Verena: End-to-End Integrity Protection for Web Applications

Nikolaos Karapanos[1], Alexandros Filios[1], Raluca Ada Popa[1,2], and Srdjan Capkun[1]

[1]Dept. of Computer Science
ETH Zurich

[2]Dept. of Electrical Engineering and Computer Science
UC Berkeley

***Abstract*—Web applications rely on web servers to protect the integrity of sensitive information. However, an attacker gaining access to web servers can tamper with the data and query computation results, and thus serve corrupted web pages to the user. Violating the integrity of the web page can have serious consequences, affecting application functionality and decision-making processes. Worse yet, data integrity violation may affect physical safety, as in the case of medical web applications which enable physicians to assign treatment to patients based on diagnostic information stored at the web server.**

This paper presents Verena, a web application platform that provides end-to-end integrity guarantees against attackers that have full access to the web and database servers. In Verena, a client's browser can verify the integrity of a web page by verifying the results of queries on data stored at the server. Verena provides strong integrity properties such as freshness, completeness, and correctness for a common set of database queries, by relying on a small trusted computing base. In a setting where there can be many users with different write permissions, Verena allows a developer to specify an integrity policy for query results based on our notion of *trust contexts*, and then enforces this policy efficiently. We implemented and evaluated Verena on top of the Meteor framework. Our results show that Verena can support real applications with modest overhead.

I. Introduction

Web applications store a wide range of data including sensitive personal, medical and financial information, as well as system control and operational data. Users and companies rely on these servers to protect the integrity of their data and to answer queries correctly. Unfortunately, web application servers are compromised frequently [28], thus enabling an attacker to tamper with data or computation results displayed in a webpage, thus violating their integrity.

The integrity of webpage content is especially important in applications in which displayed data affects decision making. This is well exemplified by medical web platforms where patient diagnostic data is stored on web servers and remotely accessed by physicians. Modification of this data might result in miss-diagnosis, lead to incorrect treatment and even death. A recent study estimates that millions of people are miss-diagnosed every year in the US with a half of these cases potentially causing severe harm [44]. Another study estimates that miss-diagnoses causes 40,000 deaths annually [47]. Some of the main reasons for miss-diagnoses were related to failure by the patients to provide accurate medical history, and errors made by a physician in interpreting test results [43]. If web applications with patient and test result data are corrupted, treatment decisions will therefore be made based on incorrect data, likely resulting in substantial harm. In §IV-A, we discuss a concrete medical web application used to monitor patients with implanted cardiac devices, where a web server compromise can lead to serious patient harm.

A bit over dramatic

What about client side?

In addition to physical safety, webpage integrity is important for basic security properties such as confidentiality against active attackers, for example, by providing integrity protection to data structures defining access control.

An uncompromised web server protects end-to-end integrity in a few ways. Many web applications involve multiple users and therefore enforce *access control* policies (e.g., a particular patient's data may be manipulated only by his physician). Furthermore, the web server ensures that clients' data requests and queries are executed *correctly* on data that is *complete* and *up-to-date* (i.e., fresh). An attacker who compromises the web server could therefore violate some or all of these properties.

Design properties

In this work, we propose Verena, the first web framework that provides end-to-end integrity for web applications, enforcing the properties above. Using Verena, the application developer specifies an integrity policy and a user's browser checks that a webpage received from a web server satisfies this policy, even when the server is fully compromised by an attacker. Verena checks the integrity of code, data, and query computation results within a webpage by ensuring that these results are complete, correct, and up-to-date.

Verifying query results efficiently in the web setting is challenging. While much progress has been made in generic tools for verifiable computation [14, 38, 46], using these tools for database queries and web server execution remains far too slow. Instead, work on *authenticated data structures* (ADS) [20, 23, 30, 31, 33, 37, 49] provides better performance by targeting a more specific, yet still wide class of functionality. These tools enable efficient verification without downloading data on the client and re-executing the computation. However, such tools are far from providing a sufficient system for web applications; work on ADS assumes that a single client owns all the data and this client has persistent state to store some hashes. Web applications are inherently *multi-user* and *stateless*. Different users can change different portions of data and a query computation can span data modified by multiple users.

What is this, why is it slow?

The first challenge for Verena is determining an API for developers that captures the desired query integrity properties, such as correctness, completeness and freshness, at the same time with multi-user access control. To address this issue, within Verena's API, we introduce the notion of query *trust contexts* (TC) coupled with *integrity query prototypes* (IQPs). A trust context refers to the group of users who are allowed to affect some query result, e.g., by inserting, modifying or deleting data used in a query. An IQP is a declared query pattern associated with one or more trust contexts. Each query runs within a specified trust context; Verena prevents a malicious server or a user outside of the trust context from affecting the results of this query. Queries may also span a set of trust contexts not known a priori; a mechanism called *the completeness chain* ensures that the returned result is complete, i.e., *all* the results of *all* the relevant trust contexts were included. The integrity policy is hence associated with queries and not with the data – nevertheless, the policy implicitly carries over to data because data is accessed through queries.

What are these?

The second challenge is verifying query results in a multi-user setting. To address this challenge, Verena builds on ADS work [20, 23, 30, 31, 33, 37, 49], and maintains a forest of ADS trees, by automatically mapping trust contexts to ADSes. To ensure completeness on queries spanning multiple trust contexts, as specified by completeness chains, Verena logically nests trees within other trees. Currently, Verena implements an ADS that can verify range and equality queries as well as aggregations, such as sum, count, and average. By substituting the underlying ADS, Verena can be extended to support a wider range of queries.

The third challenge, also brought by the multi-user and web setting, is a known impossibility result: when there are no assumptions on the trustworthiness of the server and the connectivity of clients, one cannot prevent fork attacks [32, 34] and hence cannot guarantee freshness. To provide freshness, one must use some trust server-side. Verena manages to use a small trusted base - a hash server that runs less than 650 lines of code. The hash server may also be compromised, as long as it does not collude with the main server. The hash server stores a small amount of information (mostly hashes and version numbers), based on which Verena constructs freshness for the entire database in an efficient way. The hash server also addresses the problem of web clients being stateless and not always online.

Whats this?

Fair model? Fair assumptions?

We implemented Verena on top of the Meteor framework [36] and evaluated it on a remote patient monitoring application, as well as two other existing applications. Our evaluation results show that Verena incurs a modest overhead in terms of latency and throughput. Our measurements also demonstrate the simplicity of the hash server, compared to the main server; the hash server achieves significantly higher throughput than the main server.

II. Model

A. System Model

We consider a typical web application scenario, where clients access a web server through web browsers. The *clients* could be browsers, operated by human users, or any device capable of communicating with the web server over the network. The *main server*, sometimes simply referred to as *server*, is a typical web server consisting of a web application front-end and a database server.

Our setup further consists of the following parties: a *hash server*, an *identity provider* (IDP), and the *developer* who creates and maintains the web application code. The hash server and IDP can be colocated on the same machine. They each have a public-key pair and their public key is hardcoded in Verena applications. In §III we describe the role of the different parties in the Verena architecture.

Moreover, for describing Verena's API in §IV, we use a No-SQL API, which is typical in modern web applications. For consistency, we use a syntax and terminology similar to MongoDB [5], which we simplify for brevity. This is also compatible with the Meteor framework [36], which we use to implement our Verena prototype (described in §X). Nevertheless, Verena's API could be easily cast in a variety of other database syntaxes (both SQL and No-SQL).

The web application's database consists of collections (equivalent of tables in SQL), each having a set of documents (equivalents of rows in SQL), and each document has a set of fields (which are similar to columns in SQL). A developer can issue queries to this database from the web application: "insert" (to insert documents), "update" (to update documents), "remove" (to delete documents), "find" (to read document data) and "aggregate" (to compute sum, average and other aggregate functions). The find and aggregate operations can read data based on filters, also called selectors, on certain fields using range or equality. Queries are defined using a JavaScript-like syntax. For example, "patients.find({patientID:2})" fetches all documents from the collection "patients" whose "patientID" field equals 2.

B. Threat Model

Verena considers a strong attacker at the main server; the server can be corrupted arbitrarily. This means that an attacker can modify the data in the database and modify query or computation results returned by the server. There are numerous ways in which an attacker could modify query results. For example, a malicious server can return *partial* results to a range query, it can return old data items, it can compute aggregates incorrectly or on partial or old data. Worse, the server can create fake user accounts or collude with certain users.

Attacker

This strong threat model addresses powerful attackers in the following use cases: a web application server runs in a cloud and a malicious cloud employee attempts to manipulate unauthorized information. Alternatively, an attacker hacks into the web application server through vulnerability exploitation and even obtains root access to the web and database servers, so she can change the server's behavior.

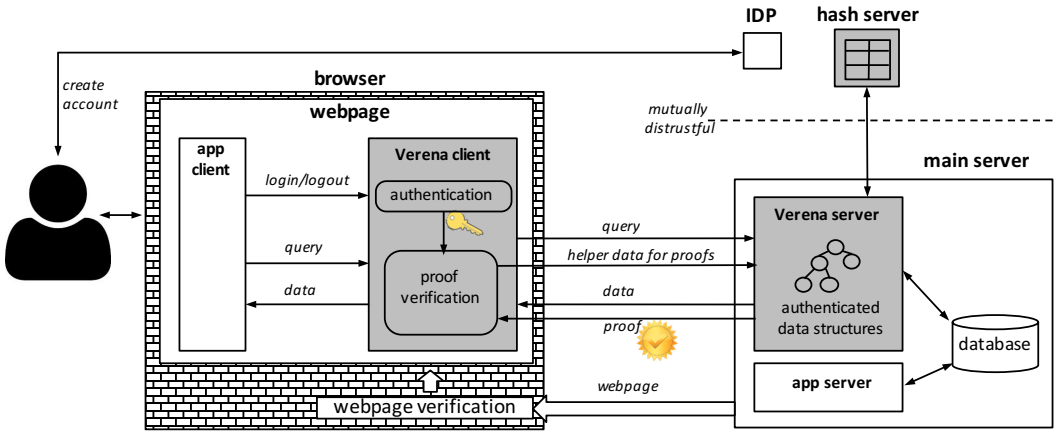

Fig. 1: System overview. Grey-shaded components are modules introduced by Verena on top of a basic client-side architecture setup.

An attacker can also corrupt the hash server, but importantly, we assume that an attacker can corrupt *at most one* of the main and hash server. In other words, we assume that at least one of the hash server and main server behave correctly. For example, these two servers could be hosted on different clouds such that the employee of one cloud does not have access to the second cloud. Alternatively, the hash server, which we show to be very lightweight compared to the main server, could be hosted in-house, while the resource-intensive main server could be outsourced to a cloud provider. We also stress that, given that the hash server runs a very small code base, and answers to a very narrow interface, it will be significantly less likely to be compromised by a remote attacker.

Network is usually the win in cloud. Is this valid?

The same threat model applies to the IDP server. We use the IDP for the task of certifying each pair of username and public key. Verena requires that only one of the IDP and main server to behave correctly. Hence, the IDP and the hash server can be colocated, as depicted in Fig. 1, where the mutually distrustful servers are separated by a dashed line.

Clients are also not fully trusted. They may attempt to bypass their write permissions by modifying data they are not allowed to change. They might even collude with either the main server or the hash server (but not with both of them at the same time). Nevertheless, clients are allowed to arbitrarily manipulate the data they legitimately have access to. If the main server colludes with a client, the server cannot affect the integrity of data owned by other clients that was not shared with the corrupted client.

Obv. The idea: attacker is a user too

Finally, we assume that the developer wrote the web application faithfully and followed Verena's API to specify an integrity policy. In contrast, the service provider and server operator are not trusted at all (and these fall in the main server trust model above).

III. Architecture

Fig. 1 illustrates Verena's architecture, which we describe throughout this section.

A. Basic Setup

To lay out the foundations for Verena's security mechanisms, Verena starts with the following base setup. First, Verena is a client-side web platform, a popular trend in recent years [15, 36]. These platforms provide advantages not only for functionality and ease of development, but also for security. ?

The dynamic webpage (with personalized content) is assembled on the client side from static code and data coming from the server. Previous work [41] has shown how to check in a client's browser whether the webpage code (such as HTML, JavaScript and CSS) has not been tampered with by an attacker at the server. Since the code is static, such a check essentially verifies the code against a signature from the developer. This check runs in a browser extension. Verena incorporates this mechanism and the browser extension as well. From now on, we consider that the webpage code passed this integrity check and we refer the reader to [41] for more details. Using mylar

Second, a standard requirement in multi-user systems providing cryptographic guarantees is an *identity provider* (IDP), i.e., an entity that certifies the public key of each user. For example, it can be similar to OpenID or Keybase [3], or could be hosted at the same place as the hash server. Without such an IDP, an attacker at the server may serve an incorrect public key to a user. For example, if user A wants to grant access to user B, user A requests the public key of user B from the server who replies with the attacker's public key such that the attacker obtains access. The IDP is involved minimally, when a user creates an account; at that point, it signs a pair of username and public key for each user creating an account. Although we do not discuss key revocation in this paper, enabling revocation in Verena would require the involvement of the IDP as well.

How easy is it to do this?

B. Verena Components

Now that we laid out the basic setup, we describe the mechanisms that Verena provides to prevent the server from corrupting data and query results. At a high level, the application developer, using the *Verena API*, specifies the integrity protection requirements (integrity policy) of the application. This allows Verena to derive the access rights of each user

for each data item or query. Based on this API, the server accompanies any integrity-protected query operation with a proof that it follows this policy and the client can verify this result. Also, whenever the client sends a query to the server, the client accompanies the query with helper data for constructing the proofs, if needed. Why is this necessary? Is this realistic?

More concretely, as shown in Fig. 1, on the client side, a webpage consists of two parts: the application's code written by the developer on top of the Verena framework, and the *Verena client*. When a user logs in, the Verena client performs authentication to derive the user's key from his password. If passwords are deemed unsafe, one can use other available secret derivation mechanisms [16].

In typical client-side web frameworks, the app client issues database queries; these queries are sent to the web server, which sends them to the database. In Verena, all such queries pass through the Verena client. Verena then determines if it is a query that must be integrity protected. If so, the Verena client provides helper data (such as challenges) to the server to be used in proofs. When the server returns the results, the server also provides a proof of correctness for these results that the Verena client checks before returning to the app client.

Many client side frameworks push data i the pub sub model to...

The main server consists of the *Verena server* and the regular app server. The app server, also written by the developer, performs operations that are not integrity-sensitive and do not require verification. All server-side operations that require verification pass through the Verena server.

How do you (dev) determine what is integrity sensitive?

The Verena server carries the difficult task of constructing proofs of correctness for query results that are efficiently verifiable by the Verena client. Verena builds upon work on authenticated data structures (ADS) and in particular tree-based ADSes [20, 23, 30, 31, 33, 37, 49]. ADSes enable efficient verification without downloading data on the client and re-executing the computation. We provide a brief background on the ADS used in our implementation in appendix A. Verena enables these ADSes to be used in a multi-user and stateless setting. In this manner, Verena can verify a wide range of common queries, but not any general query. Table I lists the read queries that are currently supported by Verena. Moreover, §IX discusses how Verena can be extended to support a broader range of query types.

What is an ads?

Since applications have different access policies, Verena needs to translate these policies into ADSes. Our new API, based on the notion of query *trust contexts* (TC), *integrity query prototypes* (IQP) and *completeness chains*, presented in §IV, captures an application's policy.

Moreover, due to the multi-user setting in Verena, different users are allowed to modify different portions of the data stored at the server. Thus, the Verena server has to maintain different ADSes for chunks of data that are modifiable by different sets of users. Respectively, Verena clients must ensure that, for each integrity-protected database operation, the server presents proofs for all relevant ADSes that data was modified only by legitimate users. To address this, Verena maintains a forest of ADS trees, by automatically mapping the developer's Verena API calls into the appropriate ADSes.

Since multiple users may be able to change the same data item, users do not know what was the hash of the last change. Additionally, in web applications, not all users are online at the same time and cannot notify each other of their changes. To make the problem worse, since the web setting is stateless, whenever a user logs off, any state he stored in his browser is typically lost. Moreover, the user should be allowed to login from a new browser where there is no state. This means that, even though ADSes help ensure integrity of some snapshot of the data, the server can still provide stale data. In fact, Mazières and Sasha [34] prove that, without any trust at the server or connectivity assumptions of users, one cannot guarantee data freshness.

To address this problem, Verena uses a *hash server*; a simple server whose main task is to serve the hash, version and last modifier for a given entry. As long as the hash server does not collude with the main server, Verena's integrity guarantees hold. To check the correctness and freshness of query results, a tempting approach is to store the entire ADS trees at the hash server. We show in §VI that we can avoid this approach, and maintain the task of the hash server simple, namely the hash server stores one entry per tree, corresponding to the root of the tree. As a result, the hash server is easier to secure; it runs a small code base, answers to a narrow interface, and is lightly utilized. The hash server could be collocated with the IDP server because Verena assumes the same trust model for these two servers.

IV. Integrity Policy API

In this section, we describe the main concepts behind Verena's API for expressing an integrity policy. In Verena we are concerned only with write access control. As discussed in §IX, systems like Mylar [41] can be used for expressing and enforcing cryptographically read access control.

In order to illustrate Verena's concepts and API, we use consistently the following running example of a medical web application.

A. Running Example: Remote Monitoring Medical Application

Our running example is a remote patient monitoring system used to connect cardiac device patients with their physicians. Such systems are deployed by a number of medical implant manufacturers such as [1, 2, 4, 17]. In order to evaluate Verena on such an application, we contacted one of the implant manufacturing companies and obtained access to the web interface of their remote monitoring system. This provided us with a better understanding of the type of web pages that these systems expose, access control that they implement and the type of data that they expose to the physicians. We then discussed with a cardiologist to gain a better understanding of the integrity policy of this application.

Modern implantable cardiac devices, such as cardioverter defibrillators (ICDs), cardiac monitors and pacemakers, monitor the patient's cardiac activity and take certain actions. In particular, implants measure data such as therapy delivered, heart rate, EKG (Electrocardiogram) data, and status of implant and leads.

To facilitate access to this data, implants communicate remotely with their clinics; this is supported by wireless telemetry devices which, when in the proximity of the patient, query the implant and then communicate the data further to the clinic server.

The server then exposes a web interface to physicians, through which they can access patient profiles, status of implants and measurements. Besides viewing this data, physicians ask the web server for certain *aggregate computation* such as average heart rate, number of heart beats per day (e.g., observed over a three-day period) and number of sinus pauses (i.e., skipped heart beats/asystoles). Physicians can change a patient's therapy by reprogramming the implant in the clinic, using short-range inductive coil telemetry.

The information a physician receives from the web server influences the decisions the physician makes for a patient and is thus integrity critical. Although practices among physicians can vary and there might be other inputs that influence the therapy decisions, we were told that incorrect modifications of these values or aggregates will likely lead to a change in the delivered therapy and can cause serious patient harm. Moreover, the status of the implant and leads connecting the implant to the heart is integrity critical. If these are thought of malfunctioning, this might trigger their replacement which requires surgery. Wtf. They should not do this remotely

The main subjects in the system that we had access to include the administrators of the clinic, physicians and the medical implants. Each implant can be seen as a user with write access to the corresponding patient's implant status and measurement data. Main objects are patient related information which are entered by physicians, as well as measurement and implant status data which are entered directly into the system by the implants.

Instantiation. To illustrate Verena's API, we give (simplified) examples of this application. The following collections are relevant:

collection	fields in a document
patients	(groupID, patientID, patient_name, profile)
patient_groups	(groupID, group_name)
patient_measurements	(recordID, patientID, heart_rate, timestamp)

Patients are organized into four groups based on their cardiac disease. Each patient is present in only one such group. These groups also represent the unit of write access control. Physicians are granted write access to one or few of these groups, and they can modify only patient profiles in those groups.

Granularity of AC

The collection patient_measurements contains measurement data originating from a patient's medical device and can be modified only by the patient's device.

B. Trust Contexts

Trust contexts are the units of write-access control in Verena. A trust context, identified by a *unique name*, consists of a set of users, called *members*. We also refer to this set of users as the trust context membership list or access control list (ACL).

Each query whose results are integrity critical runs in a particular trust context; only the members of that trust context could have affected the result of the query.

The user who creates a trust context is the *owner*. The owner of a trust context can add other members to the trust context ACL or remove them from it. Currently, only the owner of the trust context can manage its members, but delegating this to other users is straightforward. We discuss how Verena maintains and verifies the membership of trust contexts in §V-D.

Returning to our running example and its protection requirements, the developer should define one trust context per disease group (whose name can be "groupID") containing the physicians allowed to modify the corresponding patient profiles. The contents of patient_groups can be changed by members of an "admins" group so the developer declares a trust context for "admins", too. Furthermore, the data in collection patient_measurements as well as the query results on this data can be modified only by the patient's device. Hence, we also have a trust context per patientID.

C. Integrity Query Prototypes

In Verena, the developer specifies the desired integrity policy via a set of integrity query prototypes (IQPs) with associated trust contexts. The IQPs are query patterns which specify that a certain set of read queries run in a certain trust context; only members of the trust context may affect the result of those queries. The integrity specification is therefore associated with read queries and not with data – nevertheless, the policy implicitly carries over to data, because data is accessed through queries. Moreover, the IQPs tell Verena what computation will run on the data so that Verena prepares data structures for verifying such computation. We now show the syntax of an IQP and explain each element in it:

```
iqp = collection.IQP ({
    trustContext: unique_name or tc_field,
    eq-range: [rf1, rf2, ...],
    ops: {o1: [f1, ...], o2: [f1', ...], ...}})
```

- "iqp" is an IQP handle.
- "collection" is the collection on which a query with this pattern runs.
- "trustContext" specifies the trust context.

The trust context can be a name, such as "admins" or can be the name of a field in this collection, such as groupID in the patients collection. In the first case, there is one fixed trust context for all documents in this collection.

In the latter case, there can be different trust contexts for different documents. For example, if "patients" contains documents (groupID: "A", patientID: "10", ...) and (groupID: "B", patientID: "11", ...), and an IQP specifies the "trustContext: groupID", the trust context for the first document is "A" and for the second document is "B".

Each trust context must have a unique name. For example, if both patientID and groupID are trust contexts for some IQPs

Operation	Explanation
project: [f_1, ...]	projects the fields f_1, ... from the document
count	returns the number of documents
sum: f	returns the sum of the values in the field f
min/max: f	returns the minimum/maximum value over the data in field f
avg: f	returns the average of the values in the field f
sum_F: [f_1,...]	a more generic aggregate: returns the sum of a general function F whose inputs are [f_1,...]

TABLE I: Operations supported in read queries.

and can both have a value of 2, the developer should choose trust context names of the form "patient 2" and "group 2", in order to differentiate them. If it is desirable for patientID to remain an integer, the developer could include another field in the document, which will serve as the trust context field, e.g., "group_tc". In the rest of the paper, we assume that the trust contexts are the IDs and they are unique.

- "eq-range" specifies that the queries corresponding to this query pattern filter documents by range on the tuple (rf_1, rf_2, ...). A set of filter possibilities fit in this pattern. For example, if the IQP for "patient_measurements" contains "eq-range: (patientID, timestamp)", a query could have an equality match by patientID and a range match on timestamp, or there can be equality on both fields. Our current implementation supports only one range filter, namely the last declared field in the tuple (rf_1, rf_2, ...), with the rest of the fields being used as equality filters. However, Verena can be extended to support more complex filters (e.g., multidimensional range queries and text search queries) by simply using ADSes that support such operations [37].
- "ops" indicates the projections and aggregations performed and on what fields. The operations supported are listed in Table I.

Nice way of declaring a schema. Similar but more general than hails' mongo declarations

Verena will protect the integrity of all fields specified in an IQP, namely the fields projected, aggregated, in eq-range, or in trustContext – these fields can be modified only by members of the corresponding trust context. We call these fields the *protected fields* of an IQP.

Let us walk through an example. In the medical application, a physician can fetch the recorded heart rates of a patient over a period of time to visualize how the heart rate fluctuates in that time period. Additionally, a physician can view the average heart rate over a time period.

The first read operation is a projection on the heart_rate field, and the second is an average computation on the same field. The trust context in both operations is designated by the patientID field; only the patient's implant is allowed to provide these measurements. In the medical application, this entity is represented as a user with patientID. Moreover, the operations use the timestamp field as a range selector. Consequently, to integrity protect these operations we can define the following IQP:

```
iqp_measurements = patient_measurements.IQP ({
    trustContext: patientID,
    eq-range: timestamp,
    ops: {project: [recordID, heart_rate], avg: [heart_rate]}})
```

D. Queries API

Once the developer specifies the necessary IQPs, which reflect the application's integrity specification, he can express and issue queries *in the same way* as in a system without Verena, by invoking "find" and "aggregate" on the corresponding IQP handlers This minimizes the amount of effort needed by the developer to enable Verena in existing web applications.

An example query for listing the average heart rate of patientID 121 over a period of one month is:

```
iqp_measurements.find ({
    patientID: 121,
    timestamp: {"$gte": new Date("2016-03-01"),
                "$lte": new Date("2016-04-01")}})
```

Why subset?

A read query must be a subset of the queries described by the corresponding IQP. Moreover, if the trustContext of this IQP is a field, the query must specify its concrete value (for example, "patientID: 121").

The developer does not have to specify IQPs for write queries, and simply invokes "insert", "remove", or "update" operations on the desired collection. The access control for write queries is derived from read queries. For write queries, Verena checks against *all* IQPs declared whether the current user is allowed to perform them. We elaborate on these checks in §V-C.

Supported Functionality. The read queries supported by Verena are those that can be expressed using an IQP. The write queries supported are in Table II: insert, delete, update. Verena currently supports update and delete queries only by id, but extending to eq-range style filters is straightforward.

E. Querying Across Trust Contexts

So far, each read query specifies one trust context in which it runs. We now discuss how Verena supports queries spanning multiple trust contexts.

In the medical application example, recall that patient profiles are categorized in groups according to their disease and physicians may only modify profiles within certain groups. The following IQP enables fetching the *complete list* of patients within a group:

```
iqp = patients.IQP ({trustContext: groupID,
                     ops: {project:[all]}})
```

and the following IQP enables fetching all groupIDs from "patient_groups" using the "admins" trust context:

```
iqp_groups = patient_groups.IQP ({ trustContext: "admins",
                                   ops: {project: [all]}})
```

In our running example, a physician may also view the list of all patient profiles from *all* patient groups. Clearly, this query spans multiple trust contexts. In a non-Verena system,

Can you define multiple iqps for a collection?

the developer can simply run a read query fetching all entries in "patients". However, if the server is compromised, the list of patients returned can be incomplete. To ensure completeness using Verena, the developer would need to do more work. He should fetch all the groups using "iqp_groups", loop over the groups returned, fetch all patients in each group using "iqp" and merge the results. This results in a complete set of patients, but requires more work from the developer.

What else? Also slower: mot leveraging dl queries

To make the work of the developer easier, we extend slightly Verena's API with a mechanism called *completeness chain*; this mechanism essentially does the above work automatically for the developer. The completeness chain retrieves the involved trust contexts of a query by querying a different IQP and trust context, called the *root trust context*, which *endorses* the relevant trust contexts. In other words, the root trust context protects the list of trust context names of the query we want to execute, and thus we leverage it to establish completeness for that query. The developer simply runs the query:

Whats this? Why is it necessary?

```
iqp.find ({ group: iqp_groups.find({},{groupID:1}) },
          { ... })
```

The inner query projects the "groupID" fields from all documents in "patient_groups". Fig. 3 and §V-B describe how Verena implements the completeness chain mechanism.

Alternative. It is worth mentioning an interesting alternative to completeness chains, which demonstrates the expressivity of Verena's trust contexts. The developer can specify a new trust context "all_physicians", which contains the set of all physicians, and an IQP "iqp_all_patients" that fetches all patient profiles (across all patient groups) in the trust context of "all_physicians". Then, the developer can directly fetch all patient profiles by running "find" on "iqp_all_patients". In this way, any non-physician cannot affect the completeness or integrity of the patient list. However, a physician who is not authorized to modify a certain group can now affect the completeness and integrity of patient profiles in that group. As a result, the integrity guarantee provided by the above IQPs is weaker than with completeness chains, and not sufficient for this application.

However, for applications with different access control requirements or different threat models, a developer might find this alternative sufficient. In this case, verification of aggregates is faster than with the completeness chain. Due to the layout of Verena's data structures described in §V, the Verena client checks one aggregate value overall instead of one aggregate value per trust context with the completeness chain.

F. Deriving Trust Contexts From User Input

In some applications, the trust context for running certain queries is derived from user input. This requires special care from the developer and the user. Such a situation arises in applications where *anyone* can create units of data and give write access to others. For example, in a chat application, anyone can create a room and invite certain users to those rooms. Only the invited users may modify the contents of a chat room. This situation does not occur in the medical application because access control is rooted in a fixed entity, namely the "admins" trust context, which endorses and manages access to the trust contexts of patient groups.

In the chat application, a natural trust context for the messages in each room is the room name. A user, say Alice, reads the list of room names and clicks on the room she wants to visit. She expects the messages in the room to come from authorized users, and makes decisions based on them. However, an attacker can also create a room with the same name or a syntactically similar name ("business" vs. "busines") tricking Alice into clicking on the attacker's room. The contents of the attacker's room are certified by the attacker, so Verena does not trigger an integrity violation.

Hence, in such cases, the developer must display *unambiguous names* to users. In order to do this, the developer can choose human-friendly names for trust contexts (e.g., the name of a room, as defined by a user) and then, display directly the trust context names in a prominent way to the user. Our hash server prevents two trust contexts from having the same name, and one can also expand this protection to prevent two trust contexts from having syntactically similar names. Moreover, the developer can display the owner of a trust context. Depending on the use case, the developer can display both the trust context name and its owner, or either of the two, in order to help the user verify the authenticity of the displayed data. The user needs to perform this check, for example in order to verify he is entering the intended room. This requirement is similar to phishing prevention where the user needs to check the URL he is visiting.

This gets ugly. Utf

This is not solved. Attacker model makes this a plausible vector

G. Integrity Guarantees

The guarantee Verena gives to a developer, given the assumptions in the threat model (§II), is, informally:

> If Verena does not detect a corruption, the result of a read query (find or aggregate) that corresponds to an IQP with a trust context `tc` reflects a correct computation on the complete and up-to-date data (according to linearizability semantics), as long as all clients running on behalf of the members of `tc` (or all involved trust contexts in the case of a completeness chain) follow Verena's protocol.

Whats this?

In particular, the query result could not have been changed by a malicious server or any user outside of the relevant trust contexts. Moreover, a data item is "up-to-date", or fresh, if it reflects the contents of the latest committed write as in linearizability semantics. In particular, the server cannot perform fork attacks [32, 34] because every client can always get the latest committed write of any protected data.

The resulting guarantee to the user is:

> The webpage consists of: (1) the authentic developer's code; (2) correct and "up-to-date" information (data or query computation results) generated only by

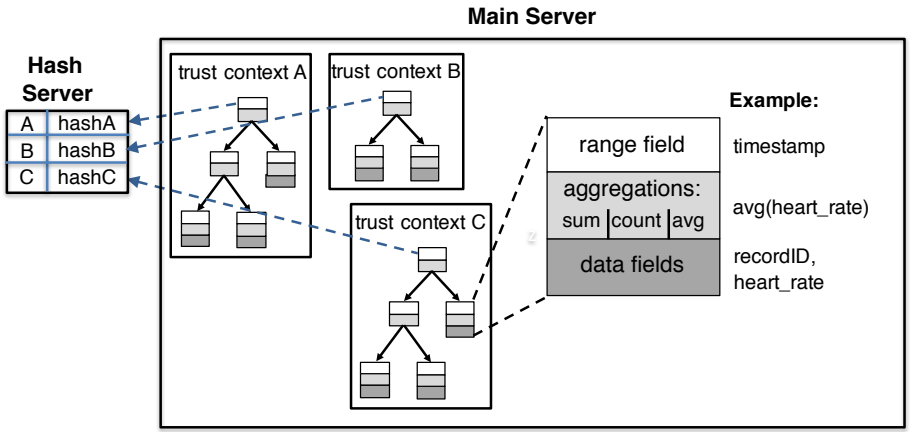

Fig. 2: Verena maintains a forest of ADS trees, in order to protect the data associated with different trust contexts.

Write Queries	Explanation
insert(d)	inserts a document d
update(id, d)	updates the document with identifier id with data from the document d
remove(id)	deletes the document with identifier id

TABLE II: Write operations in Verena.

authorized users.

Verena does not guarantee availability of the server.

V. Integrity Protection Mechanism

We now describe how Verena enforces the integrity policy the developer specified.

A. ADS Forest

Verena leverages authenticated data structures (ADS) [20, 23, 30, 31, 33, 37, 49] as its underlying integrity protection building block. The ADS [31] we use consists of a search tree sorted by the eq-range field(s) and combined with Merkle hashing. We refer the reader to appendix A, where we provide needed background on ADSes.

Based on the IQPs declared by the developer and the write operations that are issued throughout the application's lifetime, Verena creates and maintains a forest of ADSes, as illustrated in Fig. 2. For each IQP, Verena creates one ADS per trust context that is used in queries of that IQP.

For example, consider the IQP we discussed before:

```
iqp_measurements = patient_measurements.IQP ({
    trustContext: patientID,
    eq-range: timestamp,
    ops: {project: [recordID, heart_rate], avg: [heart_rate]}})
```

Based on this IQP, every patientID constitutes a trust context, and Verena will maintain one different ADS for every value of patientID, in order to protect the data and aggregation operations specified by the IQP (in this case, the projection of recordID and heart_rate, and the average calculation on heart_rate). Appendix A explains how the ADS organizes and stores the protected data.

As shown in Fig. 2, the forest of trees is stored at the main server. The hash sever stores only the Merkle hash roots (one entry per tree, containing the root hash and additional necessary information, as described in §VI).

B. Completeness Chain Implementation

ADS trees can be logically nested within other trees as shown in Fig. 3. The completeness chain mechanism logically nests ADSes within another ADS. In this example, a trusted entity, such as the administration of the medical application, uses a static, i.e., predefined, trust context, named "admins", owned by the system administrator, to manage the patient groups. One of the protected fields is used to store the trust context name of each group. This field can be used as a reference to identify all the correct trust contexts that correspond to the patient groups, which in turn protect the patient profile data. Thus, we can use the "admins" trust context as a root trust context to establish completeness, for the query that reads patient profile data from all (unspecified) groups.

C. IQP Analyzer

The IQP analyzer checks whether a user can run a certain query based on the IQPs defined and the trust contexts to which this user belongs.

For each read query, the Verena client ensures that the query matches the IQP handle it was invoked on. A query matches an IQP if all of the following conditions hold:

- the query filters on the same list of fields as in eq-range of the IQP or on a prefix of these fields,
- the query performs a subset of operations and aggregates from "ops" of the IQP, and,
- if the trust context of this IQP is a field instead of a fixed trust context, the query specifies the value for this field (e.g., "patientID: 121" in the query in §IV-D).

When inserting or deleting a document, the Verena client and server check that the user who inserts this document is a

Function	Explanation
declareIDP(*url*, *pubkey*)	Specifies the *url* and *pubkey* of the IDP.
createAccount(*uname*, *passw*, [*creator*])	Creates an account for user *uname*. Must be called by the user when his account is created or, if a creator user is specified, by the creator.
lookupUser(*uname*, [*creator*])	Looksup the user *uname* as created by *creator*. If the creator is not specified, Verena considers the default which is the IDP.
login(*uname*, *passw*)	Logs in user *uname* with the specified password.
logout()	Logs out the currently logged-in user.
createTC(name)	Creates a new trust context *tc* with name *name* owned by the current user.
isMember(*tc*, *user*)	Returns whether *user* is member of the trust context named *tc*.
addMember(*tc*, *user*)	Adds *user* to *tc* only if the current user is the owner of the trust context *tc*.
removeMember(*tc*, *user*)	Removes *user* from *tc* only if the current user is the owner of the trust context *tc*.

TABLE III: User and trust context API in Verena. Each function runs in the user's browser and *current user* denotes the currently logged in user who performed this action.

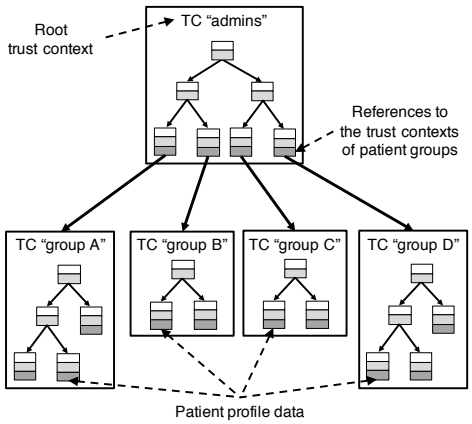

Fig. 3: Illustration of the completeness chain mechanism.

member of *all* the trust contexts defined by any IQP on this document. When updating a field f, the Verena client and server check that the user is a member of each trust context defined by an IQP on this document that has f as a protected field. When updating a field that is a trust context for an IQP, the user performing the update must belong to both the old and new trust contexts.

Recall last week's checks in hails

Of course, if both the Verena client and server performing these checks are compromised and collude, they will not perform these checks and will allow unauthorized actions. However, as we discuss in §VII, the Verena clients running on behalf of honest users will detect and flag this issue: the unauthorized client was not able to sign ADS updates with an authorized public key.

D. Trust Context Membership Operations

As described in §IV-B and §IV-C, write access control is expressed by associating trust contexts with protected data, through IQPs. Only the members of a trust context are allowed to affect the results of a query associated with that trust context. The owner (creator) of a trust context is responsible for managing the membership, or in other words the access control list (ACL), list of the trust context, by adding and removing users. We note that one can create additional groups for further levels of nesting and have a trust context consist of a list of users and groups of users. For simplicity, we do not describe groups beyond trust contexts in this paper. Also recall that, each trust context is identified by a unique name. Table III shows the API for adding or removing trust context members.

The ACL of a trust context is a piece of information that needs to be integrity protected, just like other sensitive data in the web application. Verena internally maintains a collection for storing the trust context ACLs and protects it by declaring an appropriate IQP. Consequently, both ACL modification operations, as well as read operations for verifying whether a user belongs to a trust context ACL, are integrity protected. The corresponding entries on the hash server, i.e, those that store the latest root hashes of the ADSes protecting the ACLs of trust contexts are created in a special way (see §VI), to make sure that only the owner of a trust context can update the root hash of its ACL, and thus manipulate the ACL.

We note that an extra step has to be performed when removing a user from a trust context. As we describe in §VI, each hash server entry stores the public key PK of the last user that modified the entry. When the owner removes a user u from a trust context `tc`, the owner must update the entries at the hash server that correspond to ADS trees for `tc` (these are the ADSes that protect data associated with `tc`) that were last modified by u; the owner makes a no-op modification so these entries now appear modified by the owner, which is valid, because he is a member himself. This update is necessary because, without it, clients verifying if the last modification to the ADS tree was permitted will notice that this modification was performed by a user who is not in the trust context anymore.

Membership Verification. As part of verifying the result of a query (described in detail in §VII), the Verena client needs to check that the user who last modified the relevant protected data was authorized to do so. In other words, the Verena client needs to verify that a user u with public key PK is a member of a given trust context `tc`. To construct such a proof, the server provides to the client the binding of a username u to PK along with the signature from the IDP for this binding. Based on this signature, the client can verify that user u is indeed the owner of PK. Subsequently, the server has to prove that u is a member of the trust context `tc`. For this goal, the server

This integrity protection is not the same as one provided by IFC. How are they different?

fetches the entry for `tc` from the hash server and produces a proof from the ADS that protects the ACL of `tc`, in a process similar to any integrity protected read query.

VI. Hash Server

The hash server has a simple functionality, similar to a key-value store. Its task is to store the most recent root hash for each ADS that exists in a Verena application, together with information about which user made the latest update. The hash server provides this information signed for authenticity to Verena clients. The clients use it to verify that the data they read is fresh and complete.

The hash server stores a map, in which the key is an ID and the value is an entry E = (hash h, version v, public key PK, flag `fixedPK`). The version v helps serialize concurrent operations to each entry. Depending on the value of `fixedPK`, we distinguish two types of entries, which we describe below.

Trust Context ACL Entry. Entries of this type store the root hash h of the ADS that protects the membership list (ACL) of a trust context. The ID of such an entry is uniquely derived from the trust context to which it corresponds. The version v indicates the number of modifications made so far to this entry. The public key PK belongs to the user who created the trust context, i.e., the owner, and `fixedPK` is true to indicate that only the creator of this entry is permitted to modify the entry. This reflects the fact that only the owner of the trust context is allowed to manipulate the trust context ACL.

ADS Entry. Entries of this type store the root hash h of an ADS that protects application data associated with some trust context. The ID of such an entry is uniquely derived from the IQP and trust context to which they correspond. The version v of an entry E indicates the number of modifications made so far to this entry, and PK is the public key of the user who last modified the hash of this entry. `fixedPK` is false indicating that anyone is allowed to modify this entry. The hash server does not check if the client modifying this entry was allowed to modify it. Instead, since all hash server requests go through the main server, the main server must check that the client is authorized. If the server misbehaves and allows unauthorized modifications, honest Verena clients will later detect this misbehavior by checking if the PK of the latest modification was allowed to perform this modification.

The hash server does not need to understand how each entry is used for integrity enforcement. It only implements the following simple interface consisting of two functions, `HS_GET` and `HS_PUT`:

HS_GET(ID):
1: **return** map[ID]

HS_PUT(ID, E_{old}= (h, v, PK), E_{new}= (h', v', PK', fixedPK'), sig(ID, E_{old}, E_{new})):
1: Verify sig on (ID, E_{old}, E_{new}) using PK'
2: **if** ID not in map and v' = 1 **then**
3: map[ID] = (h', 1, PK', fixedPK'); **return** true
4: **if** ID not in map **then return** false
5: E = map[ID]
6: **if** E.fixedPK and PK' $\neq$ PK **then return** false
7: **if** E.v = E_{old}.v and v' = v+1 and E.h = E_{old}.h and E.PK = PK **then**
8: map[ID] = (h', v', PK', E.fixedPK); **return** true
9: **return** false

As shown in Fig. 4, when a Verena client makes a request to the hash server, the client attaches a random nonce. The hash server assembles the response as above and then signs it together with the request and the nonce. The signature and nonce prevent an attacker from replacing the response of the hash server with an invalid or an old response.

The hash server can receive batched requests of the same type from the same client. The hash server signs all the responses into one signature, for increased performance. When the client sends multiple `HS_PUT` requests, the hash server executes them atomically: it executes them only if all of them return true.

VII. Communication Protocol and Query Processing

We now describe the protocol that governs the interaction between the client, the main server, and the hash server, as well as the operations that are executed during the processing of read and write queries.

Fig. 4 shows the communication protocol in Verena. The sequence of operations in this protocol is the same no matter what the query from the client is: whether this query is reading some data, performing an aggregate, writing some data, or adding a member to a trust context. Only the details of the operations differ.

When issuing a query, the Verena client adds a randomly generated nonce to the query, to be included in the hash server's signed response. Based on the query, the main server derives a set of hash server requests whose results will help in assemblying a proof of correctness for the query's results. The server submits them together as part of one big request to the hash server. The hash server executes the request atomically, as explained in §VI. Then, it signs its response together with the request and the client's nonce, and provides the signed response to the main server. The server computes the result of the query based on the server's state and uses the hash server's signed response to prove correctness of the query result to the client. The main server often needs to add extra information to show that some data hashes to the hashes provided by the hash server. The nonce prevents the main server from doing replay attacks on the hash server and serve old data to the client.

In Fig. 4, Step 1 is explained in §V-C. Step 3 is explained in §VI. We next describe Steps 2 and 4 for both read and write operations. For each query, if the server is honest, it still checks the regular read and write access control of an application, as coded by the developer, and rejects a query if the issuing client is not authorized to execute it. If the server is malicious, the

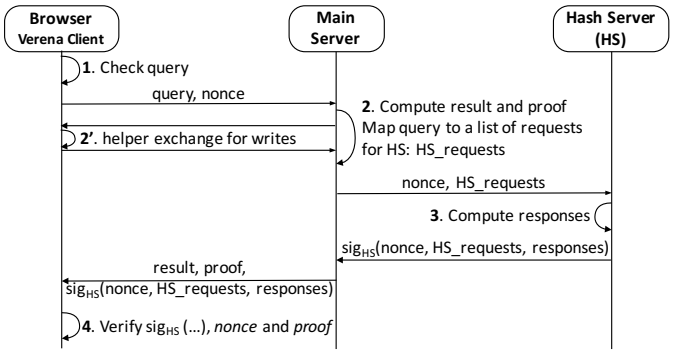

Fig. 4: Communication protocol in Verena. Step 2' consists of an additional roundtrip that happens only for write operations.

server might skip this step, but write access control specified by IQPs will still be enforced by Verena; clients will detect that the server violated the integrity access control specified in IQPs.

A. Read Query

Read queries can be projection or aggregation queries. During Step 2, the server executes:

1) Create an empty list of requests for the hash server, called `HS_requests`, and an empty list for proofs to be given to the client, called `proof_list`.
2) Execute the query on the database and produce a result `result`.
3) Identify the relevant ADS instance for the trust context of this query or ADS instances for a query using a completeness chain. Assemble a proof of correctness of the query result based on these ADSes and their roots and add the proof to `proof_list`. Add requests for the hashes of the roots to `HS_requests`.
4) Identify the last client who modified each ADS. Assemble a proof that this client is in the trust context for the relevant IQP (per §V-D), add the corresponding hash requests to `HS_requests` and the proof to `proof_list`.

In Step 4 of Fig. 4, the client verifies the proof from the server. For each ADS involved in the query, the client verifies (1) the ADS proof, (2) that the root hash of this ADS corresponds to the one from hash server's signature, (3) the hash server's signature, (4) the server's proof that the last client who changed the root hash (as specified by the public key in hash server response) was authorized, as explained in §V-D.

Only save last. Will saving more give you anything more?

B. Write Query

A write query can be an insert, update, or remove. Verena provides linearizability guarantees: there is a total order between all read and write queries, and each read will see the latest committed write. A client considers a write query committed when the protocol in Fig. 4 completes successfully.

To prevent the server from cheating during serialization, a natural solution is to have the hash server serialize requests. However, this strategy will increase the complexity at the hash server and our goal is to keep the hash server simple. Instead, the main server will do the serialization work in a verifiable way. The only job of the hash server is to ensure that each `HS_PUT` to an entry increments the version of the entry. Based on the version number, clients can verify that the server did not serve an old hash or attempted a fork attack [32].

A write query (e.g. insert) can cause modification of multiple ADSes. The server serializes the changes to these ADSes by locking access to each ADS involved. Parallel write queries affecting different ADS structures can still proceed concurrently.

Clients can issue a delete or update query to a certain document `ID`. If the developer wants to delete or update documents selected by a filter, the developer should first fetch those documents using a read operation and then update them by `ID`. Verena's design can be extended to enable such queries directly by employing verification as for read queries, but in interest of simplicity, we do not describe these changes here.

Slow

We now describe the steps involved in an insert; delete is similar. Update proceeds as a delete and insert that happen at the same time (so there is no need to run the communication protocol twice).

During an insert operation, the client must help the server update ADS trees. For each ADS tree, the client first checks that the ADS tree at the server is correct. Namely, its root hash matches the corresponding hash at the hash server and was changed by an authorized client. The client does not have to download the entire ADS tree to perform this check; only the relevant path in the tree is required. Then, the client inserts the new value and recomputes the new root hash. It signs this hash and provides it to the main server, to be included in a hash server `HS_PUT` request.

Does this work for concurrent access?

To avoid the need for an additional round trip between the main and hash servers, the main server maintains a copy of the hash server map. Thus, the client obtains the hash root $\texttt{hash}_{\text{old}}$ from the main server instead of the hash server. Of course, the main server could provide an incorrect value, but both the hash server and the client detect this behavior as follows. When the client performs the update, the client provides a new hash along with $\texttt{hash}_{\text{old}}$ in a signature sent to the hash server as part of `HS_PUT`. The hash server checks that $\texttt{hash}_{\text{old}}$ matches the value at the hash server as discussed in §VI.

:(

Due to updates, some data content may cycle back to an old hash. The version numbers prevent a malicious server from replaying previous updates on this repeated hash value.

During Step 2 and 2', the server runs:

1) Check, using regular access control, if the client is allowed to write. If not, return.
2) Identify the relevant ADSes $A_1 \ldots A_n$ that need to be updated. Acquire a lock for each one of these.
3) Send a message to the client containing a proof for each ADS A_i as discussed above. Instead of contacting the hash server for the tuple E = (hash h, v, PK), send this information from the server's storage. Also, send a proof that PK belongs to a user who is allowed to make the change as in the read operation.

In Step 2', the client:

This can't be true. These data structures contain sensitive data that may be hard to protect (confidentiality wise) without having sone so from the start.

1) Verify all the proofs as in a read operation.
2) If verification passes, for each ADS involved, provide nonce and $\texttt{sig}_{user}(\texttt{ID}, E, E_{new})$, where ID is the id of the corresponding entry at the hash server, E_{new}= (h', v' = v+1, PK'), where h' is the new hash after the change.

The server:

1) Check the client's signature, h', v', and PK'. If everything verifies, update the database, the ADS trees, and send E_{new} and $\texttt{sig}_{user}(\texttt{ID}, E, E_{new})$ to the hash server.
2) After receiving the response from the hash server, forward it to the client.
3) Release the locks.

Finally, the client verifies the hash server's response: the signature from the hash server verifies with the nonce and the hash server accepted the change. If so, the write completed. Otherwise, the main server misbehaved. The main server also has a timeout during which it keeps locks on behalf of the client. To provide liveness, if the client takes too long to answer in Step 2', the server aborts this request and releases the lock.

VIII. Informal Security Argument

In §IV-G, we describe the guarantees provided by Verena. Here, we present a high-level argument of why these guarantees hold. To argue that Verena's read queries return results satisfying the guarantees in §IV-G, we show the following two properties. Given a read query q, let ADS be the ADS corresponding to q and tc be its trust context.

1) The hashes of the roots of ADS and tc at the hash server correspond to the *latest* modification by a user in tc.
2) Given the root hash of ADS and tc that satisfy the property above, a client can detect if the query's result does not satisfy the guarantees in §IV-G

The second property follows from the properties of ADS trees (recall that tc is also implemented as an ADS tree). Let us explain why the first property holds. Assuming a trusted hash server, the hash server will return to a read query the latest hash from a write query. Moreover, the Verena client checks for each read query that the latest write was created by an authorized user. At the same time, due to the nonce used by clients when receiving responses from the hash server, a client who committed a write knows that the hash server persisted its update. For queries that span multiple trust contexts, given a root trust context and the properties of ADS trees, the completeness chain mechanism can guarantee the completeness and correctness of the results.

IX. Discussion

Limitations and Extensions. Verena does not support all possible query types, although it supports a common class of queries. §IV-D describes the queries our current system supports. Nevertheless, the overall Verena architecture is mostly agnostic to the underlying ADS. The literature provides ADSes for other types of queries, such as multidimensional range queries or text search queries [37]; adding them to Verena should be straightforward.

Moreover, Verena does not support triggers: with a trigger, a database server notices when a certain condition on the data is satisfied and contacts the relevant users. If a server is compromised, it can choose not to contact the users. A mitigation to support triggers is to have the client check the triggers after performing an update or periodically.

Hash Server Trust. The design so far assumed that the hash server is trusted. Verena can survive compromise of the hash server as long as an attacker does not compromise both the hash server and the main server. This is simple to achieve: the main server checks the answers provided by the hash server with minimal change to the design so far because (1) the main server stores a copy of the entries at the hash server anyways and (2) all hash server responses pass through the server. The main server can detect misbehavior of the hash server and warn of a potential compromise. :(

User Signature Verification. The signature verification during HS_PUT (§VI) of the user who performs the update can be removed from the hash server and instead performed in the clients. However, we decided to perform this verification on the hash server because it improves client latency and it is a simple operation; it avoids the need for clients to check this signature every time they check a proof involving it.

Data Confidentiality. Verena can be combined with a web framework such as Mylar [41], which protects data confidentiality against server compromise. This results in a solution offering both confidentiality and integrity protection against an active server attacker.

Using Verena Correctly. Verena provides protection only if the developer specifies the integrity policy correctly, which is not always easy. For example, the developer should not make write access control decisions based on data from the server that is not integrity protected. As part of our future work, we are interested in designing a tool that assists the developer and helps him make less mistakes. Can't a malicious server just run arbitrary code?? This is node.js....

X. Implementation

We developed a prototype implementation of Verena in order to evaluate our proposal and demonstrate its feasibility.

Web Platform. We implemented Verena on top of Meteor version 1.1.0.2; Meteor [36] is a JavaScript web application framework that uses Node.js [7] on the server side and MongoDB [5] as the database backend.

We chose Meteor as it offers some desirable features that make it attractive for our implementation. Meteor employs client-side web page rendering based on HTML templates that are populated by data retrieved from the server. This means that there is a clear separation between application code and data. The code, which consists of the JavaScript code, the HTML templates and the CSS files, is signed by the developer and its integrity is verified by a browser extension upon loading, as in [41]. The integrity of data, which is the dynamic part of the web application, is enforced by Verena, according to the policy specified by the developer.

Moreover, Meteor features a uniform data model between the client and the server. In other words, clients are aware of

how the data is organized in the MongoDB backend. This uniformity is beneficial to Verena because it allows both the client and server to understand the integrity policy and the database queries that will be executed. Thus, the server can identify which proofs to accompany the reply with, and the client can identify which proofs to expect from the server.

Meteor uses a publish/subscribe mechanism in which the web server automatically propagates data changes to clients who have subscribed for the results of a certain query. This mechanism is not compatible with the freshness guarantees Verena aims for; a malicious server might not propagate changes to the interested clients. Hence, Verena follows the conventional approach of explicitly requesting the data of interest through the use of RPC requests. One can transform the publish/subscribe mechanism into a pull-based approach, in which the client polls the server periodically, thus providing time-bounded freshness guarantees.

Why no pub sub.

Main Server and Client. We implemented the Verena server and client as a set of Meteor packages. The main server's implementation is 5100 LOC. The main component consists of approximately 3100 LOC. The storage and manipulation of the authenticated data structures, as well as the production of the necessary proofs is implemented as a separate service, which runs as a Node.js process and consists of about 2000 LOC. We note that in this prototype implementation, the ADSes are stored in-memory, and not persisted on disk. For a production-quality implementation of Verena, the system should implement the ADSes within the database itself for better performance. The ADS manipulation logic is also replicated to the client, so that the client can verify the proofs presented by the server. We use 224-bit ECDSA for public-key operations, and SHA-256 as a cryptographic hash function. We perform most cryptographic operations in JavaScript using the SJCL library [11]. Nevertheless, to improve client performance, we implement ECDSA signature operations as a Google Chrome Native Client module [6].

This skews benchmarks...

Hash Server. The hash server is implemented as a Go HTTP server, backed by a RocksDB [10] persistent key-value store. The cryptographic operations (ECDSA signing and verification) are delegated to a separate process, written in C, which uses OpenSSL [8] (version 1.0.2d). The reason is that the native Go ECDSA implementation is currently significantly slower than the OpenSSL one. The hash server consists of 630 LOC in total (497 for the Go component and 133 for the C component), not counting standard libraries such as OpenSSL. By contrast, an application server's total LoC consists of our Verena server's implementation plus the server-side code of the actual application. The actual application can easily have thousands to tens of thousands of lines of code.

XI. Evaluation

We used our prototype implementation to evaluate the performance of the various components of Verena. The evaluation setup is as follows. Verena's main server ran on a Macbook Pro "Mid 2012" (iCore7 2.3 GHz), while the hash server ran on an Intel Xeon 2.1 GHz processor with fast SSD storage, with a recent version of Ubuntu Linux installed. To perform our end-to-end latency measurements (§XI-A, §XI-C) we used a client using the Chrome browser, version 49, on a second Macbook Pro "Mid 2012" laptop. To measure throughput (§XI-B), we employed multiple machines running many concurrent client instances using the headless browser PhantomJS [9], in order to saturate the system under test. All the machines that we used for the evaluation were connected to the university network.

Is this set up realistic?

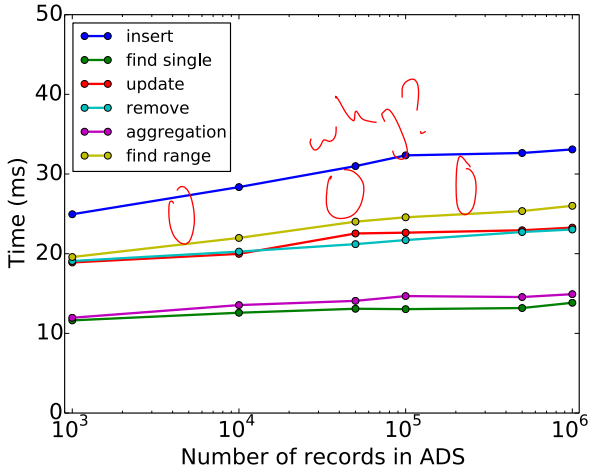

Fig. 5: End-to-end latency of various read and write operations in Verena.

A. End-to-End Latency

Fig 5 shows the end-to-end latency of the basic operations that are performed by Verena, i.e, write and read operations on a single ADS (or in other words a particular trust context). More concretely we tested insertion, update and removal of records as well as read operations, namely, fetching a single record ("find single"), fetching a range of 20 records ("find range") and computing an aggregate value (sum) on a particular field over a range of half of the currently inserted records. Each inserted record had a size of 1 KB.

We measured and averaged the latency of these operations, over 1000 iterations, for different numbers of inserted records in the ADS. We notice that latency slightly increases (for all operations) as the size of the ADS becomes larger. Even for an ADS size of 10^6 records, all operations, take less 30ms on average, except insert which takes slightly over 30ms for large ADS sizes.

B. Throughput

Main Server. We measured the throughput of Verena for read and write operations issued by multiple concurrent clients, on ADSes containing 10^4 records. When clients perform only read operations (in specific, fetch a range of 20 records) the average throughput is 200 (±8) requests/sec. When performing only write operations (specifically, inserting records to different trust contexts so that they can be processed in parallel), the average throughput is 156 (±10) requests/sec. Finally, when clients perform a mix of the above read and write operations (4

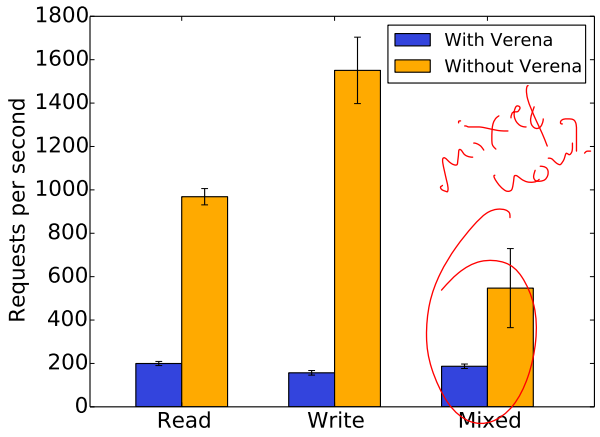

Fig. 6: Throughput evaluation on the main server when running with Verena enabled, as well as without Verena.

Are these numbers good?

Operation type	Requests/second	Std.Dev.
GET	8420	673
PUT	2100	92
Mixed	5890	548

TABLE IV: Hash server throughput.

reads for 1 write) the average throughput becomes 187 (±10) requests/second.

Fig. 6 displays the above throughput measurements and contrasts them with the throughput of the same server, but without Verena. As expected, the throughput is higher in all cases when Verena is not activated. We note that performing operations to different ADS trees can be run in parallel and independent of each other, which can increase the overall throughput when using additional server machines.

Hash Server. We also measured the throughput of the hash server, and the results are displayed in table IV. When the hash server receives only `GET` requests, the average throughput is 8420 (±673) requests/seconds. When it receives only `PUT` requests the average throughput is 2100 (±92) requests/second. This result is as expected because `PUT` requests perform an additional signature verification. When the hash server receives a mix of requests (4 `GET` requests for 1 `PUT`) the average throughput is 5890 (±548) requests/second.

It is important to note the significant difference in performance between the main server and the hash server. The hash server, which provides a very simple functionality, compared to the main sever, achieves an order of magnitude higher throughput than the main server.

C. Evaluation on the Example Medical Application

As introduced in §IV-A, our running example is a remote patient monitoring system, that is used to connect cardiac implant (e.g., pacemaker) patients with their clinics and physicians. After receiving access to the provider (clinic/physician) web interface of a remote monitoring system and after discussions with a cardiologist, we implemented an example application of this system in Meteor and used Verena to secure its most relevant functions.

We specified three types of trust contexts as discussed in §IV-B. In our measurements, we create 1000 patients per group, totaling 4000 patients. We evaluated the average latency (over 1000 iterations) of some representative example views that are displayed by the application to the physicians (summarized in table V). For these particular views the integrity policy can be captured with 4 IQPs, three of which were discussed (simplified) in §IV-C and §IV-E. We describe these views below.

Patient List. This view shows a list of patients across all groups, limited to 20 patients per page. This is one of the most complex views. The application has to first perform a query on the "admins" trust context in order to retrieve all the patient groups and corresponding trust contexts. Then, for each group it needs to perform a range query over that particular trust context to fetch the patients of each group, and then merge the results together. In other words, 5 read operations are required, assuming 4 patient groups. The overall latency for loading this view is 66ms (±7ms), the individual read operation latency being 13ms (±2ms). In other words the total latency for loading the page is approximately the sum of the individual read operations. We note that, through the use of a completeness chain (as described in §IV-E), the developer can express the view using a single read query, and then Verena automatically takes care of performing all five needed queries.

Patients for Review. This view displays all the patients that are flagged for review. This view performs similar operations with the previous one. Assuming there are 50 patients from each group that are flagged for review, the overall latency for loading this view is 82ms (±7ms).

Patient Profile. This view displays the basic profile information of a single patient. This view requires a single read operation to fetch the particular profile. The overall latency for view is 14ms (±2ms).

EKG. This view displays a 30sec EKG recording of a particular holter episode. The recording contains double precision values of the measured heart electrical activity, sampled at 200 Hz, thus having a size of approximately 50 KB. This view requires a single read operation and the overall latency is 23ms (±4ms).

Average Heart Rate. This view displays the average heart rate of a patient as measured by his monitoring device, over a period of a few months. This view requires a read aggregation operation on the average over a set of samples. The latency for loading this view is 13ms (±3ms).

We summarize the end-to-end latency of the above views in table V. We can see that Verena gives acceptable latencies even for the most complex views that are implemented in this web application. We argue that Verena can be used to protect the integrity of an application such as this medical application, without disrupting the experience of its users.

Application view	Load time (ms)	Std.Dev.
Patient list	66	7
Patient for review	82	7
Patient profile	14	2
Episode EKG	23	4
Avg. heart rate	13	3

TABLE V: Latency for loading views in our example medical application whose data integrity is protected by Verena.

D. More Applications

To further evaluate the expressivity of Verena's integrity API, we considered two other applications: a chat and a class application. Both of these applications are written in Meteor, existed before Verena, were written by other developers, and have a multi-user setting that benefits from Verena's integrity guarantees.

We investigated whether Verena's API can express the write access control policy of these applications and how many IQPs need to be declared for this purpose. As we elaborate below, we found that Verena can express these applications' policy. In a few cases, we found that Verena provides a time-bounded freshness property as opposed to strict freshness. This happened for queries ran via Meteor's publish-subscribe mechanism. As discussed in §X, these rely on the server to notify the client of changes to a query result. Since the server cannot be trusted, clients must instead poll and run this query periodically. Thus, a freshness violation is confined to the period's duration.

Is this fundamental? Can model be extended to account for pubsub?

Chat Application. In kChat, users can create rooms, the creator of a room can invite users to the chat room, and users within the room exchange messages. Each user is allowed to write only in a chat room to which he was invited. In terms of queries, users fetch all messages in a room, perform range queries to select the latest messages, count the messages in the room, fetch the list of people who are online, and so forth.

We found that Verena's IQPs can capture the write-access policy of this application. This means that Verena brings freshness, completeness and correctness for kChat.

Interestingly, there are multiple natural integrity policies for this application providing different integrity guarantees. A common query in this application is fetching all the messages in a room. If the developer trusts the users in the room and wants to protect against users outside of the room, the developer can specify the trust context to be all users in this room. In this case, Verena won't prevent a user in this room who colludes with the server from changing the messages of another user in the same room. The resulting integrity policy is short and consists of 3 IQPs. If the developer wants stricter integrity, namely, to prevent a user in the room from changing the message of another user in the same room, the developer declares two IQPs for this query: the first IQP is for fetching the trust context names of users in the room. The trust context for this IQP is owned by the creator of each room. The second IQP is for fetching the messages of a user in the room with a trust context of that user. The second IQP has should be chained to the first, so that the Verena's completeness chain mechanism can be employed, when performing a read query to fetch messages of a particular room. In this case, the kChat integrity policy can be captured with 4 IQPs.

Homework Submission Application. We also examined a web application used at MIT for managing student assignments, homework and grades for a computer science class. Students are allowed to submit their homework, as well as review and grade the homework of other students. The staff (i.e., the professor who teaches the course and the course assistants) are responsible for managing the student accounts, the homework assignments, the allocation of peer reviews for submitted homework of each student, as well as the final feedback and grade for each submitted homework. Verena can capture the integrity policy of this app with a total of 7 IQPs. CSB

E. Storage and Memory Overhead

Main Server. We evaluated the overhead imposed by Verena in terms of storage and memory and found it to be modest. The memory footprint of adding Verena to the remote patient monitoring application (§XI-C), is roughly 1.2× that of running the application without Verena. This overhead is mostly due to the memory required by the ADS storage service, as well as additional data structures that are maintained by the server for implementing the functionality of Verena.

Regarding the storage overhead, the main contributing factor is the storage of the ADSes. The space required to store an ADS (red-black Merkle binary tree, in our implementation) depends on the cardinality of its nodes, which depends linearly on the number of records that are protected by the ADS. An ADS that contains 10^4 records needs ~1.64 MB (~1.95 MB if the ADS also computes one aggregate value on the record). The relative overhead in this case depends on the size of the records under protection. Assuming an average record size of 0.5 KB, like the user messages in a chat application (§XI-D), the overhead of storing the ADS that protects their integrity is approximately 1.4× the storage required for simply storing the messages themselves. For protecting larger records, as in the medical application for example, the storage overhead of protecting the 30sec EKG recordings (each recording amounts to 50 KB of data, as described in §XI-C) is only ~1.003× the storage needed for storing just the EKGs. Finally, Verena requires approximately 1 KB per user for storing his wrapped private key, public key, and IDP certificate.

Hash Server. The hash server storage requirements are minimal compared to the rest of the system. The hash server stores only one entry for each trust context and ADS that exists in the system. Our hash server prototype stores the users' ECDSA public keys and the SHA-256 digests of the ADS roots as hexadecimal strings, and each entry needs less than 200 bytes of storage. This can be further reduced by using base64 or binary encoding for storing keys and digests, and by reducing the redundant storage of copies of public keys that may be stored in multiple entries. As an example, for the medical application (which we described and evaluated in terms of performance in §XI-C), which contains thousands of users,

Usability? Where did it fail? What happens when integrity is violated!?

trust contexts and ADSes, the storage requirements are less than 5 MB.

XII. Related Work

Verena is the first web framework that provides integrity and freshness of the data and query results contained in a webpage, in the presence of a fully compromised web server.

A. Systems Providing Integrity

File Systems and File Storage. A few file systems, such as SUNDR [32], Sirius [21], Plutus [25], SAFIUS [45], Tresorium [27], CloudProof [39], Athos [22], and Caelus [26] aim to provide integrity in the face of a corrupted server. However, these are constructed for the simpler setting of a file server, so they do not verify query computation results (range queries or aggregations, as well as completeness and freshness for these computations), and do not consider the web setting which is stateless. Some of these systems (e.g., SUNDR) make no trust assumption on the server, but as a result, they either support only one client or do not provide freshness (e.g., SUNDR provides fork consistency which allows a server to present different views to different users). Caelus [26] provides time-bounded freshness by assuming a trusted always-online attestor per client.

Trusted Hardware. Trusted hardware systems such as Haven [12] promise confidentiality and integrity against a compromised server. Unlike Verena, Haven relies on trusted hardware, and places the entire application code in the trusted code base. The only server assumption in Verena is that the hash and main servers are do not collude. Moreover, in Verena, the server application code is not in the trusted code base: in fact, if the application is buggy or exploitable, and thus corrupts integrity protected query results, clients will be able detect it.

B. Work Related to Verena's Building Blocks

In recent years, there has been much progress in tools for generic verifiable computation [14, 38, 46]. Nevertheless, for the web setting considered here, such tools remain impractical. Instead, work on authenticated data structures (ADS) [20, 23, 30, 31, 33, 37, 49] provides better performance and Verena uses these as building blocks. This line of work targets a more specific class of computation, such as aggregations on range queries, thus being more efficient. As discussed in §I, these tools alone are not sufficient for addressing all the challenges of providing integrity protection for web applications.

Verena's hash server is related to the trinket component of TrInc [29]. The trinket is a piece of trusted hardware that stores and increments a counter; it can sign the counter along with a supplied string (e.g., a hash), and ensures that each counter is signed only once. TrInc can be used to provide freshness in SUNDR. Our hash server additionally stores the hash, public key and the `fixedPK` flag. These extra values enable useful properties in Verena, while the hash server still has high performance and small code base (§XI). Providing freshness in SUNDR+TrInc requires clients to download and verify a chunk of the history of changes to an item and to treat each "get" operation as a "put", which results in significantly lower performance.

C. Complementary Systems

A few systems can be used in complement to Verena, to provide a wider range of security guarantees.

Language Approaches/Information Flow Control. A few systems aim to help a developer not make programing mistakes that can lead to integrity or confidentiality violations. Using information flow control and/or language-based techniques, systems such as SIF [19], [42], Urflow [18], and Resin [48] ensure that an application obeys a security policy. However, if an attacker takes control of the server in these systems, the attacker can run any code of his/her desire, bypassing these tools completely, and violating integrity. In contrast, Verena protects against this situation. Nevertheless, these tools can be used in conjunction with Verena to ensure that a developer does not inadvertently leak data, as well as prevent against various client-side attacks.

Confidentiality. Mylar [41], CryptDB [40], and ShadowCrypt [24] aim to provide confidentiality against a corrupted web server, but do not address most integrity properties, such as freshness, completeness or query computation correctness. Mylar, also implemented on top of Meteor, is easy to integrate with Verena.

XIII. Conclusion

Verena is the first web application platform that provides end-to-end integrity guarantees for data and query results in a webpage against attackers that have compromised the web server. In Verena, the user's browser can verify the integrity of a webpage, by verifying the results of the database queries which are used to populate the page content. Our evaluation results show that Verena can support real applications with modest overhead. Verena attempts to close the gap between the research efforts of protecting the integrity of database systems, and the application of this research in one of the most popular use cases of databases, web applications.

References

[1] "Biotronik home monitoring," https://www.biotronik.com/wps/wcm/connect/en_us_web/biotronik/sub_top/healthcareprofessionals/products/home_monitoring/.

[2] "Boston scientific remote patient monitoring," http://www.bostonscientific.com/en-US/products/remote-patient-monitoring.html.

[3] "Keybase," https://keybase.io/.

[4] "Medtronic CareLink network for cardiac device patients," http://www.medtronic.com/for-healthcare-professionals/products-therapies/cardiac-rhythm/patient-management-carelink/medtronic-carelink-network-for-cardiac-device-patients/index.htm.

[5] "MongoDB," https://www.mongodb.org/.

[6] "Native client," https://developer.chrome.com/native-client.

[7] "Node.js," https://nodejs.org/.

[8] "OpenSSL," https://www.openssl.org/.

[9] "PhantomJS," http://phantomjs.org/.

[10] "RocksDB," http://rocksdb.org/.

[11] "Stanford JavaScript crypto library (SJCL)," https://crypto.stanford.edu/sjcl/.

[12] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[13] G. Becker, "Merkle signature schemes, Merkle trees and their cryptanalysis," 2008.

[14] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *Proceedings of the 33rd International Cryptology Conference (CRYPTO)*, 2013.

[15] M. Bodnarchuk, "Why should you use client-side MVC framework?" http://jster.net/blog/why-should-you-use-client-side-mvc-framework, 2013.

[16] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, 2012.

[17] H. Burri and D. Senouf, "Remote monitoring and follow-up of pacemakers and implantable cardioverter defibrillators," *Europace*, vol. 11, no. 6, 2009.

[18] A. Chlipala, "Static checking of dynamically-varying security policies in database-backed applications," in *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[19] S. Chong, K. Vikram, and A. C. Myers, "SIF: Enforcing confidentiality and integrity in web applications," in *Proceedings of the 16th USENIX Security Symposium*, 2007.

[20] S. A. Crosby and D. S. Wallach, "Authenticated dictionaries: Real-world costs and trade-offs," *ACM Transactions on Information and System Security,*, vol. 14, no. 2, 2011.

[21] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*, 2003.

[22] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Athos: Efficient authentication of outsourced file systems," in *Proceedings of the 11th International Conference on Information Security*, 2008.

[23] M. T. Goodrich, R. Tamassia, and A. Schwerin, "Implementation of an authenticated dictionary with skip lists and commutative hashing," in *DARPA Information Survivability Conference and Exposition II (DISCEX II)*, 2001.

[24] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song, "Shadowcrypt: Encrypted web applications for everyone," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

[25] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *2nd USENIX conference on File and Storage Technologies (FAST)*, 2003.

[26] B. H. Kim and D. Lie, "Caelus: Verifying the consistency of cloud services with battery-powered devices," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[27] I. Lam, S. Szebeni, and L. Buttyan, "Tresorium: Cryptographic file system for dynamic groups over untrusted cloud storage," in *Proceedings of the 41st International Conference on Parallel Processing Workshops*, 2012.

[28] S. Lamb, "Vulnerability statistics and trends in 2015," 2015, http://www.contextis.com/resources/blog/vulnerability-statistics-trends-2015/.

[29] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "Trinc: Small trusted hardware for large distributed systems," in *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[30] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2006.

[31] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Authenticated index structures for aggregation queries," *ACM Transactions on Information and System Security*, vol. 13, no. 4, 2010.

[32] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[33] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, "A general model for authenticated data structures," *Algorithmica*, vol. 39, 2001.

[34] D. Mazières and D. Shasha, "Building secure file systems out of Byzantine storage," in *Proceedings of the 21st Annual Annual Symposium on Principles of Distributed Computing (PODC)*, 2002.

[35] R. Merkle, "Secrecy, authentication and public key systems / A certified digital signature," 1979.

[36] Meteor, Inc., "Meteor: A better way to build apps," Sep. 2013, http://www.meteor.com.

[37] H. Pang and T. Kian-Lee, *Query answer authentication*. Morgan and Claypool Publishers, 2012.

[38] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, May 2013.

[39] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, "Enabling security in cloud storage SLAs with CloudProof," in *Proceedings of the 2011 USENIX Annual Technical Conference*, 2011.

[40] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting confidentiality with encrypted query processing," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[41] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan, "Building web applications on top of encrypted data using Mylar," in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[42] W. Robertson and G. Vigna, "Static enforcement of web application integrity through strong typing," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[43] H. Singh, T. Giardina, A. Meyer, S. Forjuoh, M. Reis, and E. Thomas, "Types and origins of diagnostic errors in primary care settings," *JAMA Internal Medicine*, vol. 173, no. 6, 2013.

[44] H. Singh, A. Meyer, and E. Thomas, "The frequency of diagnostic errors in outpatient care: estimations from three large observational studies involving US adult populations," *BMJ Quality and Safety*, 2014.

[45] V. Sriram, G. Narayan, and K. Gopinath, "SAFIUS - A secure and accountable filesystem over untrusted storage," in *Proceedings of the 4thInternational IEEE Security in Storage Workshop*, 2007.

[46] M. Walfish and A. J. Blumberg, "Verifying computations without reexecuting them: From theoretical possibility to near-practicality," in *Communications of the ACM*, 2015.

[47] B. Winters, J. Custer, S. M. Galvagno, E. Colantuoni, S. G. Kapoor, H. Lee, V. Goode, K. Robinson, A. Nakhasi, P. Pronovost, and D. Newman-Toker, "Diagnostic errors in the intensive care unit: A systematic review of autopsy studies," *BMJ Quality and Safety*, 2012.

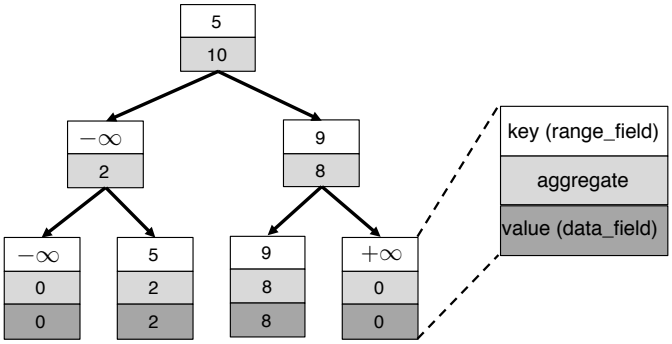

Fig. 7: Example tree-based ADS, supporting aggregations. Node hashes realizing the Merkle hash tree, as well as node color, realizing the red-black tree balancing mechanism used in our implementation, are not shown.

[48] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[49] Y. Zhang, J. Katz, and C. Papamanthou, "IntegriDB: Verifiable SQL for outsourced databases," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.

APPENDIX

A. Background on Authenticated Data Structures

Verena leverages authenticated data structures (ADS) [20, 23, 30, 31, 33, 37, 49] as its underlying integrity protection building block. In fact, Verena does not rely on a specific ADS. In our implementation of Verena we make use of one-dimension red-black binary Merkle hash trees with the ability to support projection queries, as well as aggregation queries based on the tree-based technique of [31].

Here we summarize how such trees work and refer the reader to the literature for a detailed analysis. We assume that the reader is familiar with Merkle hash trees [13, 35]. Consider a database table consisting of two fields (i.e, columns); a range field and an aggregation field, i.e, a field in which stored data is used in aggregation queries. In SQL notation, a user runs queries of the form "SELECT sum(*aggr. field*) FROM table WHERE $x \leq$ *range field* $\leq y$".

Fig. 7 shows an example of such a tree, sorted (i.e., keyed) by the range field. Note that the red-black property is only used for keeping the tree balanced, and hence Fig. 7 omits the color of the nodes. Also not shown in Fig. 7 are the hashes of each node, which constitute the Merkle hash tree. The client, who is the owner of the data stored in the tree, keeps the root hash of the Merkle hash tree, and the untrusted server keeps the entire tree.

Data is stored on leaf nodes. Each node stores a key and an aggregate value; leaf nodes further store the data values. Given our aforementioned example, the keys are the range field values and data stored on leaf nodes is the data of the aggregation field. For leaf nodes, the aggregate value is equal to the data

value itself. For each internal node, the aggregate value is the aggregation over the aggregate values of its children. The tree in Fig. 7 features sum as the aggregation operation.

Assume that the client issues the query "SELECT sum(*aggr. field*) FROM table WHERE $2 \leq$ *range field* ≤ 5". The server responds with the sum of interest, 2 in this case, together with a proof that the sum is correct. We explain briefly what the proof consists of and refer the reader to [31] for more details. The proof consists of two parts. The first part is, for each edge of the interval, the server provides two nodes in the tree whose range fields include the edge of the interval, together with their Merkle hash paths to the root. The client checks these Merkle paths against the Merkle hash root it stores and ensures that the edge of the interval is inside this interval. Note that this is always possible because the keys $\pm\infty$ (containing dummy data values) are also in the tree. The second part of the proof is a minimal covering set for the range $[2,5]$ together with a Merkle hash path up to the root. In our example, this minimal covering set consists of the internal node with key $-\infty$ and aggregate value 2. In general, the covering set is a logarithmic number of nodes.

The client then checks that these nodes cover the range of interest entirely and verifies their hashes and Markle hash paths against the root hash that the client stores. By the properties of Merkle hash trees, if the verification is successful, the server provided the correct aggregate value. Overall, the client performs $O(\log n)$ work per value returned where n is the number of nodes in the tree. A similar computation happens when inserting, updating and deleting data, with some additional details.

Note that the server does not have precomputed the aggregate value for each range. The ADS tree has one data entry (leaf node) per range field value and there is a quadratic number of possible ranges. Clients can query arbitrary ranges, and these ranges could contain a large number of nodes. The server transforms these ranges into a set of subranges, and the client then aggregates the aggregate values for each range. The maximum number of subranges is logarithmic in the number of nodes in the tree. Hence, the client does little aggregation work because it aggregates only a logarithmic number of values.