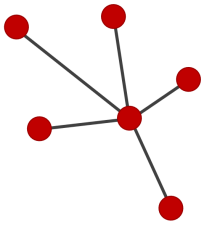
ABETS – A Beaver Exposure Tracing System



6.033 Design Project Report

Enrique Casillas, Itamar Chinn, Daniel Stein
enriquec@mit.edu, itamarc@mit.edu, djstein@mit.edu

WRAP Instructor - Atissa Banuazizi
Recitation Instructor - Henry Corrigan-Gibbs

Massachusetts Institute of Technology

May 11, 2021

Contents

---

1. Introduction

During the COVID-19 pandemic, fast and accurate contact tracing has become a critical task to limit disease spread within communities. Several existing systems already use smartphone technologies to automate this process, with each system contending with trade-offs between functionality, scalability, and privacy. Existing systems either collect a large amount of personal information at the expense of user privacy or prioritize privacy at the expense of functionality. We therefore introduce A Beaver Exposure Tracing System (ABETS) that meets the contact-tracing requirements of a university. ABETS integrates several components—a central server and a set of Wifi routers operated by the university, as well as a large number of individual smartphones—to support key functions. First, the system *identifies contact events*, where individuals are close together for an extended period of time, using Bluetooth Low Energy (BLE) communications between phones. Second, it *notifies infected individuals* of positive test results via the central server, providing support as necessary. Third, using known contact events, it *determines individuals who were exposed* to an infected person and notifies them so they can quarantine. Fourth, it provides information for *medical support and public health needs*.

ABETS supports these functions while prioritizing the design principles of *scalability*, *reliability*, and *user privacy*. Scalability means our system is able to handle a large number of users and situations in which a large number of positive cases arise simultaneously, whether these are spread throughout the community or concentrated within a cluster. In our design, demands on the server and on the network grow linearly in the number of users. Additionally, the demands on each individual smartphone are small and constant even as the total number of users increases. Reliability means that our system fulfills its functions in an accurate and timely manner. ABETS handles all positive cases appropriately with the necessary notifications being made automatically within an hour. Finally, ABETS supports all of these functions while protecting user privacy, with only the minimal amount of necessary information being collected and stored on the server. Importantly, the central server keeps all contact events anonymous and maintains them separately from the university's database of user information. In the following sections, we describe the implementations and interactions of our modules, showing how they achieve these key design goals of *scalability*, *reliability*, and *user privacy*.

In Section 2, we describe the system design and the components involved. In Section 3, we demonstrate how these components interact to support the required functions. In Section 4, we address several security considerations that prevent malicious behavior from affecting the accuracy of our system. Finally, in Sections 5 and 6, we describe several use cases and evaluate our design within the system constraints.

2. System Design

This section provides a detailed description of ABETS. Following an overview of the system design, we explain in detail the functions of the *central server*, *phones*, and *communication protocols*.

2.1. Overview

Figure 1 outlines the main system design. It shows the main modules and components, and provides a general idea of how information is directed between the different components.

The overall implementation of ABETS takes advantage of three major components. At the lowest level, ABETS uses mobile smartphones to identify and communicate contact events using Bluetooth Low Energy (BLE). At the second level, the system uses routers to serve as packet switches in the communication network. Routers enable direct communication between the phones and the central server through Wifi and through ethernet, and offer a scalable, distributed approach to communicating data. Some data from phones is also communicated anonymously using *onion routing* through multiple routers (see Section 2.4.2). These two methods constitute the communication module between phones and the central server. At the highest level, the system uses a central server that handles much of the storage and computation, including receiving data from university services and phones, processing and storing information, and sending data out to devices. The server houses an anonymized *contact graph database*, used to determine exposure events, as well as a separate *user information database*.

Importantly, ABETS maintains three unique user identifiers. Each user's phone generates its own *BLE ID*s on an hourly basis that are used in BLE broadcasts to build contact notifications. Phones also generate *secret ID*s daily that are used to identify nodes in the *contact graph database*. These two IDs constitute the anonymous IDs within our system. Lastly, the central server generates *identified ID*s that are directly tied to users within the *user information database*.

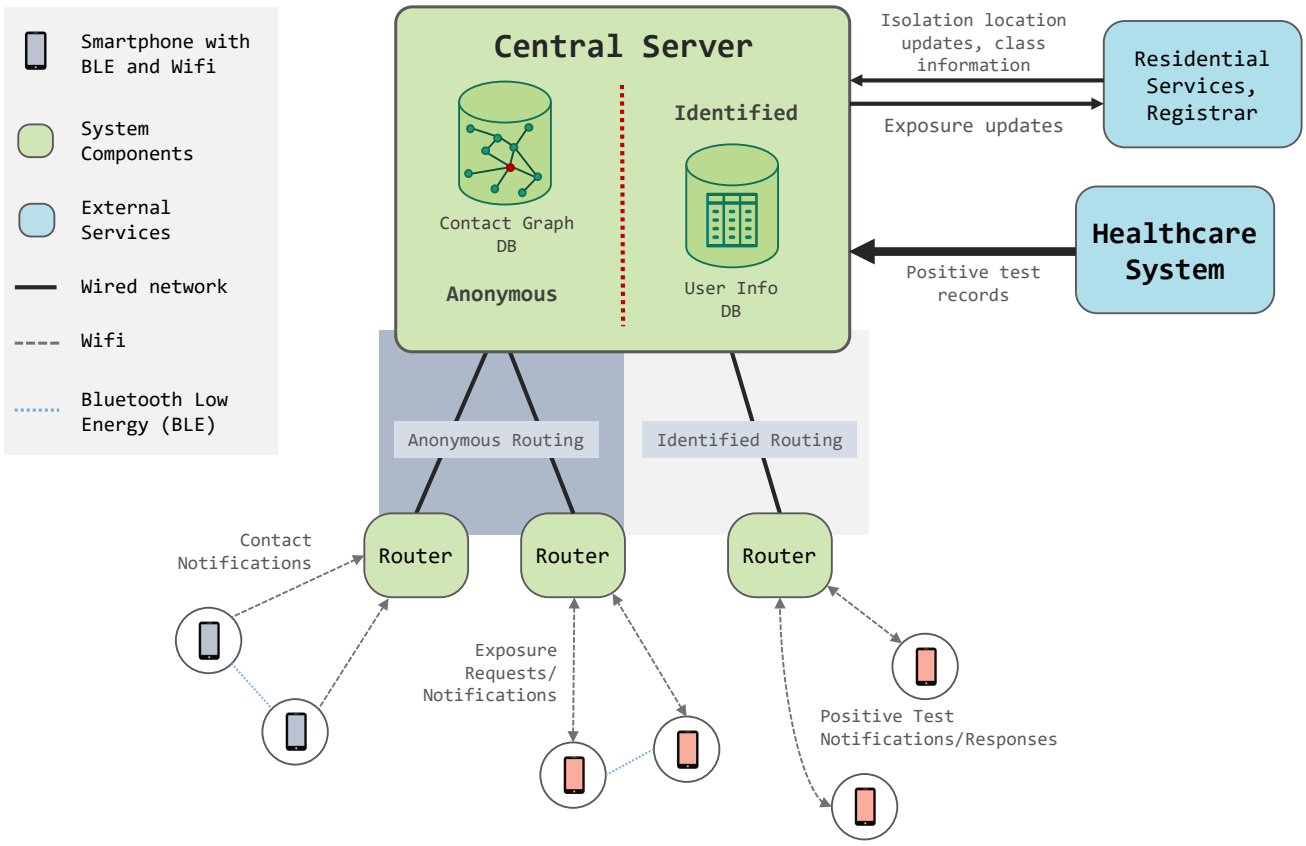


Figure 1: This System Diagram illustrates the three core components - the central server, the phones and the routers - as well as the three different communication protocols that the system uses - Bluetooth Low-Energy, anonymous onion routing and standard Ethernet and WiFi using TCP.

The following sections describe the different modules and components introduced here in greater detail. We also provide additional details on how the different IDs are used throughout the system.

2.2. Central Server

The central server fills two key roles: the *storage of user data*—including *user information* and *contact information*—and the identification and notification of infected or exposed individuals. Centralization of these tasks by the server allows for scalability as more users join the system, and reduces burdens on individual phones.

2.2.1. Data Structures

The central server consists of two data structures—a contact graph database and a user information relational database.

User information is stored in a relational database that keeps track of each user's name, student ID, phone number, living arrangements, associated classes, as well as tests and isolation status. This information can be used to quickly identify exposures based on shared classes or living spaces as this information is already known to the university. Positive tests are also known to the university, so there is minimal infringement of user privacy.

Contact events are stored on the central server as a graph database for scalability. If exposure events were determined by individual phones using their own contact logs, then each phone would need to know about all positive cases in the system, placing a large burden on each device and on the network as the size of the system

Function Name	Specification	Modules
`receive_contact (ble_id, secret_id)`	Receives a contact notification from a single phone. (*Not* a contact event)	Phone to Server *Anonymous*
`add_contact_event (contact_A, contact_B)`	Adds a contact event if the contact notifications (the *BLE IDs*) from phones A and B match.	Server *Anonymous*
`update_user_info (identified_id, data)`	Updates the identified user record. (e.g. positive test notification). Users may also update their information themselves.	Services to Server *Identified*
`get_secret_id (identified_id)`	Sends a notification to an infected user's phone and asks for its *secret IDs*.	Server to Phone *Identified*
`blind_sign (msg)`	"Signs" a message from a phone without knowing the message contents and sends it back. See Section 4.3 for usage details.	Server to Phone *Identified*
`create_exposure_graph (ble_id, secret_id)`	Creates an exposure graph in response to the user with `ble_id` being infected.	Phone to Server *Anonymous*
`exposure_request_response (secret_id)`	Responds to user phones asking if they have been exposed with an exposure status by checking all exposure graphs.	Server to Phone *Anonymous*

Table 1: Table of key central server's functions and how they relate to the other modules. Functions execute entirely on either the anonymous side or the identified side of the server to reinforce privacy. Note the three modules involved in central server functions—the external university services, phones, and the server itself.

grows. Instead, by maintaining the full community's contact event graph on the central server, the system can efficiently query a list of positive individuals to find exposures. However, a crucial property that we enforce is that **the server has no way of determining who the individuals are within the contact graph database on its own**. In other words, contact events stored on the server are completely anonymized. Each node has only the *secret IDs* and infection status of phones involved in contact events, which are not linked to user information in any way. Additionally, each edge has only an approximate timestamp associated with it to allow for queries of recent contact events and removal of old data. In this way, the university has no way of knowing the identities of individuals within the contact graph or who is interacting with whom.

Our choice to use an anonymous contact graph database requires additional considerations. First, the database must truly be anonymous—phones and individual identities must remain hidden even if the phones are sending messages to the central server. Section 2.4.2 describes how we addressed this challenge. Second, positive test records stored on the *user information database* and contact events stored in the *contact graph database* are both needed in order to determine potential exposure events. However, as mentioned, the server has no way of knowing the identity of the individuals in the contact graph. Therefore our system must be able to determine where and how to link this data together while still maintaining privacy. Sections 3.3 and 4.3 describe how we addressed this challenge.

2.2.2. Updates to Server

Table 1 provides a list of key functions provided by the server that allow it to update the internal databases and send out messages to other modules. Section 2.4 details how the updates are actually communicated between modules.

Updates to the server can occur in three ways: 1) through updates from the healthcare system, 2) through updates from smartphones, or 3) through internal updates such as regular removal of expired data. The healthcare system notifies the central server of positive tests to initiate exposure tracing and can also send general user updates

by sending them through the `update_user_info()` function. Smartphones communicate *contact notifications* to the central server, allowing the server to create *contact events* and update the contact graph database. Additionally, phones of infected individuals can notify the central server of their *secret IDs* and infection status to enable exposure tracing using the anonymized contact graph. Finally, edges are removed from the contact graph after 2 weeks, unless they involve an exposure event, in which case they are maintained for 180 days.

Lastly, ABETS can manage updates to the following attributes for identified user records in the user information database. Again, these updates can be executed using the `update_user_info()` function.

- User records keep track of living arrangements, so university services can send updates regarding whether or not users are in isolation, quarantine, or neither.
- Services can also update information on individuals being moved to a different isolation locations. Users may also indicate whether they transitioned from isolation to quarantine or directly quarantined after getting an infection notification.
- The healthcare system sends *all* positive test records to the central server, so user records contain information on testing, even during an infected user's isolation period.

2.3. Phones

In designing the phone module, the central consideration was to preserve privacy, without compromising scalability. This section will discuss the design choices in the phone module, focusing on its primary functions and data management.

2.3.1. Bluetooth Low Energy Communication

The phone recognizes potential contacts by listening for nearby phones' BLE broadcasts. All phones broadcast a 70-byte long message every 250 milliseconds, consisting of the phone's *BLE ID*, a timestamp, and signal strength. All phones store the information in Table 2 in the `list_of_broadcasts` table for each BLE record they detect.

The function `determine_contact` recognizes if the same *BLE ID* appears in `list_of_broadcasts` for 20 minutes or more at a high signal strength (under 3 meters) indicating a possible contact event. Eventually, both phones involved in a contact event must notify the central server for it to be added to the contact graph database (See Section 3.1).

Another function of the phone module is to query the central server *bihourly* as to whether or not the phone's unique *secret ID* was a part of an *exposure event*, in which a neighboring node in the contact graph was infected. This communication happens using an anonymizing *onion routing* communication protocol (Section 2.4.2) which allows communication of the *secret ID* to the central server without exposing the identity of the phone user.

The full list of functions available to each phone module is shown in Table 3.

Two key design decisions were made in the phone module. First, we initiate most communications from the phone, rather than the central server. This allows us to prioritize privacy by separating *secret IDs* from users' identities through anonymous communication. Second, we store most of the data and do most computations on the server rather than on the phones to preserve battery life and storage. The main weakness of our design choice is that the phone will only be able to notify the user of potential exposure when it queries the server (bihourly). This is an acceptable compromise for preserving our key design goals of *scalability*, *reliability*, and *user privacy*.

Field Name	Type	Description	Example
`ble_id`	ID	64-bit ID number, representing the person of contact.	da6af62d9-c574-4e9e-8 e62-cd549a722ee6
`timestamp`	datetime	The time when the contact occurred.	"2021-03-19 11:53"
`contact_strength`	Real	A metric of distance between two phones.	0.82133333

Table 2: Bluetooth Low Energy Log Data

Function Name	Specification	Modules
`broadcast_id (ble_id, send_time)`	Loops every 250 ms and broadcasts the 70 byte BLE message at the given time.	Phone to Phone
`receive_broadcast (ble_id, data)`	Scans for nearby broadcasts and determines signal strength.	Phone to Phone
`determine_contact (list_of_broadcasts)`	Checks `list_of_broadcasts` to determine if any contact events have occurred.	Phone
`send_contact_notification (secret_id_A, secret_id_B, contact_data)`	Sends the central server its *secret ID* and the *secret ID* of the user `B` it came in contact with, along contact data such as a general timestamp.	Phone to Server
`send_secret_data (secret_id, signature)`	Sends the central server its *secret ID* with a valid signature. (Section 4.3)	Phone to Server
`query_exposure (secret_id)`	Contacts the server bihourly to ask whether its user has been exposed.	Phone to Server

Table 3: A table of key phone module functions. Note that *secret ID*s are only sent to the central server, and *BLE ID*s are only sent to other phones. In addition, the described Phone to Server functions operate across *anonymous* routing (Section 2.4.2), though the phone can also respond to server queries on the *identified* network (Section 2.4.1).

2.4. Communication Protocols

To achieve the proper balance between functionality and privacy, ABETS uses two different protocols for communication. The first is a basic routing protocol where the server knows the MAC address of the phone it needs to send information to and sends messages directly to its destination. The second is for when the smartphones initiate contact with the central server in an anonymous manner, such as after a contact event. In order to help maintain the anonymity of the individuals using the system, messages are sent using *onion routing* to hide the communication source.

Section 3 describes in more detail when each protocol is used within the system.

2.4.1. Identified Communication

ABETS requires that *positive test notifications* reported from the medical services are sent to the appropriate people and *exposure notifications* are sent to those in shared classes or living spaces. To accomplish this, ABETS supports an identified communication protocol where the central server can directly communicate with certain phones based on the users' *identified ID*s. This ensures the system operates quickly and reliably.

The identified protocol is rather simple. Since the central server maintains a mapping of identified user IDs to their phone's MAC addresses, it just sends the aforementioned updates out to the routers with the phone's address as the target, and the routers forward the message to its destination. This is done under a standard TCP protocol.

Note that in order to communicate with the university healthcare system, registrar, and residential services, a *direct communication link* is established between said systems and the central server. If the healthcare system wants to send positive test records to the central server, for example, it can do so using the direct link. This choice supports the system's reliability, since messages can be delivered quickly and accurately between ABETS and any external services.

2.4.2. Anonymous Communication via Onion Routing

ABETS requires that the update of a *contact event* be untraceable to a specific phone to maintain the anonymity of each phone's *secret ID*, and so we use *onion routing* to anonymize the source of the data communication (in the same manner as Tor [3]).

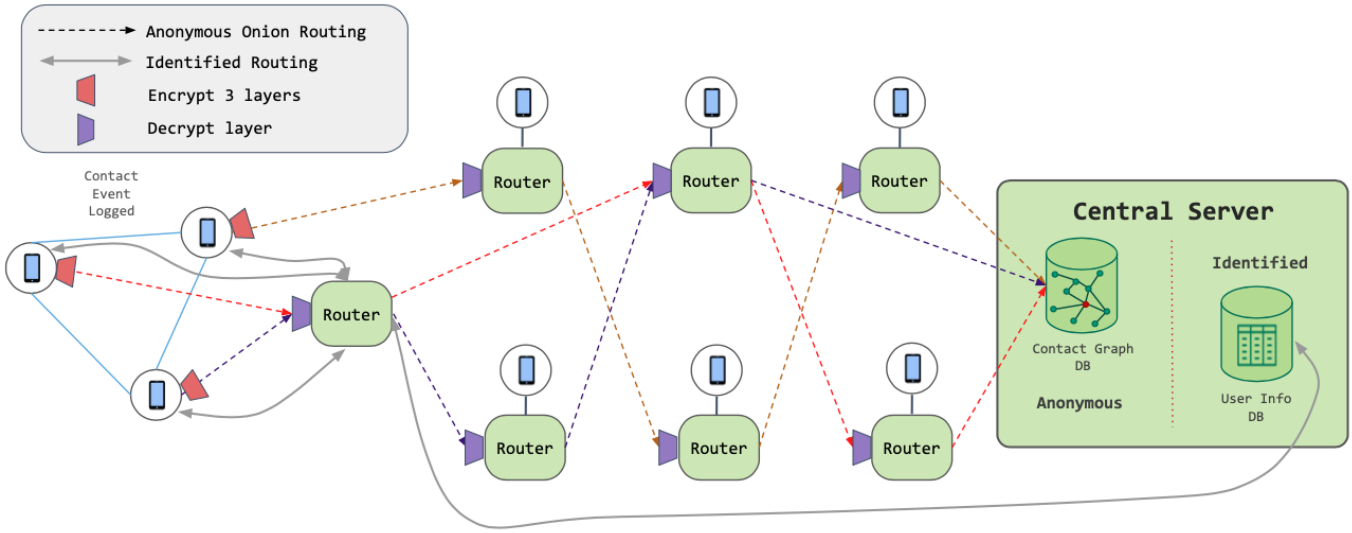


Figure 2: This figure illustrates how onion routing is used to hide the identity of the phone communicating with the central server. The message is encrypted three times over and each node (either the phone or router) knows only to decrypt the identify of the next node, thus by the time the message has reached the server, the server only knows the identity of the previous node.

Onion routing works by applying multiple layers of encryption (ABETS uses three layers), where each node in an overlay network decrypts one layer at a time, and identifies only the next node to forward the encrypted message to. The identity of the updater is obscured by doing the initial three-layer encryption with three randomly selected nodes to forward the message. The last node has a fully decrypted message but does not know the origin of the message, and forwards it to the central server. This is illustrated in Figure 2.

When the central server module receives a new *contact event*, it responds by sending the phone module an acknowledgement—in a similar process, the response follows the route back by hopping from node to node, where each node only knows the next node. The phone then deletes expired contact data from the phone, as this is now stored on the central server.

In ABETS we leave the decision of how to implement onion routing to the system implementer and offer three potential implementations (which are by no means exhaustive):

1. Each phone that participates in the system can act as a relay node, this means that each hop (out of the three hops) is to another phone. This solution is attractive as the software can be built in to the ABETS software on the phone module. One potential downside however, is that this solution is likely to affect battery life and performance of the phone.

2. The routers can act as relay nodes. Using the routers as relay nodes is ideal from a performance and simplicity point of view, however, it may require building custom software to run on them, and updating all the routers on a university network is no simple task.

3. Another option is to use the existing Tor network. Tor is a worldwide onion-routed network and an open-source project. To use the Tor network, minimal if any changes to existing modules (routers or phones) would need to be made and so it is an attractive option, yet it does mean that personal health data (although heavily encrypted) may travel all over the world before reaching the destination on the university campus. This will have certain performance issues and potential data security issues as well.

We place further analysis of onion routing implementations on a university network out of the scope of this paper.

3. Integration of Components

Now that the key protocols and modules have been established, we present the sequence of interactions involved in the four key functions of ABETS. The first function is identifying *contact events* between users and communicating those events with the server, which is the responsibility of the phone module. This can be seen in Figure 3. The other functions of notifying those *infected*, determining and notifying individuals who were *exposed* to an infected person, and providing information for *medical support and public health needs* are all largely the responsibility of the central server module. The general control flow of the system in these use cases is shown in Figure 4. This section will discuss and illustrate these different functions of ABETS, referencing the central server and phone functions found in Tables 1 and 3.

3.1. Identifying Contact Events

In the event that two or more users come into close proximity (under 3m) for 20 minutes or more, a contact event is triggered on the phone (See Section 2.3.1). This initiates a series of events that culminates in the contact event being added to the contact graph in the central server *with no identifying information*. The events are illustrated in Figure 3 as follows:

1. Users constantly receive and record each others' BLE broadcasts using the `broadcast_id()` and `receive_broadcast()` functions, from which the phone module can recognize a contact event using `determine_contact()`.

2. Each phone sends a contact notification to the central server through the anonymous onion routing protocol (`send_contact_notification()`). This contact notification consists of the *BLE IDs* of both phones involved, as well as the phone's own self-generated *secret ID*, which is only shared with the server.

3. If the server receives two contact notifications involving the same pair of *BLE IDs*, it adds the contact event as an edge in the contact graph. This is done using the `receive_contact()` and `add_contact_event()` functions. The nodes are labeled by the *secret ID* of each phone, and edges are marked with an approximate timestamp (see Section 4.2 for the security implications of using a *secret ID*).

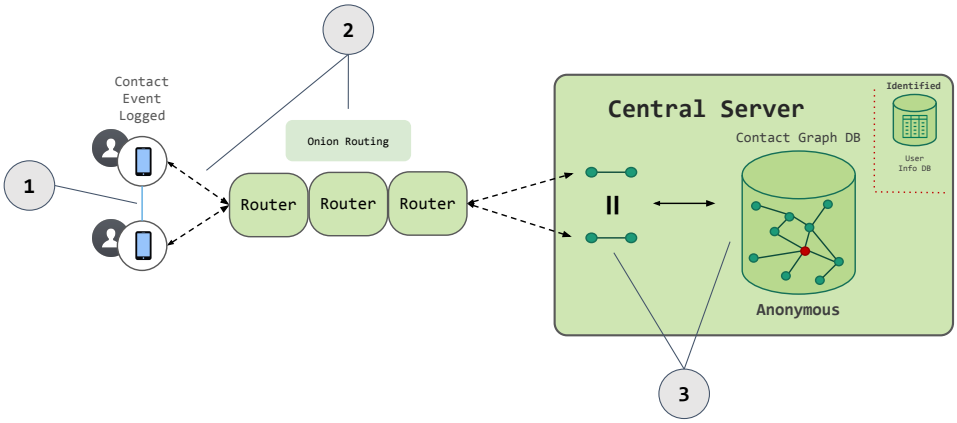


Figure 3: This figure illustrates the series of events when a user is involved in a contact event (Section 3.1): 1) the phones first identify the event, which they 2) communicate anonymously to the server using onion routing, after which 3) the server verifies the event and updates the contact graph database.

3.2. Notifying Infected Individuals

Once an individual is infected and tests positive, the system must first notify that individual to begin quarantine and to initiate the exposure tracing process. The events are illustrated in Figure 4 as follows:

1. A user tests positive at a testing site.

2. The medical system gives the central server the name of the infected user through the direct link, and the server calls `update_user_info()`. The central server checks for the individual in the user information database and marks that user's status as infected/quarantining.

3. The server communicates over an identified communication channel to notify the infected user by calling `get_secret_id()`; the user acknowledges this in an exchange that ultimately allows the server to mark the user's node in the contact graph as infected without violating anonymity (in step 4; see Section 4.3 for details of this process). The user then enters quarantine status.

   (a) The server also notifies all members of the same class or living community through the identified routing method (as in step 3). They then enter isolation status.

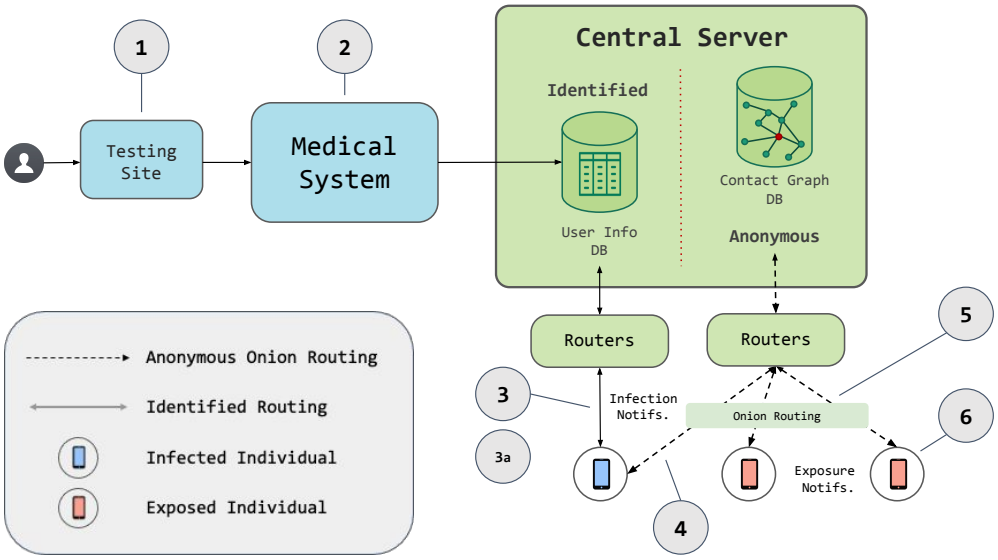


Figure 4: This figure illustrates the series of events when a user is infected and other exposed users must be to determined and notified (Sections 3.2 and 3.3): (1-3) The system first notifies the infected user of a positive test. (4) The infected user anonymously sends the central server its *secret IDs* from the past two weeks; for validation, this message received a blind signature from the server in step 3. (5-6) Other users anonymously check in with the central server twice each hour, providing their *secret IDs*. The server checks these and notifies of any exposures.

3.3. Determining and Notifying Exposed Individuals

After a user acknowledges their infection status, ABETS can determine who was in contact with them and notify those exposed that they must enter isolation. The events are illustrated in Figure 4 as follows:

4. The infected user sends the central server its *secret IDs* over the past two weeks; this is done anonymously using onion routing and a blind signature scheme described in Section 4.3. Using this information, the server calls `create_exposure_graph()` to generate an exposure graph of users that came into contact with the infected user by querying the contact graph.

5. Users query the server bihourly with their past two weeks of *secret IDs* via the anonymous onion routing protocol (`query_exposure()`). The server responds with whether or not they were exposed after the it consults with the exposure graphs.

6. If exposed, the user acknowledges and enter isolation status by passing updates through the server's function `update_user_info` across the identified network.

3.4. Supporting Research and Public Health

ABETS is also able to support public health and research needs, while maintaining user privacy. Because the contact graph database is completely anonymous it can easily be shared with researchers to study virus transmission within the community — such as the duration and closeness of contact necessary for transmission, or whether cases originate from different sources or from a single super-spreading event. We provide a function for the healthcare system and researchers to extract the contact graph over the prior 2 weeks, in addition to information about which nodes tested positive. The *secret IDs* used to identify each node are reassigned when the graph is exported to further anonymize user information.

4. Security

4.1. Threat Model

By prioritizing privacy and maintaining the anonymity of users and communications within the system, we risk exposing ourselves to malicious activity that could compromise the accuracy of events and notifications. In particular, we must guard the system from malicious users both internal and external to the university, who might wish to falsely send others into isolation or quarantine. This might happen, for example, if the malicious user falsely tells the server that Alice and Bob had been in contact, and then says Bob had been infected, sending Alice into isolation. To avoid this, we introduce security measures that make it infeasible for a malicious user to create either 1) fraudulent contact events or 2) fraudulent infection notifications. Furthermore, we require that all communication is fully encrypted end-to-end.

4.2. Preventing Fraudulent Contact Events

In theory, a malicious user could send a false contact event log to the central server, since onion routing hides the sender's identity. To circumvent this, we propose a security mechanism that confirms the authenticity of contact events at the time that an infection is submitted. Our solution prevents any fake events from leading to actual exposure notifications.

Each phone has a *secret ID* that represents it on the contact graph in the central server. Each phone generates this *secret ID* for itself (a 64-bit random int, with low probability of collision), which it shares *only with the server, never over BLE*. Each time a contact event occurs, all phones involved send a log to the server that includes: their *BLE ID*, the other phone's *BLE ID* and their *secret ID*. The server only adds nodes and edges to the contact graph if it receives such a log from all parties, creating an edge between two nodes labeled with the phones' *secret IDs*.

At this point, the server still doesn't know if any of these edges are true events or maliciously submitted events. However, when the server discovers that a user has been infected, only that user's real *secret IDs* will be marked as infected (steps 3 and 4 in Figure 4). Additionally, when other users ask the server about exposures, only their real secret IDs will be queried (steps 5 and 6 in Figure 4). As a result, the server will identify exposures only from true contact events involving real *server IDs*, which are highly unlikely to be forged.

For example, let's consider two putative events, shown in Figure 5: one where Alice was in contact with Bob, and another where Alice was in contact with Charlie. The first event is real, while the second was submitted maliciously, perhaps by someone who was snooping Alice and Charlie's *BLE IDs*. As a result, the first event contains Alice and Bob's real *secret IDs* (say, `A.secret_id = 1` and `B.secret_id = 2`). However, an attacker does not know Alice or Charlie's real *secret IDs*, so they must make them up (e.g., by setting `A.secret_id = 7` and `B.secret_id = 3`). Now suppose we find that Alice was infected, so the central server sends Alice a message, and Alice responds, saying "I am infected and my *secret ID* is 2". In this case, as desired, Bob will be notified of exposure while Charlie never will. Since a malicious user will never know another user's *secret ID*, the central server knows that only edges with the correct *secret IDs* are true events.

For added security and to reduce the risk of privacy loss, we require that phones change their *BLE ID*s once an hour and *secret IDs* once per day at randomly selected hours. The above process and all other described processes work exactly as described with a single ID (they were explained this way for simplicity), but all ID numbers used over the last two weeks are stored locally on the phone and communicated with the server, just as in the case of one

non-changing ID. Furthermore, each time a phone receives a BLE message from another phone (during a contact event) it delays reassigning itself a new *BLE ID* so that it does not change mid-contact event.

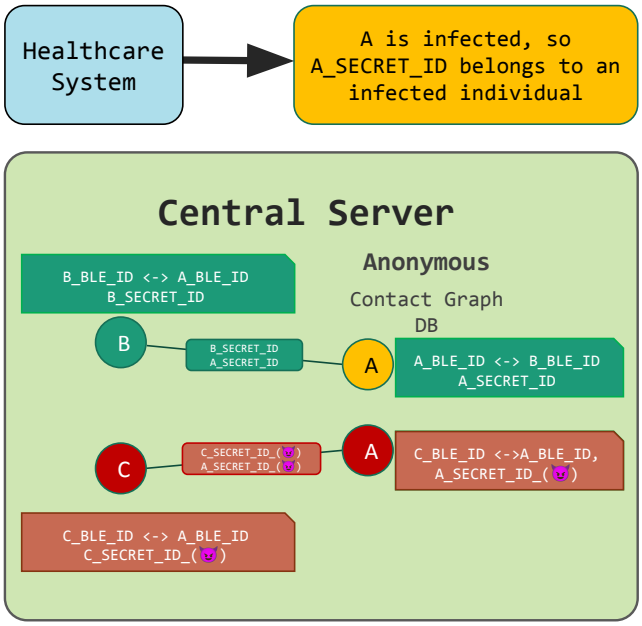


Figure 5: An illustration of two contact events on the central server: one a real contact between Alice and Bob (green), and another a fake event between Alice and Charlie (red), submitted by a malicious user. When Alice becomes infected, the central server receives Alice's real *secret ID* and ignores the fake contact event between Alice and Charlie.

4.3. Preventing Fraudulent Infection Notifications

Another situation where malicious activity could occur is when a user pretends to be infected in order to send their contacts into isolation. Note that the strategy in Section 4.2 depends on the authenticity of infected users, since they send their *secret IDs* to verify contacts.

To prevent fake messages of this kind, we use a blind signature scheme that ensures reported infections originate from *real* infections discovered by the medical system. When the server initially communicates to the user's phone that the user is infected (step 3 in Figure 4), that phone generates the message it will send anonymously to the server, blinds it, and asks the server to sign it. The phone can then unblind the message before sending it anonymously to the server (in step 4) [1]. With this strategy, only phones of those who have actually been notified by the server of infection can produce valid signed messages announcing that certain anonymous and secret IDs correspond to an infected individual (this is encapsulated in the phone's `send_secret_data()` function which requires the server's signature). Thus, the server can validate which *secret IDs* correspond to real infections without ever learning the user identities of those IDs.

This approach maintains the greatest level of privacy while ensuring security, because all the information in the contact graph remains anonymous, even in the case of infections. While it may still be possible for the university to match the identities of infected individuals to their *secret IDs* by mounting a timing attack, we consider this to be outside the scope of our threat model. Furthermore, such an attack would yield little information, as the university already knows who is infected, uninfected individuals remain completely anonymous, and ordinary contact tracing would also have uncovered those with whom an infected individual interacts.

4.4. Protecting Sensitive Information

Given that our system stores the full contact graph on the central server, it would be extremely problematic if an attacker or user administrator could access the server and deduce sensitive information about each student and

who they interact with. For this reason, it is important to reiterate that all of the IDs stored within the contact graph remain anonymous through every function provided by the server. By design, the mapping from user identity to the *secret IDs* is not known or stored anywhere within the system except on each user's own device. Additionally, the *BLE IDs* that are announced to other phones are updated hourly, and the *secret IDs* daily, to further obscure the mapping between IDs stored in the contact graph and individual users.

5. Use Cases

5.1. Example Scenarios

Three scenarios demonstrate what could occur during the system's operational cycle, and how the system may respond.

The first scenario is *very low numbers* in which there is a low infection rate and there are few new daily cases. In this scenario, all 20,000 phones send periodic requests, but most are not infected. Of course, the system's goals would be fulfilled, though most phones would be sending what are essentially useless requests, wasting energy within the system.

The second scenario is *very high numbers* with a high infection rate. ABET operates well in this scenario, since positive tests are processed by the central server continuously. In addition, because of the randomized, periodic requests sent by phones, the expected number of packets passing through the network at any time remains relatively low.

Lastly, in a *compressed high numbers* scenario, positive test results are delivered in "batches" at the same time each day. This event presents the greatest challenge to the system, as it requires a large amount of processing power, and there could be high network activity in short periods of time as infected individuals are notified. Fortunately, because of the bihourly requests initiated by the phones (rather than the server), the strain on the network is mitigated.

5.2. Phone Crashes

Individual phones within the system can crash at any time and take up to 10 minutes to fully recover. Such failures have a minimal impact on the reliability and performance of our system. There are three situations where we might expect a phone crash to disrupt our system:

1. A phone crashes for an infected individual during or before an infection notification.
2. A phone crashes for an exposed individual who has not yet been notified of exposure.
3. A phone crashes in the middle of a contact event.

In the first case, it is possible that a phone misses the initial notification from the server that the individual is infected. However, this is easily addressed if, upon restarting after a crash, the phone simply asks the server whether it is infected and missed the notification.

In the second case of an exposed individual who had not yet been notified, the phone should just proceed normally after restarting, since all exposure notifications are initiated by the phone anyways. If more than 30 minutes has elapsed since the phone last checked in with the central server, it should contact the server to ask whether it has been exposed. Otherwise, it can wait until it's time to check in with the server again.

In the third case where a phone crashes in the middle of a contact event, the risk is that a contact event may not get recorded accurately. This can also be an issue if BLE is unreliable and messages are dropped. This case is also easy to deal with if we are conservative and count anything as a contact event that would qualify if 10 (or fewer) minutes of missing data were filled in. After restarting the phone and determining any contact events, contact events get reported to the server as usual.

6. Evaluation

This section discusses the evaluation of ABETS as an exposure tracing system. We go over quantitative evaluation metrics such as storage and network capacities as well as qualitative metrics including feasibility and flexibility.

6.1. Storage Capacity Estimation

ABETS is able to maintain all of the data needed for accurate contact tracing within phones and in the central server. Recall that most data are stored in the central server, which as per the specification has 12TB of storage. Phones contain a small log of BLE signals, so the 1GB of storage available is more than enough to store the necessary information to detect contact events. Table 4 contains information about sizes of certain records and messages used throughout the system. These values are used in evaluating capacity.

Record/Message	Size (Bytes)
BLE broadcast	70
Contact Notification	100
Exposure Request/Notif.	150
Positive Test Notification	50
Contact Event	200
User record	100,000 (100KB)

Table 4: Records and messages used in throughout the system and their sizes.

6.1.1. Phone Storage and Battery Life

Recall from Section 2.3 that phones emit BLE signals every 250 milliseconds in the form of 70-byte long messages. Let $T = 250$ ms be our period of BLE broadcasts, and $x = 70$ bytes be the size of each broadcast. Given 1GB of storage, each phone can theoretically store $n = 10^9/x \approx 1.4 \cdot 10^7$ records across a time frame of $T \cdot n \approx 1000$ hours or 41 days. Taking some storage from the phone to keep track of user's information such as their past *secret IDs* and *BLE IDs* changes the calculations only slightly. Since old records are deleted after a certain period of time (usually two weeks for actual contact events, and more frequently for all the individual BLE records), there is more than enough storage in the phone to maintain the BLE broadcast logs over the course of the system's lifespan.

We can also be confident that ABETS also has a minimal impact on a phone's battery life because of our choice to keep most of the processing and storage on the central server. Unless implementing onion routing through phones (Section 2.4.2), phones only need to perform simple appends and queries to logs and make fairly simple calculations.

6.1.2. Central Server Databases

The central server's job is to manage the contact data and user data across a long period of time. As per the specification, ABETS stores contact logging information for 2 weeks and exposure information for 180 days. Given the 70-byte BLE message which can be stripped from unnecessary information, we set our *contact notification* to be 100 bytes long to store the information outlined in Section 2.3.1. Recall that contact events, created by the `add_contact_event()` function, use information from 2 contact notifications (see Table 1). We then let a contact event be $2 \cdot 100 = 200$ bytes.

Given these calculations, we first consider the contact graph of $N = 20,000$ individuals. There are N nodes maintaining an ID and other information, with $\binom{N}{2} = O\left(N^2\right)$ contacts (edges) *in the worst case*. However, considering a large enough social setting, it is highly unlikely that every person will come into contact with *every other* person. In a regular setting without infection preventions, people may come into contact with about 25 people per day on average [2]. Although ABETS operates under the assumption that infectionary precautions are taking place, we make a conservative assumption that people come into contact with about 50 people per day on average because of the nature of housing and in-person classes. Therefore, we argue *sparsity* of the contact graph, meaning that it grows linearly with the number of participating people, $\Theta(50 \cdot N)$.

With these assumptions in mind, using our 200-byte contact events stored in the contact graph database, there are on average $50 \cdot N \cdot 200 = 2 \cdot 10^8$ bytes of storage needed to represent a contact graph of a single day. Since we are required to keep at least 14 days worth of contacts, our total capacity requirements for the contact graph database is $14 \cdot 2 \cdot 10^8 = 2.8 \cdot 10^9$. We can also consider the size of exposure graphs. Note that the aggregate size of the exposure graphs can be no greater than the size of a complete contact graph, meaning that everyone eventually

became infected. Therefore, we only need $N^2 \cdot 200 = 8 \cdot 10^{10}$ bytes of storage if we maintain contact events as edges (Table 4).

The user information database is much smaller, since it only needs to maintain a persistent table of users and can be dynamically updated. We give 100 KB for each user record (and metadata) stored in the user information database (see Section 2.2 for details). For the N total users, we therefore need $N \cdot 10^5 = 2 \cdot 10^9$ bytes of storage for the user information database.

If we add the two database capacity requirements together, we get that the central server needs to store $2.8 \cdot 10^9 + 8 \cdot 10^{10} + 2 \cdot 10^9 \approx 8 \cdot 10^{10}$. Therefore 12TB ($1.2 \cdot 10^{13}$ bytes) of storage is sufficient storage space needed to maintain an expected contact graph (including worst case exposure graphs), and a user database.

6.2. Network Capacity Estimation

This subsection discusses capacity of the network as messages are sent across the network. According to the specification, the central server has two 10Gb ethernet ports, and routers each have 2.5Gb ports. Looking at the routers since they provide the least capacity, we apply dimensional analysis to figure out how many messages can be sent through the network. Letting one message be the sum of the sizes of all possible messages being sent across routers (see Table 4), we calculate

$$\frac{2.5\text{Gb}}{s} \cdot \frac{1.25 \cdot 10^8\text{B}}{1\text{Gb}} \cdot \frac{1\text{ Msg}}{100\text{B} + 150\text{B} + 50\text{B}} \approx 10^6\text{ Msg/s}$$

Across $N = 20,000$ users, we then get that a single router can handle around 50 messages being sent by each user every second. Furthermore, phones only send exposure requests bihourly, a key design choice made to mitigate network congestion. Therefore, assuming no network failures, this capacity is more than enough to handle day-to-day operations being performed in our system.

6.3. Additional Evaluations

Throughout this report, we have demonstrated how our design produces a system that is *scalable* and *reliable* with strong support for *user privacy*. Besides these properties, two additional metrics to consider are *feasibility*—how practical the system is to implement—and *flexibility*—how easy it is to add users, change components, or adapt the system to a new setting.

In general, our system design is *feasible* to implement. Standard hardware can be used for servers, smartphones, and networking, along with standard protocols for establishing secure client-server connections and ensuring atomicity and isolation for client-server transactions. One area of concern is how to implement onion routing within the university network. To avoid using custom routers, the simplest solution would be to establish a Tor-like network among the phones within the system. The software to communicate anonymously over this network would be included as part of the application that users install. Other options were discussed in Section 2.4.2.

In addition, our system is *flexible* in several ways. First, adding and removing users is simple. Once users have registered with the university to associate their phone with their identified user information, their phone can immediately start to send and receive BLE contact notifications, and communicate contact events anonymously with the central server. There is no need for any user registration or set-up on the anonymous portion of the central server; rather, notifications are simply handled as they come. Second, changing components of the system—e.g., by adding or removing routers, adding or replacing servers, or replicating or distributing data—can be done without significant impacts on the functionality of the system. Finally, our system can adapt to new settings where the number of users and frequency of contacts differ and support different policies regarding what counts as a contact event and what the consequences of exposure are. Overall, our system design provides a general method for storing, updating, and querying a full contact graph on the server while maintaining complete anonymity of user identities.

Finally, it is worth noting that our system requires users to have a phone in order to determine contact events and notify them of exposures. It would not be possible to add users without a phone into the contact graph on the database because all of the information stored there is anonymous and requires BLE-generated events. Students without a phone would either have to be provided with an embedded network-enabled device that could serve the role of a phone, or resort to self-reported exposure tracing using identified data on the server.

7. Conclusion

Our proposed system accomplishes the goals outlined in the specifications: the system determines *contact events* and *exposure events*, notifying users of the need for isolation or quarantine; supports public health needs; and prioritizes user privacy. With most processing done on the central server, our system supports *scalability* as the number of users or infections increases. All functions are done automatically without need for user input, improving the *reliability* of the system. Finally, *privacy* is achieved through anonymity through the anonymous *secret IDs* used in the server's contact graphs, the hourly changes in *BLE ID*s, and the communication protocols used to relay information between phones and servers.

8. Author Contributions

This project was a true test of international collaboration, and as such, it was of great importance to us to collaborate on all matters of the project. Each author contributed equally to all sections, where **Enrique Casillas** in particular designed the diagrams and tables, **Daniel Stein** initially suggested having different IDs for anonymous and identified data, and **Itamar Chinn** proposed using onion routing for anonymous communications.

9. Acknowledgements

A special thanks to Henry Corrigan-Gibbs for technical feedback, particularly around security issues, and to Atissa Banuazizi for feedback on communication. We would also like to thank Katrina LaCurts and the rest of the course staff for an instructive and engaging semester.

References

[1] David Chaum. "Blind signatures for untraceable payments". In: *Advances in cryptology*. Springer. 1983, pp. 199–203.

[2] Sara Y Del Valle et al. "Mixing patterns between age groups in social networks". In: *Social Networks* 29.4 (2007), pp. 539–554.

[3] Roger Dingledine, Nick Mathewson, and Paul Syverson. *Tor: The second-generation onion router*. Tech. rep. Naval Research Lab Washington DC, 2004.