PATH`;
5. From this point on, control is transferred to the code written by the untrusted DS.

When the script completes its execution (successfully or after any error or uncatched exception), the associated process dies and its return value goes back to a DO-bot process for checking the non-confidentiality of results (see Section 2.5.3.2).

As we mentioned at point (2), it is of the uttermost importance that the code written by the untrusted DS is executed in a restricted, isolated environment, whereby the code in question can do no harm to the BLE environment nor communicate in any way with DS.

This is ensured via a series of technical devices, as explained in Appendix A.3.

3.2 Runtime checks

3.2.1 Mutex Access System - MAS

One of the most important actions SA-bot is responsible for at runtime is to ensure data confidentiality w.r.t. DS. All confidential (and all *possibly* confidential) information is to be made *unaccessible* from within the BLE environment as soon as the DS attempts to enter it.

The DS, properly authenticated via the enterprise AD, enters the BLE via SSH to configure/update DS-bot. Right before the SSH shell becomes responsive for DS, the following happens:

1. All processes implementing all bots (except **ble-SABOT**, the service responsible for confidentiality and integrity—check Section 4.10) are **terminated**
2. The encrypted disk containing ICD (mounted at `/confidential/`) is **unmounted**.

Note that this step renders any confidential information inaccessible to any user, not just to the interactive DS user, and is performed upon any SSH login attempt (not just those by DS). When the DS user closes his last shell, the reverse process takes place: Volumes are re-mounted and all bots are re-started.

Conceptually, all these operations are performed by SA-bot. In practice, they are performed by both the SA-bot which enforces the MAS by an event-driven control on logged-in users and by the SSH daemon, which runs as root and enforces the MAS by exploiting the Pluggable Authentication Module (PAM) configuration policy.

3.2.2 Integrity checks

The **SA-bot** periodically takes the initiative to check the integrity of the environment; two major checks are performed:

- The entire directory tree is checked against the nominal correspondences among permissions and users—see Section 4.6;
- The integrity of all scripts and source code executed by the trusted bots (DO-bot, SA-bot); this is done by comparing MD5 hashes of current scripts against the signed originals.

3.3 Event-driven architecture

The entire BLE architecture is event-driven. Two types of event exist:

- **External events**, triggered by either DS or DO, originating from outside the BLE; most of them are mail-based (see Section 2.6);
- **Internal events**, triggered by one of the bots and originating from within the BLE; most internal events consists in updates to the queue of jobs (see Section 3.1) through which software agents implicitly synchronize their operations.

The relevant events within the BLE and the corresponding actions are as follows.

External Event	Detected when	Detected by	Detected how	Action
New DS request to execute ML code in the BLE	Incoming message to the mailbox `ds.bot@ble.bankit.art` coming from `name.surname@ble.bankit.art`(everything else is ignored	`DS-bot`	`IMAP` protocol	`DS-bot` checks the incoming message for the expected syntax, then unwraps (and possibly unzips) its content and attachments into the proper position in the file system (i.e., enqueues a new job); acknowledgment message sent back to `name.surname@ble.bankit.art`
Clearance granted/denied by DO about a previously submitted case	Incoming message to the mailbox `do.bot@ble.bankit.art` coming from `do@ble.bankit.art` (everything else is ignored) that refers to some job outcome previously put on hold	`DO-bot`	`IMAP` protocol	Forwards the message content to `name.surname@ble.bankit.art` and to `do@ble.bankit.art` with or without stripping all non-metadata content from it, depending on the `DO` assessment—see Section 2.5.3.2

Internal Event	Detected when	Detected by	Detected how	Action
New job to process	(DS-bot stops running when todo area is not empty) or (todo area becomes non-empty when DS-bot is not running)	DS-bot	inotify API listening for new subdirectories created at /non-confidential/jobs/todo/ plus ps on process DS-bot	The oldest job in the queue (i.e., the subdirectory in /non-confidential/jobs/todo/ with the oldest timestamp in the name) is moved (mv) into the /non-confidential/jobs/doing/ area
Job ready to start	New (and unique) job appears in the doing area	DS-bot	inotify API listening for new subdirectories created at /non-confidential/jobs/doing/	The job is executed as explained in Section 3.1
Job completion	DS-bot not running and doing area not empty	DS-bot	inotify API listening for new subdirectories created at /non-confidential/jobs/doing/	Folder /non-confidential/jobs/doing/<JOB_ID> is moved into /non-confidential/jobs/done by SA-bot
New job outcome to check	New job appears in the done area	DO-bot	inotify API listening for new subdirectories created at /non-confidential/jobs/done/	Checks from Section 2.5.3.2 are applied to results in /confidential/<JOB_ID>/results/ and results are packed/compressed as mail attachments, and the mail is sent to name.surname@ble.bankit.art and to do@ble.bankit.art

4 Implementation details

The BLE and its inner working have been described at a very high level of abstraction, so far. Most of its primitive functionalities can be implemented in different ways, with different trade-offs. In this section, we:

- Provide specific **details** about how the components and behaviors of the BLE are actually implemented;
- Introduce all **security features** that make the execution of jobs inside the BLE "safe" from the confidentiality point of view;
- Show how the BLE is **embedded** into a specific **enterprise environment**; namely, within the [*omissis*].

4.1 Surrounding IT infrastructure

The IT infrastructure surrounding the BLE is supposed to have the following structure:

- There is a **Trusted enterprise Mail Server (TMS)** that supports LDAP groups and encrypted content via PKI certificates; this component is embodied by the [*omissis*];
- There is a **Trusted enterprise Data Base (TDB)** with sufficient security to hold and manage ECD. Encrypted sessions to read content from it are allowed; this component is embodied by an [*omissis*];
- There is a **Public Key Infrastructure (PKI)** capable of providing and managing credentials for every DO/DS user; it is integrated with the mail service and the authentication services; this component is embodied by the [*omissis*];
- There is a network security infrastructure in place capable of realizing and segmenting separate trusted network zones; in particular, there is a **Trusted Network Security Zone (T-NSZ)** where BLE is placed;
- There is a **Trusted Package Repository (TPR)** internal to the enterprise that mirrors all relevant Internet repositories and implements the protocols of the main package management systems (e.g., Python/PIP); this component is embodied by [*omissis*].

4.2 The host machine

BLE is hosted on a dedicated server, [*omissis*] and features a **NVIDIA Tesla P40** GPU for computationally-intensive deep learning tasks. It has the following main characteristics: [*omissis*].

4.3 Virtualization

A physical server like the one we employ could host more than one instance of our BLE; to accomodate for possible future BLEs, a virtualization server (*VMware EXSi 6.0 type 1 hypervisor*) is installed, capable to handle more than one Virtual Machine. The virtualization server is [*omissis*].

At present, the physical server has a single VM in which BLE runs. [*omissis*]

4.4 Network configuration and firewall openings

The BLE production environment lives inside an enterprise T-NSZ. All default services accessible through the network are disabled (and in any case unreachable as per T-NSZ), apart from those strictly required to allow the BLE to work: it is necessary to open specific firewall ports for those services to work. A thorough hardening process is executed by SA on the machine.

[*omissis*]

4.5 Setup and configuration via Ansible

The construction and configuration of all the BLE environments are automated: It is not necessary to interactively log into the system.

Of course, infrastructural prerequisites—such as the DNS, firewall configurations, the email server, and other technical systems and accounts—have to be already in place; they are managed via the enterprise Change Management process.

The automation scripts are part of the code base and are located in the top-level `/ansible` directory.

A brief description of the various phases of BLE installation follows. [*omissis*] Note that the automation scripts are organized to strictly segregate environment-specific parameters into a dedicated file written in YAML (a human-readable data serialization language). So, each BLE installation script is defined by the same Ansible code base, which is immutable and invariant across environments, plus a YAML file (the *inventory*) that defines its free parameters. As such, creating a new BLE installation only requires writing a new inventory YAML file and performing the aforementioned Change Requests.

4.6 Filesystem structure and permissions

The whole BLE application resides under the base directory [*omissis*]

The following table shows filesystem paths needed by BLE, bots, and human counterparties. For each path the corresponding permissions are shown. [*omissis*]

4.7 The Reserved Area

[*omissis*].

4.8 Encrypted RAM Disk

ICD are stored into an Encrypted RAM Disk (ERD) instantiated during the boot sequence (see step 1 in Section 4.9). ERD is created if and only if a test about the existence of a `/dev/ram` device is successful. At the creation of the ERD, a random key **k** is generated and overwrites the old version of the key inside [*omissis*]—if present. This makes us sure that, even if ICD are stored into a volatile RAM disk, it is impossible to access data of a previous version of ERD.

The randomly generated key **k** is then stored at [*omissis*] and used to LUKS-encrypt the RAM-disk device in such a way that the LUKS-encrypted device becomes ERD and can be mounted at [*omissis*].

Note: At the boot of BLE, a kernel parameter must be edited in order to define the maximum dimension (in bytes) for a block device to be instantiated in RAM. A reboot is needed after any change to such parameter.

4.9 Booting the “BLE core"

Booting the BLE environement consists of starting a sequence of services. This process is done *atomically*: If anything goes wrong in the boot sequence, the entire environment is shut down (“all works, or nothing works").

In particular, the boot sequence, that results in a functional *BLE core*, is as follows:

1. The **first service** sets up the Encrypted Ram Disk (ERD), described in Section 4.8;
2. The **second service** is responsible for launching the ETL process (Extraction-Transform-Load) on CD to populate the freshly created ERD, as described in Section 5;
3. The **third service** is the SA-bot, that cryptographically checks the consistency of the binary code of all other bots (see Section 3.2.2). Then, after checking that no DS is logged inside the BLE, SA-bot launches the DS-bot and the DO-bot.

At this point, the *BLE core* is up and running.

4.10 Instantiating bots

The following BLE-specific services run on the machine dedicated to the learning environment, and concur to realize most of its features:

- One service runs trusted code to implement **SA-bot** functionalities:
- **ble-SABOT**, a service which implements the most critical behaviors of **SA-bot**, namely those aimed at guaranteeing confidentiality and integrity; it runs as *ble_sa_bot* user (and is the only BLE-specific process to do so); its code is written in C++ and can be found here; its responsibilities are:
  - To launch and terminate at proper times all other processes and daemons in this list;
  - To implement the MAS (Mutex Access System) described in Section 3.2.1;
  - To perform the integrity checks described in Section 3.2.2.
- Three services with different concerns run trusted code to implement **DS-bot** functionalities:
  - **ble-DSBOT-TaskScheduler**, a service run by user *ble_ds_bot* which is responsible for managing the queue of jobs; its code is written in C++/Ruby and can be found here; its duties are detailed in Section 3.1.5.
  - **ble-DSBOT-MailReceiver**, a service run by user *ble_ds_bot* which is responsible for handling all incoming emails (see Section 2.6); its code is written in C++/Ruby and can be found here;
  - **ble-DSBOT-Worker**, a service run by user *ble_worker* which is responsible for the instantiation of an isolated process that executes DS jobs on ICD (see Section 3.1.5); its code is written in C++/Ruby and can be found here;
- One service runs trusted code to implement **DO-bot** functionalities:
  - **ble-DOBOT**, a service executed by *ble_do_bot* user that runs code meant to check the content of the output produced by the DS job (see Section 2.5.3.2). Its code is written in C++/Ruby and can be found here and is also responsible of communication with DO and job output clearance process.

4.11 Confining DS-bot

After the sequence of steps described in Section 4.9 completes, the control is passed to the untrusted code written by the DS.

This code runs into a fresh, dedicated, per-job system process, that is spawn as follows:

1. The job to be executed is moved into `doing` by **ble-DSBOT-TaskScheduler**;
2. **ble-DSBOT-Worker** recognizes the presence of a new job directory in `doing`;
3. **ble-DSBOT-Worker** executes a trampoline code (written in C++/Python) that spawns the new process in which the job will execute.

Even if in general we refer to a *benevolent* DS here, the measures we take to confine the actions of the DS job process are extreme, and well capable of confining the actions of most *malevolent* DSs as well. In this sense, we observe that DS is unable to modify or inject unauthorized software inside BLE environment or modify ICD and only authorized DS are able to submit ML tasks and to receive output (please refer to Appendix A.2)

First, we observe that the DS job is executed “inside" the ERD, so it obviously has the right to access ICD (this is the main objective of the entire BLE architecture). What we want is that the code written by the untrusted DS should be free to write whatever results in the dedicated output folder of the ERD, but *nowhere else*. It should not be able to exit this folder in any way, which is equivalent to say that DO-bot will see and check whatever information is produced as output by DS-bot.

This property is attained by exploiting the following (to some extent redundant or orthogonal) technical solutions:

1. **File System**: The file system permissions are configured in such a way that **DS-bot-Worker-user** is able to read and write **ONLY** into the CD and ERD directory trees (see Section 4.6);
2. **Mount Namespaces**: The trampoline code executed by SA-bot forks and executes a script activating the **Mount Namespace** Linux feature. Mount namespaces provide an OS level isolation of the list of mount points seen by the processes. Thus, the processes in each of the mount namespace instances will have a less privileged view of the system. In the BLE the trampoline code executed by SA-bot declare a mount namespaces that isolates DS jobs from any networking channel, mail service, database service, system service, and file system access other than the CD and job directory trees;
3. `Chroot` **and yet Another Mount Namespaces**: Within the namespace configured as described in point (2), a `chroot` is performed to further isolate the job environment in terms of file system visibility. Moreover, in order to prevent `chroot` evasions, a new mount namespace is called inside the `chroot`, to prevent job from executing a `chroot` (this is the main evasion trick);
4. **Virtual Environment**: The task is executed inside a DS editable **Virtual Environment**; this is not a protection feature by itself but it is a way to improve the isolation of the Python packages importable by the DS job, while making the process auditable;
5. **Restricted Python**: At last, the DS job is executed inside a Restricted Python environment. The restricted python environment is setup to prevent the DS job code from calling OS primitives that are useful to operate `chroot` evasion tricks, such as changing directory or working path.

All the execution isolation features described in this section are implemented by immutable configuration scripts executed inside SA-bot code and are part of the *BLE core*.

4.12 Checking mails

To facilitate runtime checks of all incoming and outgoing mails, all users associated with DS, DO and SA roles are added to dedicated LDAP groups:

- Users associated with SA role are added to [*omissis*];
- Users associated with DS role are added to [*omissis*];
- Users associated with DO role are added to [*omissis*].

The Data Owner is responsible to grant and remove user access from the Data Owner and Data Scientist Group. Then, the two main checkpoints are as follows:

- **Avoid job submissions by unauthorized users**: Messages from `name.surname@ble.bankit.art` to `ds.bot@ble.bankit.art` submitting a new job need to be signed by DS: DS-bot verifies the signature and then looks up the LDAP group [*omissis*] to check if the sender is actually a lecit DS; this avoids job submissions from unauthorized (possible non-enterprise, external) senders. If any of these checks fail, then DS-bot rejects the message; otherwise, it enqueues the new job.
- **Avoid output cleareance by unauthorized users**: When DO-bot receives an email claiming to contain a confidentiality decision by DO, DO-bot verifies the signature of DO and then looks up the LDAP group [*omissis*] to check if the sender "on behalf of" DO is actually a legit DO; this avoids the reception of counterfait clearance email. If any of these checks fail, then DO-bot rejects the message; otherwise, it continues with the workflow.

DO-bot is the only bot sending possibly confidential email out of the DIB; it uses certificates from [*omissis*] to sign and encrypt them, as mentioned in Sections 2.6.3 and 2.6.4.

4.13 Exposing the state of ingestion through the dashboard

The ICD repository functionalities and status are made available to DS and DO via the control dashboard (see Section 6), that is fed by a Key-Value store (KVS), which in turn makes use of an inotify mechanism.

The ETL service and timer natively publish their status on a D-Bus channel. The information thus needs to be propagated to the KVS store. The component dedicated to perform this adaptation is `ble-dbus-adapter.service`, a Python program that:

- Subscribes to D-Bus change messages;
- Filters them;
- Publishes the events on the KVS in the format specific for the application.

The resource consumption is extremely low, because there is no polling involved. Like the other components, ble-dbus-adapter is installed in an isolated virtual environment and runs as a low privilege process under `systemd` supervision.

4.14 Logging

Each output of a system service managed by `systemd` is collected and visualizable by querying another system-level service named `journald`. Messages coming from services are assigned to `INFO` log-level in `journald` by default; please, check Section 4.9 to remind which system services are referred in the following as *BLE core*.

BLE Logging system uses `journald` convention for messages as follows:

```
<LOG_LEVEL> Log Message information
```

where `LOG_LEVEL` is a number from 0 to 9 representing the following log levels:

- 0: `EMERG`, a very big issue happened, the BLE dedicated physical server can be dangerously damaged;
- 1: `ALERT`, a big issue happened, the *BLE core* can be damaged and no longer functional;
- 2: `CRIT`, an issue happened that prevents *BLE core* to continue functioning;
- 3: `ERR`, an error occurred that prevents a specific functionality to complete;
- 4: `WARNING`, something expected by develper occurred that could produce unexpected results on a specific fuctionality;
- 5: `NOTICE`, this is a message that must be taken into account if something is not working well;
- 6: `INFO`, this is a standard message that the developer wants to log;
- 7: `DEBUG`, this is a standard message that only the developer needs to see.

In the following, we provide further details on available logging enterprise systems.

4.14.1 Enterprise Security Information and Event Management (SIEM)

[*omissis*]

4.14.2 Enterprise Technical Monitoring

BLE physical server is stored into the [*omissis*]. Basic controls are provided such as operating system controls, file system capacity control or check on url status of BLE dashboard.

4.15 Running and manteinance

In this section, we briefly summarized some [*omissis*]

5 Data Ingestion

Data Ingestion is the process whereby (a portion of) the confidential information contained in ECD is transported into the ICD, ready for use by the DS-bot. Data Erasure is the process whereby any such information is erased from ICD.

This process is highly dependent on the structure and content of the actual, external repository of confidential information. [*omissis*]

5.1 Structure of the external CD repository

[*omissis*]

5.2 Triggering the Data Ingestion/Erasure process

Data Ingestion takes place in two different circumstances:

- (DI-i) **automatically**, when the BLE is booted, or
- (DI-ii) **on-demand** (manually), by DS.

Similarly, **Data Erasure** takes place in two different circumstances:

- (DE-i) **automatically**, due to DS inactivity, or
- (DE-ii) **on-demand** (manually), by DS.

The ICD repository is available to DS-bot after the *Data Ingestion* phase completes (it may takes minutes to hours).

The on-demand commands for **Data Ingestion** and **Data Erasure** are triggered by DS using the control dashboard (see Section 6, which contains two dedicated controls (check Sections 6.1.4.1 and 6.1.4.2).

The "DS inactivity" mentioned in case (DE-i) is defined as follows: *72 hours have passed since the last DS job completed its execution*.

5.3 Light night-time ingestions

The ECD Repository [*omissis*] is a shared infrastructure employed by crytical business processes. The Data Ingestion operation should thus be carefully engineered to be light on the DB, and to avoid to break the SLA in terms of availability (to other critical applications). Two ingredients are used to limit these risks:

1. **No server-side computation** happens in the external DB, and the only kind of load imposed on the DB server is sequential I/O; e.g., queries on External DB do not perform JOINs or other costly operations, just plain SELECT *. The query does not affect DB operations and the active Oracle fine-grained auditing process.
2. The ingestion phase is only ever **activated at 2:00 AM**, outside the operating time window of any other applications. If the Data Ingestion is triggered at a different time, what happens is that a timer is scheduled to fire a 2:00 AM and start the actual ingestion.

The technical implementation of (2) builds on the architecture described in the previous section, and makes use of `systemd` timers. `Systemd` offers a timer functionality similar to the well known `cron` service, with the added advantage of being integrated with the system and being observable via `D-Bus`. In practice, when the user clicks on the control dashboard button that requests the **Data Ingestion**, the `ble-etl.timer` service is started. The timer is set to run `ble-etl.service` at 2:00 AM, reporting its status to the control dashboard.

5.4 ETL: Extraction, Transformation, Loading

The core step of the Data Ingestion phase is the execution of the ETL process: *Extraction - Transformation - Loading*. This process is responsible for extracting, transforming, and loading (a portion of) the ECD repository into the ICD repository.

As we will see, the ETL process makes use of a `SQLite` database in the phase of Extraction, and of `pandas` libraries in the phase of Transformation.

The overall ETL process is implemented by a single `etl.py` script, that is not run directly. It is, instead, executed as a system service `ble-etl.service` under `systemd` supervision. The following properties are guaranteed:

1. **At most one** ETL process is running at a given time;
2. The service status (waiting, running, error) is **reliably observed** via a D-Bus subscription.D-Bus is a user-space IPC and message bus service used by default in many Linux distributions;
3. The supervision mechanism can guarantee **preconditions and postconditions** (e.g.: A certain file can only exist if a service is running) in a reliable way;
4. There is **native integration** with the system logging facilites, including sending logs to remote locations for security inspection;
5. It is easy to reliably reduce the privilege level of the service: Root rights are only needed for starting and stopping the service. The actual process always **stays unprivileged**.

Property (1) protects against correctness and performance issues. Property (2) and (3) are particularly important in an event-based architecture like the BLE: The alternative would be polling the system to obtain information, which would impose long latencies on the control dashboard and would be prone to inconsistencies. Property (3) and (4) are useful from a security perspective, even more so considering that `ble-etl.service` interfaces with ECD.

As a requirement for the ETL process to run, a csv folder must be already created at path `write_fil e_path` specified as an argument of `etl.py`.

The function `main` of `etl.py` encompasses the usual three phases of an ETL process:

1. **Extraction**: Data are extracted from the ECD and loaded into an `SQLite` database with equivalent schema (and residing in the ERD—See Section 4.8); [*omissis*];
2. **Transformation**: From the local `SQLite` data structures, data are loaded into `pandas` and transformed in so as to build a new data model;
3. **Loading**: The new data model is saved into the final ETL output format, which is a (set of) CSV file(s).

Note that by their nature, these 3 phases are (partly) *designed by the DS*, although under the supervision of DO. In the present case of study, a typical ML job includes classification of reports on suspicious financial transactions based on their purpose or based on the risk rating. The DS thus picks which data is to be loaded into the ICD to perform meaningful ML tasks, and a validation by DO follows. The DO tries to minimize the amount of information "leaking" from ECD to ICD, whereas DS hopes for a very large transfer, because it is not always clear a-priori which data a ML approach would benefit from. As a compromise, DS avoids to request the loading of data tables that are clearly out of scope. Obviously, any change to the process of data extraction from the ECD will necessarily be approved by DO.

More details follow.

5.5 Extraction

The phase of **Extraction** aims at extracting useful data from the ECD reporitory. The ECD can contain any kind of confidential or micro data, useful for statistical analysis performed by DSs or ERs.

[*omissis*]

[*omissis*]

5.6 Transformation

In this phase, the new data model is built using the six `SQLite` tables. The tables are loaded in memory into `pandas` data structures. Some transformations are applied, using the `pandas` libraries. [*omissis*]

5.7 Loading

[*omissis*]

5.8 Tagged ICD: the Named-Entity Recognition (NER) model

As seen in Sections 2.5.1 and 2.5.3, a Named-Entity Recognition (NER)-based preprocessing of unstructured data is performed after the ETL phase to ehnance input/output data leak checks, as in Figure 13.

The applied NER uses a transformed model of BERT-XLARGE-MULTILANGUAGE optimized for use in both GPU and CPU. The fine-tuning of the model has allowed to maximize recall on Wikipedia datasets.

The NER phase takes place after of the data ingestion phase. First, the model is initialized and the processing of the unstructured data [*omissis*], is started (1000 at a time). The result of the operation creates a list of possible named entities which is in turn filtered by keeping only the unique data and mixing names and surnames. The resulting set of words is saved in a CSV file in [*omissis*].

The input and output checks are carried out on the reserved fields of [*omissis*].

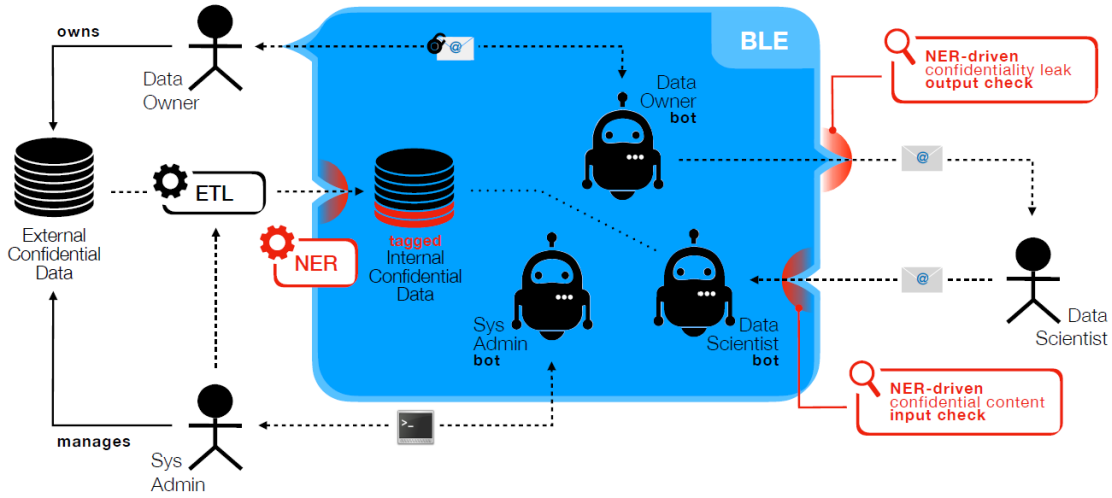

Figure 13: NER phase.

5.9 Data Erasure

This process clears the entire content of `.../confidential/`, and in particular:

- The ICD Repository, i.e., the output of the ETL operation, i.e., the contents of `/confidential/csv/`;
- The working space, i.e., the output results of any jobs ever executed and of any cache ever created, i.e., the content of `/confidential/db/` and of `/confidential/<JOB_ID>/` for all `JOB_ID`.

As a result, the BLE is made entirely free of confidential information. Data Erasure is executed by running the process meant to (re)create the ERD (see Section 4.8: A new random key will overwrite the old one (permanently destroying it), and this key will be used to format the ERD, removing any trace of previous data. Then, a new EXT file system will be formatted on the freshly recreated ERD.

6 The control dashboard

Non-critical communications path (5) is implemented as a **web service with websocket connection (and a supporting REST API)** exposed from within the BLE by the process/agent associated with DO-bot; it can be accessed by authorized DS in Single Sign On (and optionally by DO and SA) over an encrypted channel (SSL) after PKI-based authentication.

An HTML5 interface is rendered and served by an application server resident on the same machine as the BLE but external to it. So the directional information boundary is crossed at *the connection between the API endpoint (inside BLE, at SA-bot) and the web server (outside BLE)*.

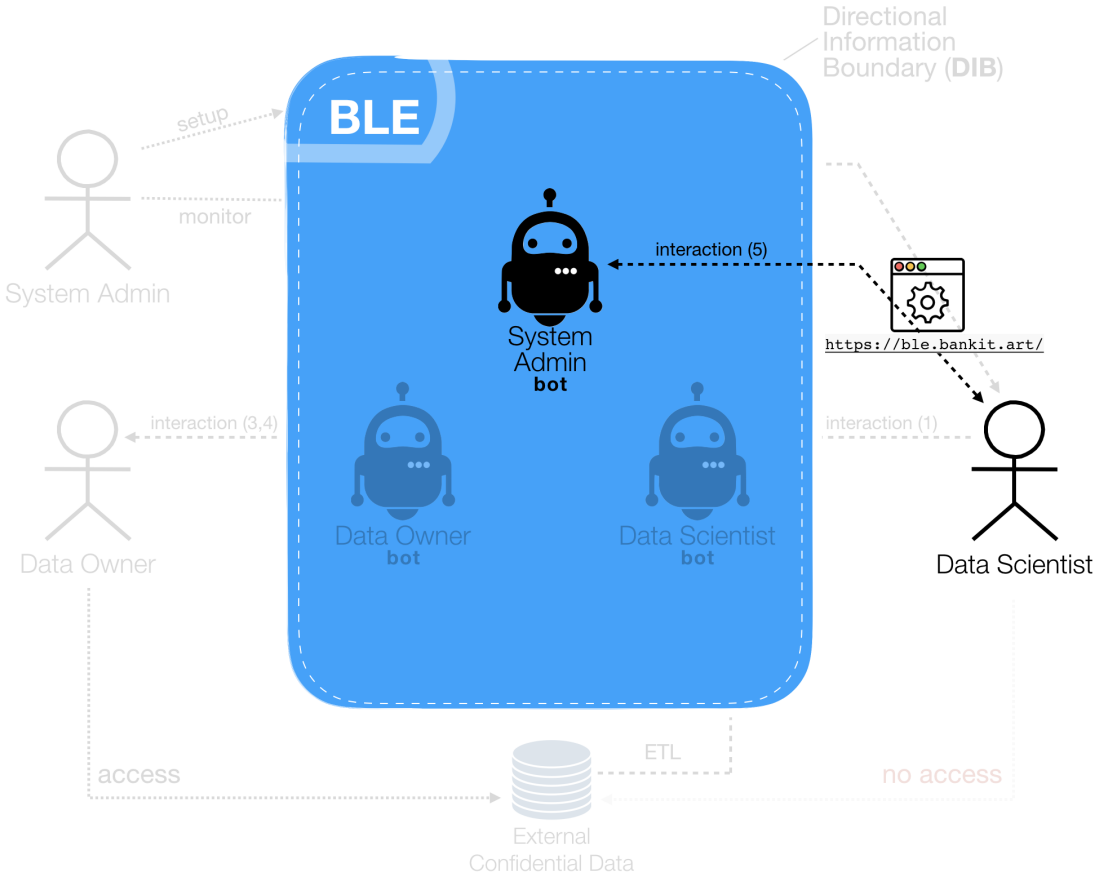

Figure 14: DS interaction with BLE control dashboard.

The control dashboard is available at `http://ble.bankit.art/`.

6.1 Dashboard usage

The interface has two main areas:

- A fixed-with leftmost column menu to navigate dashboard pages;
- A variable-width main area.

A screenshot of the control dashboard is available in the following figure.

Figure 15: BLE control dashboard.

6.1.1 Menu

The menu hosts user information and navigation menu.

User information are derived from the user id yielded by Apache (see Section 6.4). Actual information shown (user real name, user picture) is locally stored in the codebase for a restricted set of users (in `ui/ble-ws/src/services/user-db.ts`); users not present in that database only get default values (basic user id and default avatar).

The navigation menu has three entries:

- Dashboard;
- Completed jobs;
- Administration panel.

6.1.2 Dashboard

The dashboard shows live updates of BLE internal status (see Section 6.2). Nevertheless, it hosts some basic function to interact with BLE without exposing confidential information.

Dashboard information is divided in sections:

- `Environment info`: Gives mostly static information about host environment;
- `Machine load`: Exposes information about the load of the host on which BLE runs, be it disk, memory or CPU;
- `Datasets`: Shows some non-confidential metadata about the confidential data stored in the BLE;
- `Bot status`: Illustrates the actual liveness of BLE bots;
- `Job queue`: Shows the actual state of BLE as a machine-learning machine (as in the Figure 16).
  - `Scheduled jobs` is the queue of jobs sent through email that are waiting to be executed. It hosts three buttons, as shown in the picture below: `Cancel queue` (which erases all jobs in the queue, `Pause queue` (which stops the scheduling of jobs to the DS-bot) and `Resume queue`;

  - `Running job` is a widget that shows in real time which ML job is running and, optionally, includes a progress indicator;
  - `Completed jobs (last day)` shows the list of complete jobs, independently of how they ended up (success, failure, forwarding to Data owner, ...), limited to the previous 24 hours;
- `Confidentiality integrity` shows information relevant for BLE data confidentiality, like where the confidential volume is mounted (on ram or disk) or DS access.

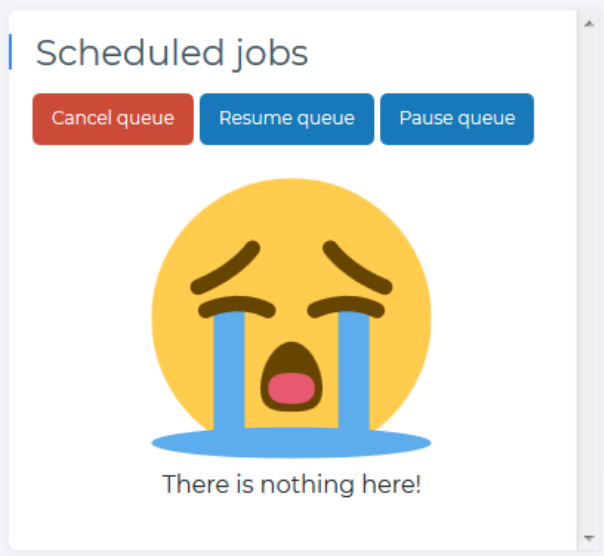

Figure 16: BLE Scheduled jobs status.

6.1.3 Completed jobs

This page is just like the `Completed jobs` widget in dashboard section, but with no time limits.

6.1.4 Administration panel

The administration panel hosts the most important commands allowed to DS, DO and SA to manage BLE data, as it is described in the following sections.

6.1.4.1 ECD data ingestion

The panel in Figure [*omissis*] allows you to schedule the ingestion of the ECD data for 2:00am. By clicking on the `Schedule` button, the ingestion is scheduled. By clicking on the `Unschedule` button, the schedule is canceled and no modification of BLE data occurs.

6.1.4.2 Erase BLE data

The panel in Figure 17 allows the user to completely wipe data in job directories and working area. When clicking the `Erase BLE data` button, the erase process is started immediately causing the deletion of confidential information and the restart of bots.

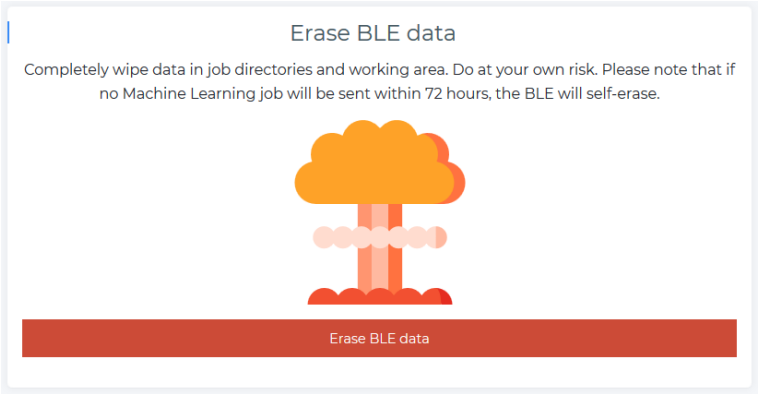

Figure 17: BLE Data Erase.

6.2 Dashboard Information

The dashboard exposes information about the environment running inside the BLE. What the dashboard knows is deliberately limited by BLE permission setup and the actual implementation of BLE UI web service code.

As of today, the information shown in the dashboard is shown below, completed with examples from the development host:

[*omissis*]

6.3 Encryption

In order to guarantee confidentiality and integrity of the communication between the web browser and the control dashboard [*omissis*] all the traffic is encrypted via TLS. This also guarantees that no tampering is possible.

6.4 Web authentication

Access to the UI web interface is integrated with the enterprise AD infrastructure, and only allows authenticated connections. Authentication happens transparently via Kerberos: A front-end web server (Apache) is placed upstream the nodejs application. The Active Directory administrator placed a Keytab file in it, which allows the web server to act as authentication gateway, granting access to network resources only if a workstation is able to securely identify itself as being part of the [*omissis*] domain. In this case, the user's unique identifier is passed downstream in the `x-remote-user` HTTP header [*omissis*]

In the web service (on certification and production environments), the authorization header is checked for existence, and and an error is thrown if it is not present:

```
let kerberosUser = headerList['x-remote-user'];
if (!kerberosUser) {
  next(new Error('No "x-remote-user" information in header'));
}
```

From the point of view of a user connecting from his workstation, the identification is transparent and automatic.

6.5 Web authorization

Once a user is authenticated to the UI web interface, the authorization phase takes place based on user roles. User roles are assigned based on the LDAP group definition for the specific environment. The roles

are defined in this way:

[*omissis*]

Note: the UI shows even an *ART member* badge for those users with role [*omissis*]; this role has nothing to do with authorization.

Upon user connection, the web service looks up in LDAP that the user actually belongs to one of the allowed groups and assigns him/her the proper roles.

Belonging to one group or another makes no difference with respect to UI features or permissions.

LDAP is queried using `id` command as follows: `id -Gn ${user}`, where user is the user id in the `x-remote-user` header. The `id` command works in the current host setup and requires no further credentials to be set. Since the `id` command can show lags which would reasonably be noticed in the UI, received values are cached for a short time. This time is hardcoded as:

```
const CACHE_LIFETIME_IN_SECONDS: number = 1 * 60 * 60; // 1 hour
```

Upon connection, if one or more of `BLE_ADMITTED_ROLES` do notappear in the groups mentioned by LDAP, a server error is thrown (`BLE_ADMITTED_ROLES` are the roles mentioned in the table above).

```
let kerberosUser = headerList['x-remote-user'];
const userInfo = await self.userInfo.getInfo(kerberosUser); // Takes info from LDAP
// Create UI specific roles to be shown
userInfo.roles = uiRolesFromLDAPGroups(userInfo.groups);
// Matches platform-specific groups to app roles
// Check if user is allowed inside
if (oneOfIsIn(BLE_ADMITTED_ROLES, userInfo.roles)) {
  // Register user for later
  socket.bleUser = userInfo;
} else {
  next(new Error('You are not allowed to see this.'));
}
```

6.6 Interface between BLE and ble-ws

Every communication between `BLE` and `ble-ws` happens through a file-persisted Key-Value store (KVS) in `/non-confidential/kvs/` directory. Each file in the directory represents a key (filename) and has a value (the content). The key/filename has a specific extension which tells the file content format (no extension means plain utf8 text).

This file-persisted KVS allows to share a common state between `BLE` and `ble-ws` without traversing BLE security boundary: `BLE` writes out only a minimal set of information which is in turn read by `ble-ws`, the same goes for `/ble-ws` and `BLE/`.

Here are the implemented keys:

[*omissis*]

`user-sessions.yml` has been introduced in commit [*omissis*], eventually freeing the dashboard from checking on its own the established sessions; now `ble-ws` relies on the information actually detected by BLE (which are actually used for access control of BLE features). Before this change, `ble-ws` executed on its own both `netstat` and custom `get_active_users` commands to fill in required information, actually doubling BLE checks (which used the same set of commands).

The UI (and so the UI technical user) is able to directly launch specific commands which are hosted in `ui/commands` directory. If the command accepts command-line parameters from the client, these parameters are checked. These commands are:

Command	What it does	Parameters checks
`$ cancel-active-job.sh`	Deletes everything in `${JOBS_DIRECTORY}/doing/*`	None
`$ cancel-queued-job.sh <job>`	Deletes everything in `${JOBS_DIRECTORY}/todo/<job>`	is regex-checked against the "timestamp-email" pattern (e.g. `name.surname@ble.bankit.art`, see Section 3.1.1)
`$ clear-todo-jobs.sh`	Deletes everything in `${JOBS_DIRECTORY}/todo/*`	None
`$ delete-done-job.sh <job>`	Deletes everything in `${JOBS_DIRECTORY}/done/<job>`	is regex-checked against the "timestamp-email" pattern (e.g. `name.surname@ble.bankit.art` , see Section 3.1.1)
`$ erase-data.sh`	Literally brings down the system recreating the encrypted volumes and clearing non-confidential directory	None
`$ push-job-offset.js <job> <offset>`	Changes the schedule timestamp of the job by a positive/negative offset	is regex-checked against the "timestamp-email" pattern (e.g. `name.surname@ble.bankit.art`, is parsed and checked to be a number
`$ clear-done-jobs.sh`	Deletes everything in `${JOBS_DIRECTORY}/done/*`	None
[*omissis*]	Schedules download of ECD	None
[*omissis*]	Clears any schedules of the download of ECD	None

6.7 BLE UI web service component, aka `ble-ws`

BLE exposes a web user-interface which contains jobs status, machine load and basic commands to support the actors in their work.

The web service is available on port 4001 and exposes all dashboard information and commands through a web socket interface, available on the same port. Upon connection, authentication is performed and authorization is verified as described before, so that access is given only to users pertaining to specific groups.

The web service exposes even an HTTP interface. The API endpoints are not protected and are publicly accessible. To be publicly accessible, the HTTP server has Cross-Origin Resource Sharing (CORS) enabled. The API endpoints are: [*omissis*].

6.7.1 Health endpoint

The health endpoint serves the sole purpose of yielding liveness information. It can be used for checking whether the server is up and running normally, and only contains basic stats. In the future, it may be used by an external service as and endpoint for health checking.

Here follows an example response on the development server:

[*omissis*]

When a command is executed in the UI, this health endpoint shows the last executed command and the respective log; for example, one would get:

[*omissis*]

6.8 BLE UI web client component, aka `ble-client`

BLE client is an Angular 7 application; it's compiled to static HTML/JS/CSS files which are in turn deployed to a specific folder served by Apache static file server on port 80. The application is available at [*omissis*].

While Apache identifies the user (see Section 6.4), it poses no limitations to the content behind it, this meaning that both the client application and the web service are reachable through Apache. This feature allows to let the client manage all the errors on the client application and gracefully manage any errors (client and server-side) that may happen.

6.9 Dashboard Logging

BLE client application does not log anything to development console in the browser, apart from warning and error messages that may happen. The web service, on the other hand, outputs to the console some log messages referred to specific meaningful events. A typical log from startup (extracted with `journalctl -u ble-ws` on the development host) is available in Appendix B (edited for clarity).

Appendix A. Summary of Security Features

We introduced and discussed several security features of the BLE aimed at preventing confidential data from leaking to the DS or to third parties. They are listed here for quick reference.

A.1. Security features meant to protect data as they enter the BLE and are temporarily stored in it

1. **Secure Data Ingestion**: ECD coming from the ECD are transferred via a scheduled ETL process (check Section 5.3).
2. **Limited ETL process**: The Data Ingestion process writes only part of the ECD into a specific area of the file system (check Section 5.4).
3. **Encrypted RAM Disk**: ICD are stored in an ERD with a random generated key stored in a reserved, root-only area (check Section 4.8).
4. **ERD Managment**: The random generated key of the ERD is overwritten each time the ERD is created/re-created (check Section 4.7).
5. **ERD auto-Erasure**: All confidential data are deleted if the BLE is not used for a long time (72 hours) (check Section 5.9).

A.2. Security features meant to ensure the integrity and confidentiality of the email-based workflow

1. **Active Directory enforcement**: Only users belonging to a specific authorized group are able to send mail to the BLE. BLE checks this at application level when a mail is received (check Sections 2.6 and 4.12).
2. **Enterprise LDAP and enterprise PKI authentication enforcement**: User authorized to send mail to the BLE as described in point 1 must sign with their PKI certificate any mail sent. BLE checks this at application level (check Sections 2.6.1 and 2.6.4).
3. **Encrypted DO $\longleftrightarrow$ DO-bot comunications**: Any mail sent to DO, that may contain CD, is encrypted with a specific key owned by the DO-bot user, generated by the enterprise PKI and stored in a reserved area (check Section 2.6.4).

A.3. Security features meant to ensure the integrity of the execution environment

1. **Immutable bots**: The BLE is implemented by a set of bots (SA-bot, DS-bot, DO-bot) whose code is immutable, distributed through approved `rpms`, and upgradable only through enterprise-approved changes. The SA-bot continuously checks for integrity of all other bots (check Section 3.2.2).
2. **Execution isolation**: The DS jobs, that is the only mutable part of the environment and the only part coming from an untrusted actor, are executed in a sandbox in which the job execution is able to access CD but is isolated from the entirety of the BLE. The job may write only to a specific location of the file system, and all its output is checked by DO-bot. This isolation is obtained through:
    - **File System Permissions**: The process that executes the DS's jobs may write only in a specific area (check Section 4.6).
    - **Network Configuration**: The process executing DS jobs has no access whatsoever to the network (or to the Internet).

- **Mount Namespaces**: Directives to reduce namespace visibility for the DS process are applied. The process is not able to access networking, mailinng, and system services (check Section 4.11).
- **Chroot**: Chroot inside the mount namespaces and new mount namespace to disable another `chroot` (check Section 4.11).
- **Virtual Env**: Python Virtual Env is used to ensure isolation of Python modules installed by DSs (check Section 4.11).
- **Restricted Python**: The usage of potentially dangerous OS primitives is limited at byte-code level, (check Section 4.11).

A.4. Security features meant to check the input script and sanitize the output of every DS job

DS input script are inspected by DO-bot before their execution (check Sections 2.5.1, 2.5.3 and 5.8):

- **NER-based input script check**: DO-bot checks raw strings in the script against the named entities in the ICD.

DS jobs, upon completion, are evaluated by the DO-bot (check Section 2.5.3.2):

- **Automatic checks on Confidential Fields**: DO-bot checks job results against tabu words present in sensitive fields in CD.
- **Automatic checks on Throughput**: DO-bot checks the dimension of the results in terms of outbound compressed size per unit of time.
- **Automatic checks based on Templates**: DO-bot checks the output against templates of common ML output results.

A.5. Security features meant to ensure safe interaction with the Dashboard

1. All the traffic to/from the Dashboard is **encrypted** via TLS (check Section 6.3).
2. **Authentication** to the UI web interface happens through (enterprise) Kerberos (check Section 6.4).
3. **Authorization** to the UI web interface happens through (enterprise) LDAP (check Section 6.5).

Appendix B. Example of Dashboard Log from startup

Please note that timestamps may be discontinuous since lines have been changed at different times due to documentation updates.

[*omissis*]

Appendix C. List of Technical Features

- **External Confidential Data (ECD)**: It is an authoritative repository of Confidential Data that lives somewhere outside the BLE.
- **Trusted enterprise Data Base (TDB)**: It contains ECD. Encrypted sessions to read content from it are allowed; this component is embodied by [*omissis*].
- **Internal Confidential Data (ICD)**: It is the internal encrypted temporary version of ECD, which is copied, fully or in part, via a carefully tuned and secured process of Extraction - Transformation - Loading (ETL), over an encrypted, secure channel.
- **Encrypted RAM Disk (ERD)**: ICD is copied on a volatile, encrypted copy, residing in RAM, which is erased in two circumstances: (i) the DS declares her/his job is temporarily over, or (ii) a certain amount of time passes since the last interaction with the DS. ERD is created if and only if a `/dev/ram` device exists. ERD is mounted at `/confidential/`: The underlying mount point has the immutable flag set, to ensure that nothing is written to the underlying cleartext file system in case a misconfiguration happens. ERD uses a random generated key, stored in the `/reserved/` area.
- **In- memory LUKS-encryption**: It is a standard for Linux hard disk encryption. It is used to create ERD via the random generated key.
- **Mutex Access System (MAS)**: It is the mechanism that allows the DS, properly authenticated, to enter the BLE via SSH to configure/update DS-bot. Right before the SSH shell becomes responsive for DS, all processes implementing all bots (except **ble-SABOT**) are terminated and ERD is unmounted. The DO and SA, otherwise, properly authenticated, are able to enter the BLE via SSH too, but no processes are terminated and ERD is not unmounted.
- **Pluggable Authentication Module (PAM)**: It is the policy exploited by SSH daemon to enforce MAS.
- **Python Virtual and Restricted Environment**: It is used to ensure isolation of Python modules installed by DS. It is created with the –always-copy option, this Virtual Environment does not share any components with the system Python.
- **Restricted Python**: It is a limitation of the usage of potentially dangerous OS primitives at byte-code level; the restricted primitives are useful to operate `chroot` evasion tricks, such as changing directory or working path.
- **Mount Namespaces**: They provide an OS level isolation of the list of mount points seen by the processes. Thus, the processes in each of the mount namespace instances will have a less privileged view of the system. In the BLE the trampoline code executed by SA-bot declare a mount namespaces that isolates DS jobs from any networking channel, mail service, database service, system service, and file system access other than the CD and JOB directory trees;
- **Chroot and yet Another Mount Namespaces**: A `chroot` is performed to further isolate the DS environment in terms of file system visibility. Moreover, in order to prevent `chroot` evasions, a new mount namespace is called inside the `chroot`, to prevent ML job from executing a `chroot` (this is the main evasion trick).
- **Microsoft Exchange Mail Server**: It is the Trusted enterprise Mail Server (TMS); it supports PKI and LDAP group. All human actors and bots interact with ERD via TMS.
- **Lightweight Directory Access Protocol (LDAP) & Active Directory enforcement**: Only users belonging to a specific authorized group are able to: (i) send mail to the BLE, (ii) enter the BLE via SSH, (iii) access to UI web interface. Three LDAP groups are created: DO group, DS group, SA group.
- **Public Key Infrastructure (PKI)**: PKI is capable of managing credentials for every DO/DS user; it is integrated with the mail service and the authentication services. User authorized to send mail

to the BLE must sign with their PKI certificate any mail sent. BLE checks this at application level. Furthermore, any mail sent to DO, that may contain CD, is encrypted with a specific key owned by the DO-bot user, generated by the enterprise PKI and stored in a reserved area.

- **Trusted Network Security Zone (T-NSZ)**: It is a security zone, placed in [*omissis*].
- **Trusted Package Repository (TPR)**: It is the enterprise repository that mirrors all relevant Internet repositories and implements the protocols of the main package management systems (e.g., Python/PIP).
- **Physical Host Machine**: The server is an [*omissis*] and features a NVIDIA Tesla P40 GPU for computationally- intensive DL tasks.
- **Smart Temporized Extract-Transformation-Load (ETL)**: The data Ingestion process writes only part of the CD into a specific area of the file system; the part is made by certain columns of certain tables, according to DS needs and DO clearance.
- **Full logging on D-Bus**: The ICD repository functionalities and status are made available to DS and DO via the control dashboard, that is fed by a Key Value Store (KVS), which in turn makes use of an inotify mechanism. The ETL service and timer natively publish their status on a D-Bus channel and then propagated to the KVS store.
- **Enterprise Kerberos**: It provides authentication to the UI web interface.
- **Transport Layer Security**: It provides encryption for all the traffic to/from the UI web interface.
- **Configuration of BLE**: The construction and configuration of BLE is automated via *Ansible* scripts
- **SIEM & logging**: All relevant - from a security point of view - events from system services (DS-bot, DO-bot and SA-bot) are sent to enterprise Security Information and Event Management.

Appendix D. Acronyms & Abbreviations

AD Active Directory

AI Artificial Intelligence

API Application Program Interface

APP Business Application

BLE Blind Learning Environment

CD Confidential Data

CPU Central Processing Unit

CSS Cascading Style Sheets

CSV Comma Separated Value

DB Database

DDL Data Definition Language

DEV Software Developer

DIB Directional Information Boundary

DL Deep Learning

DNS Domain Name System

DO Data Owner

DS Data Scientist

ECD External Confidential Data

ER External Researcher

ERD Encrypted RAM Disk

ETL Extracting Transforming Loading

EXT Extended File System

FMB Functional Mailbox

GPU Graphics Processing Unit

HE Homomorphic Encryption

IAAS Infrastructure As A Service

ICD Internal Confidential Data

IDP Intrusion Detection Prevention

iLO Integrated Lights-Out

IMAP Internet Message Access Protocol

IN Input message

IPC Inter-Process Communication

IS Information System

JS JavaScript

KVS Key Value Store

LDAP Lightweight Directory Access Protocol

ML Machine Learning

MAS Mutex Access System

MSG Data package

NDA Non-Disclosure Agreement

NER Named-Entity Recognition

NSZ Network Security Zone

OUT Output message

PAAS Platform As A Service

PAM Plugglable Authentication Mode

PCI Peripheral Component Interconnect

PII Personally Identifiable Information

PKI Public Key Infrastructure

PPSA Pre-Production Security Assessment

NLP Natural Language Processing

NN Neural Network

RAM Random Access Memory

Regex Regular Expression

SA System Administrator

SAAS Software As A Service

SF Server Farm

SPI Sensitive Personal Information

SSH Secure Shell

SSL Secure Sockets Layer

TD Test Data

TEE Trusted Execution Environment

TDB Trustred enterprise Data Base

TMS Trusted Mail Server

T-NSZ Trusted Network Security Zone

TPR Trusted Package Repository

UI User Interface

VM Virtual Machine

YAML YAML Ain't Markup Language