Aroki Systems

End-to-end Database Encryption

Aroki's Cryptographic Technology

The core cryptographic primitive that underlies our technology is *structured encryption* or STE for short. STE encrypts data in such a way that it can be privately and efficiently queried by a database, relational or non-relational. It was introduced in 2010 by Aroki Systems co-founder Seny Kamara and continues to be an active area of cryptographic research with publications appearing in all the leading cryptography conferences (CRYPTO, Eurocrypt, ACM Computer and Communications Security, IEEE Security Symposium and the Symposium on Privacy-Enhancing Technologies). STE research has been funded by major funding agencies like the National Science Foundation (NSF), The Intelligence Advanced Research Projects Activity (IARPA) and the Defense Advanced Research Projects Agency (DARPA).

STE makes use of standard cryptographic primitives (e.g., AES-256, HMAC-SHA256) in non-standard ways in order to support queries over encrypted data. Unlike fully-homomorphic encryption (FHE) — which is limited to theoretical interest due to its slow performance — STE is fast and achieves strong performance using stable, well-understood, and highly trusted cryptographic libraries.

How does it work? STE works by splitting data into pieces, encrypting them and arranging them in such a way that their contents and ordering appear random. To someone with the secret key, however, the encrypted data appears organized in a very structured and predictable manner. Knowledge of the secret key also allows one to generate an encrypted query that can be safely used to process the encrypted data and efficiently find the pieces relevant to the (encrypted) query. [1]

An Example

In this section, we describe an example of an STE implementation. This scheme is what we refer to as a multi-map encryption scheme because it encrypts multi-map data structures. Multi-maps are also known as inverted indices or reverse indices and can be used to search over file collections very quickly. Here we will see how we can

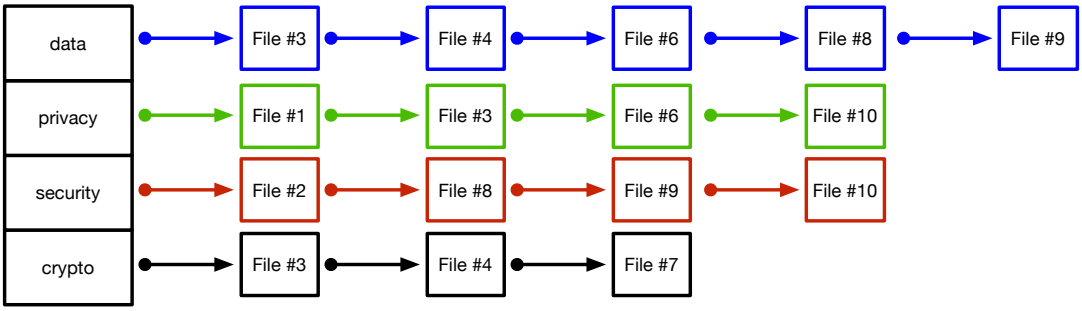
---

[1] For a concrete state-of-the-art example, see *Breach-Resistant Structured Encryption* by G. Amjad, S. Kamara, T. Moataz in the Symposium on Privacy-Enhancing Technologies (PETS); available at https://eprint.iacr.org/2018/195.pdf

encrypt a multi-map and a file collection to perform a (single) keyword search over the encrypted file collection quickly.

Before proceeding, we want to stress that the multi-map encryption scheme we describe here is a toy example and is not what we use in Aroki products. The scheme is only presented to illustrate how an STE scheme might work at a high level. There are many important properties that a real-world implementation would have to achieve that this scheme does not. The actual schemes our products rely on are very different. They handle more complex data structures and more complex queries.

Suppose we have a multi-map data structure that indexes a collection of files File #1,…, File #10. An example is illustrated below. The number associated with a file is its *identifier*. A multi-map stores label/list pairs and supports Get and Put queries, though we will only consider Get queries in this example. Here we assume the files only contain subsets of the keywords “data”, “privacy”, “security” and “crypto”.



The multi-map above stores 4 label/list pairs. The first is the pair with label “data” and list (“File #3”, “File #4”, “File #6”, “File “#8”, File “#9”). The second is the pair with label “privacy” and list (“File #1”, “File #3”, “File #6”, “File #10”) and so on. A Get operation on this multi-map works as follows. If I want to retrieve the list associated with the label “security”, I look up the label “security” and follow the pointer to its list.

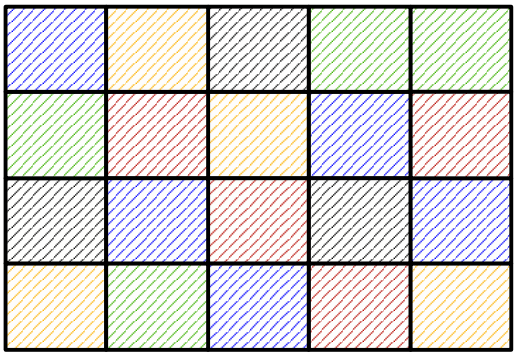
Our goal now is to “encrypt” the multi-map and the file collection in such a way that we can later perform a keyword search. Here is one way to do it.[2] First, we encrypt all the files using AES under a standard mode of operation. Throughout the rest of this discussion we will assume the AES-encrypted files are now stored on an untrusted server and we will not illustrate them in our diagrams. The key used to encrypt the files is never revealed to this server.

We now proceed to encrypt the multi-map. To do this we also make use of AES and of a keyed hash function like HMAC-SHA256. We first generate two 256-bit keys K_T and K_R for HMAC-SHA256 and AES, respectively. We then create a lookup table T and an

---

[2] The approach described is a simplification of a solution first proposed by Curtmola, Garay, Kamara and Ostrovsky in the paper “Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions”, ACM Conference on Computer and Communications Security (CCS ’06), 2006.

array A. We start by taking each node of each list in the multi-map and storing it at a random location in the array A. We then update the pointers in all the nodes so as to preserve the integrity of the lists. For our concrete example, this results in the following:

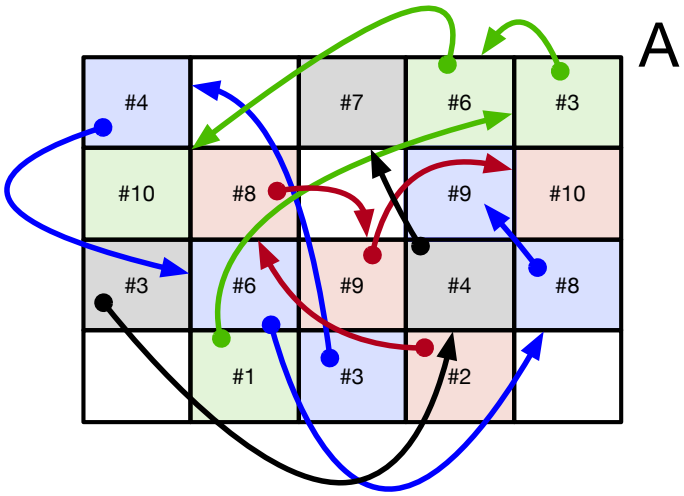


We now encrypt each cell in A as follows. If the cell stores a node from "data"'s list then we encrypt is as $\left((\#N \| \text{ptr}) \oplus \text{AES}_{K_D}(r_i), r_i\right)$, where $\#N$ is the file identifier stored in the cell, $\|$ denotes concatenation, ptr is the pointer to the next node, $\oplus$ is the exclusive-OR operation, $K_D = \text{HMAC}_{K_R}(\text{crypto})$, i is the cell location, and r_i is a random 128-bit string. Notice that we "encrypt" the string $\#N \| \text{ptr}$ as $(\#N \| \text{ptr}) \oplus \text{AES}_{K_D}(r_i)$ and not as $\text{AES}_{K_D}(\#N \| \text{ptr})$ which is what one might expect. This is important for security but the reasons are very technical.

As a concrete example, the first node of A will now store the pair of values:

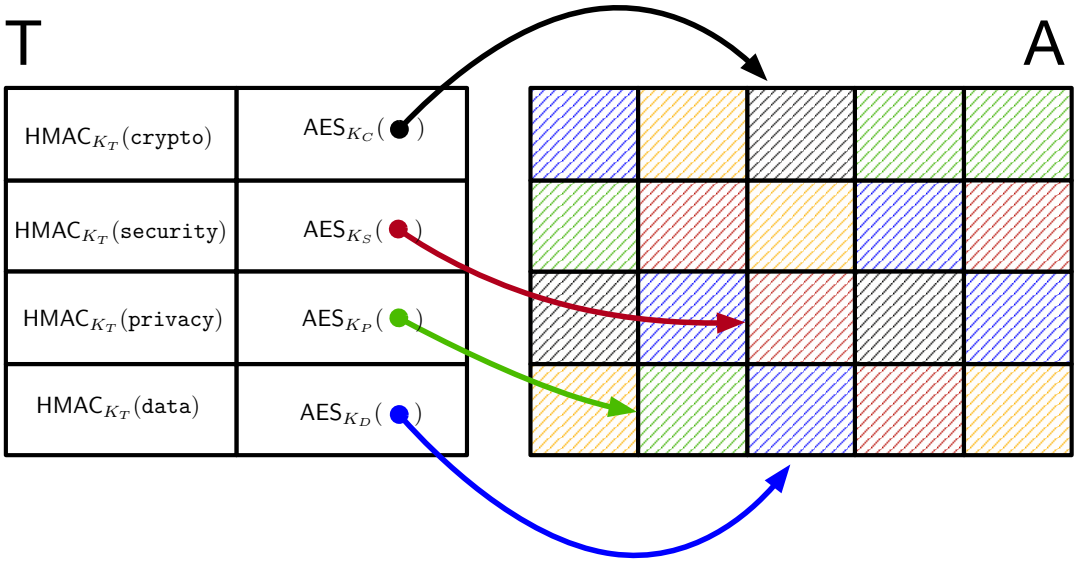
$$\left((\#4 \| 11) \oplus \text{AES}_{K_D}(r_0), r_0\right)$$

We do the same for the cells that store the nodes of "privacy", "security" and "crypto". If the cell is blank we store a random string. This results in an encrypted array that looks like:

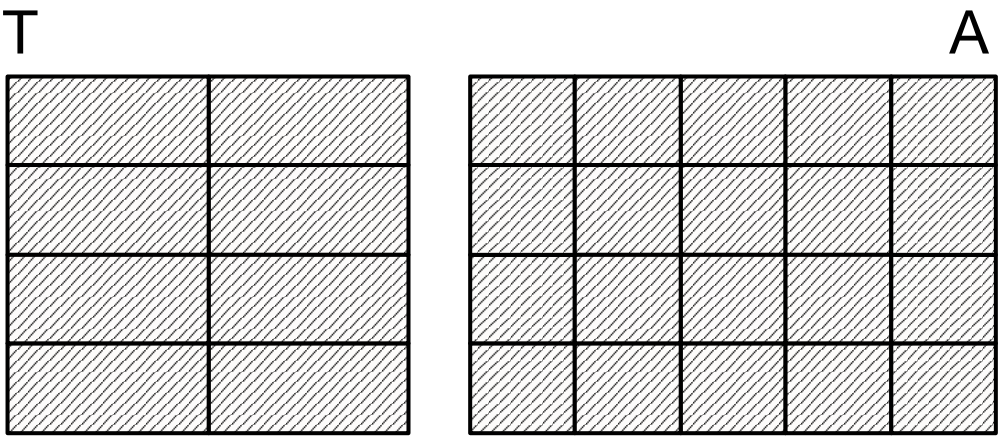


Here, the cross-hatch pattern represents the fact that a cell is encrypted. The yellow cross-hatch means that the cell stores random strings which, effectively, look like AES encryptions.

We now create a table T that will store encryptions of pointers (to the head of lists) under lookup values that are outputs of HMAC. For example, for the keyword "crypto" we store $\mathsf{AES}_{K_C}(2)$ under lookup value $\mathsf{HMAC}_{K_T}(\mathrm{crypto})$. Notice that 2 here is the location of the head of "crypto"'s list. We do this for all keywords including "privacy", "security" and "data". We now have something that looks like this:



Notice that T and A are effectively an encrypted version of our original multi-map, what we call an encrypted multi-map. To someone who does not know the keys K_T and K_R, the encrypted multi-map looks like this:



We can now safely store this encrypted multi-map together with the AES-encrypted files on any untrusted server. To perform keyword search, we need to give the untrusted server the ability to query the encrypted multi-map so that it can figure out the identifiers of the AES-encrypted files it needs to return. If we can do that, then we have effectively performed a keyword search over encrypted data.

We will proceed as follows. Recall that the encrypted multi-map and the AES-encrypted file collection are both stored on the untrusted server. Suppose we wish to query for the keyword "data". Instead of sending "data" in plaintext to the untrusted server, we send it $\mathsf{HMAC}_{K_T}(\text{data})$ and the key K_D. Notice that we can recompute the key K_D easily by computing $K_D = \mathsf{HMAC}_{K_R}(\text{data})$. We call these two pieces of information a query token (not to be confused with the practice of "tokenization" which attempts to protect private information by replacing sensitive data with non-sensitive substitutes). This query token will give the untrusted server the ability to query the encrypted multi-map without knowing what the underlying keyword is.

Given the query token, the untrusted server can use $\mathsf{HMAC}_{K_T}(\text{data})$ as a lookup value in T to recover the encryption $\mathsf{AES}_{K_D}(17)$ which it can decrypt using K_D. This decryption gives it the pointer 17 which it follows to recover the contents of cell 17 which is $\left((\#3\|0) \oplus \mathsf{AES}_{K_D}(r_{17}), r_{17}\right)$.

To decrypt this, the untrusted server then computes $(\#3\|0) \oplus \mathsf{AES}_{K_D}(r_{17}) \oplus \mathsf{AES}_{K_D}(r_{17})$ which is equal to $(\#3\|0)$. It then follows the pointer 0 and so on. At the end of this process, the untrusted server has recovered the list $(\#3,\#4,\#6,\#8,\#9)$ and can send back the AES-encrypted files identified in that list.

There are a few things to notice about this process. The first is that the untrusted server never decrypts the AES-encrypted files. It does not have the key used to encrypt these files. But there is no need to as it can effectively do the search by operating entirely on the encrypted multi-map. Here, when we perform a query, the untrusted server learns: (1) the file identifiers so that it can return the appropriate AES-encrypted files [3]; and (2) whether a query is repeated but not what the query was [4].

Notice also that if the server is trusted but potentially vulnerable to a disk-level or even memory-level breach, the attacker will learn nothing at all, not even the file identifiers or whether a query was repeated or not.

A third thing to notice is that the query process on the untrusted server is very fast. In fact, it is almost as fast as querying an unencrypted multi-map. The only additional cost on the server is the cost of decrypting the value that is looked up in T and the encryptions and exclusive-OR operations needed to "decrypt" the nodes in the lists.

---

[3] There are more sophisticated solutions that can even hide that.

[4] Again there are more sophisticated solutions to hide this as well.

We stress again, that the solution we described here is not what we use in our products and is provided only to illustrate that secure query operations over encrypted data — that is to say, end-to-end database encryption — is possible. There are many security, efficiency and functionality properties that this solution does not achieve.

Contacts

Web: https://aroki.com
email: info@aroki.com