Improving the SecureDNA System

Max Langenkamp, Andrea Lin, Alex Quach, Grace Hu

Abstract

One of the greatest threats to human civilization today is an engineered pandemic. This risk has existed for many decades, but is rising with advances (and the lack of protective measures) in DNA synthesis. SecureDNA is a system that provides protective measures against such a scenario by privately and screening DNA orders against a database of known dangerous sequences. In this paper, we begin by surveying SecureDNA, a system that allows secure and private screening of DNA orders. After explicitly describing its threat model, we propose two efficient extensions to the SecureDNA system: key-holder authentication and database proof of membership. We finish by considering three promising features to further strengthen the security of SecureDNA.

1. Introduction

In March of 1995, the doomsday cult Aum Shinrikyu released several canisters of a nerve agent in subway stations surrounding the Japanese parliament [4]. Fortunately, their plan failed: less than a hundred people had persistent damage from the bioweapon and the perpetrators were caught. However, subsquent investigations revealed that a scientist with a graduate degree in biology had been part of Aum Shinrikyu had tried to get ahold of infectious viruses such as Ebola, which have far greater capacity for harm than nerve agents. Because of advances in rapid DNA synthesis, it is now faster and cheaper than ever before to acquire dangerous sequences of DNA. The risks to human civilization from bioterrorist attacks has never been higher.

SecureDNA is a cryptographic system which aims to address this concern by providing platform for DNA synthesis companies to privately and securely screen DNA orders [1]. Once a customer (e.g., a biology lab) places a sequence order with the client (i.e., a DNA sequencing company), the client interacts with key-holding servers to form an encrypted query that then gets checked against a database of encrypted queries. This particular design meets several constraints around performance, privacy, and resistance to a range of attack vectors.

The contributions of this paper are fourfold. First, we provide a succinct explanation of the SecureDNA system and its threat model. Second, we detail two modifications to significantly strengthen the threat model: key-holder authentication and database proof of membership. Third, we provide performance analysis to argue that the both of our solutions require low overhead. Finally, we explore three more speculative extensions which we believe will further strengthen SecureDNA.

2. SecureDNA Overview

There are 5 main parties involved in SecureDNA:

- *Customer*: wants to synthesize DNA. Sends DNA sequence order to the client.
- *Client*: queries database with encrypted DNA order. Synthesizes DNA for customer if order approved.
- *Database Server*: tells client whether a given encrypted order is dangerous. Contained within a distributed network of cloud servers that store hashed DNA sequences.
- *Key-holders*: receive encrypted DNA order from client, further transform order, and send it back.
- *Administrator*: responsible for adding sequences to the database.

The system is subject to three particular constraints:

1. Database queries must be rate-limited. Adversaries cannot determine database contents by repeatedly querying.
2. Database contents must be kept private and cannot be recoverable. A computationally bounded adversary should not be able to recover the sequences in the database.
3. Customer query must be kept private. Guaranteeing privacy of customers is necessary for widespread adoption.

SecureDNA processes DNA synthesis orders by breaking a requested sequence into 40-50bp fragments and then comparing the fragments to a database of known dangerous fragments. If any sequence from the order matches, then the order will not be synthesized. Specifically, SecureDNA employs the following basic screening protocol:

1. Customer sends client a request to synthesize sequence x.
2. Client chooses a random β and creates $H(x)^\beta$ before sending it to t/n total key-holders. Each individual key-holder creates $H(x)^{\beta\alpha_i}$ and sends $(H(x)^{\beta\alpha_i}, \lambda_i)$ back to the client.
3. The client does a calculation to get the hash of x:

$$f_a(x) = (\prod_{i=1}^{t} H(x)^{\beta\alpha_i\lambda_i})^d \tag{1}$$

   where $d = \beta^{-1} mod(q)$
4. The client sends $f_\alpha(x)$ to the server, which checks if x is in the database using binary search.
5. Server responds with 0 or 1 depending on whether the sequence is stored in the database.

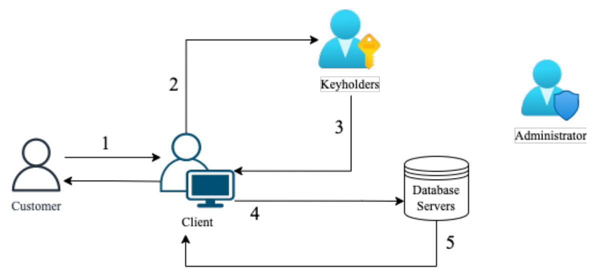

Figure 1. Basic Screening Protocol Overview

For simplicity, we are focusing on the protocol for screening DNA, rather than the initalization protocol. For a more detailed overview, we point the reader to the original SecureDNA paper [1].

The proposed SecureDNA system successfully addresses each of the constraints. The existence of key-holders allow for queries to be rate limited. Storing the distributed oblivious pseudorandom function (DOPRF) hashes ensures that the sequences are computationally indistinguishable given the Decisional Diffie-Hellman and random oracle assumptions. Finally, the DOPRF also ensures customer privacy is maintained because sequences are concealed even before being sent to key-holders.

In the next section, we will explicitly discuss the SecureDNA threat model, and how it can be extended.

3. Extending the SecureDNA Threat Model

The current SecureDNA threat model assumes that key-holders, clients, and database servers will always provide honest responses. However, as the system grows in scale, the likelihood of a compromised party also increases.

There are two attacks we are concerned about outside of the current threat model: Key-holder response corruption

Parties	**Current System** *SecureDNA*	**New System**	**New Attack Considered**
Database server	🟩	🟥	Database Connection Hijacking (false replies)
Key holders	🟩	🟥	Keyholder Server Corruption (fake keys)
Client	🟩	🟩	N/A
Customer	🟥	🟥	N/A
Administrator	🟩	🟩	N/A

Figure 2. Differences in an adversary's capabilities in current vs new threat model. Green indicates the adversary is always trusted, yellow and red means the player may be malicious/corrupt.

and Database connection hijacking. We will elaborate on each of these in their respective sections.

In the next section, we propose two changes to relax the threat model. key-holder authentication removes the need to trust key-holders, database membership proofs allow the client to verify database responses. In the last section, we will explore more speculative approaches to further enhance the security of our system.

3.1. Threat scenarios

Database Connection Hijacking: The database provides the answer for whether a DNA fragment is dangerous or not by checking for set membership. If the client's query is in the database, then the DNA synthesis order is dangerous. If the client's query is not in the database, then the DNA synthesis order is safe. However, a sophisticated man-in-the-middle attack may be able to modify the response to one or more of the distributed databases. The adversary can then return an arbitrary response, potentially allowing the synthesis of a dangerous sequence.

Key-holder Server Corruption: The key-holders must provide the correct keys for the secret sharing to work to generate the hash of the client's query. However, the operations of one of the key-holders may have been corrupted. Since there is no checksum in place, however, there is no way for any parties to know that the query encryption has been corrupted. In this case, the database would almost certainly give permission for synthesis. In this way, an adversary could also corrupt a key-holder server when they want to synthesize a dangerous sequence.

4. SecureDNA Extensions

4.1. Database Membership Proof

The current SecureDNA model relies upon trusting the database's answer without proof, which fails if the database is hijacked. We can remove this assumption by requiring the database to provide a proof of membership/non-membership alongside each query.

There are two reasons for requiring the database to provide proof. First, the DNA may be compromised by an adversary. Second, the customer may demand proof that they sequence is dangerous. Our system proposes a method for the database to provide proof that it is telling the truth without revealing information hazards. That is, the database should be able to provide proof of set membership or set non-membership without revealing the entire database.

4.1.1 Protocol

We reduce the problem to authenticating a dictionary. Each dangerous sequence is a key in the dictionary, and the value is the sequence itself.

The data structure we propose for this is a Prefix-Merkle trie (PMT). This is also known as a Patricia-Merkle trie. Figure 3 provides a concrete example of such a trie.

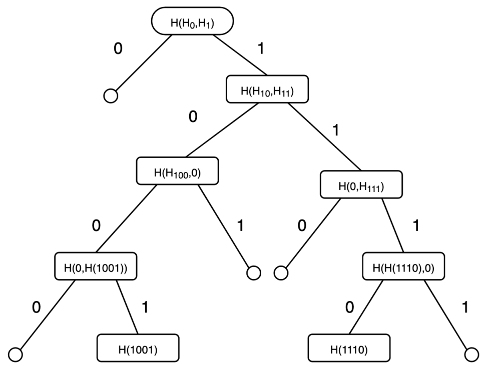

Figure 3. Prefix-Merkle Trie example

Setup. The database adds each dangerous sequence to the dictionary as a key, and its corresponding value is also the sequence. The database then constructs a Patricia-Merkle trie from this dictionary. Each key in the dictionary is converted to a binary number, and the hash of the value is stored at a leaf node. Each digit in the binary number corresponds to which branch the element takes. 0 corresponds to taking the left path, and 1 corresponds to taking the right path. Now, every parent node is the hash of the concatenation of its two children. If a parent node only has one child, the parent is a concatenation of its single child node and 0. Finally, the database publishes the root hash.

Membership proofs. If the client provides a sequence x and the database sees that x is a dangerous sequence, then the database provides a membership proof. This membership proof is the audit path for x in the Patricia-Merkle trie. The audit path lists the sibling to each node on the path from the leaf node corresponding to x up to but not including the root node.

To verify the proof, the client checks that hashing the concatenation of $H(x)$ and its sibling, and then the hash of that and its sibling, all the way up to the children of the root node, is equal to the root hash published by the database.

For example, for the database in Figure 3, to prove that 1110 is in the trie, the database provides the following audit path: (L/R = Left/Right node) R:0,L:0,L:H(H(0,1),0),L:0.

Non-membership proofs. If the client provides a sequence x and the database checks that x is safe, then the database provides a non-membership proof. This non-membership proof is also the audit path for x in the Patricia-Merkle tree, where x's leaf node may hold the value 0, or x is not in the tree at all.

In the first case, where x's leaf node stores 0, then the database provides the same proof for x as in the non-membership proof. To verify, the client checks that hashing the concatenation of 0 and x's sibling, and then the hash of that and its sibling, all the way up to the children of the root node, is equal to the root hash published by the database.

In the second case, if x is not a key in the trie, then some prefix of x is a key in the tree and the prefix's leaf node stores 0. Then, the database provides the audit path for the prefix, and the client verifies that the prefix is indeed a prefix of x, and then performs the same verification of the audit path for the prefix.

Next we will argue that the PMT provides membership proofs without significant costs to performance.

4.1.2 Performance Analysis

Setup. The setup has to happen only once and is thus a constant cost.

Membership proofs. Assuming that the hash function maps values to 256-bit numbers, since the tree is 256 layers deep, proof of membership is constant time. In the case of every query, value of 256 neighboring nodes will need to be provided. Since these are calculated in advance during the setup, proof of membership uses constant time.

Non-membership proofs. Proof of non-membership is also constant time. At worst case, the proof requires providing 256 neighboring nodes, but for many cases, the proof may be shorter if a prefix of the query has value 0 in the trie.

4.2. Key-holder Verification

We can remove the need to trust key-holders by verifying their outputs. SecureDNA estimates the need to handle 10^{15} transactions per year by 2029 [1]. Many key-holder servers are required to make sure the SecureDNA system can continue to operate in the face of such large demand. However, with a growing number of servers, there is increased risk of adversarial or corrupt key holders. Naturally, our system would want to increase the number of total key-holders, n. To insure a consistent level of security, it would make sense also to increase the number of key-holders, t, required to perform the DOPRF. We suggest a proportional scheme, where t should be a certain percent of the number of the n total shares, since the security parameter is dependent on compromising t key-holders.

However, since the (t, n)-DOPRF requires all t key-holders to provide valid values for the DOPRF to be successful, any single compromised key-holder can impede the verification process for a transaction. Compromised key-holder servers can be either corrupt or adversarial. This can be used to allow the synthesis of dangerous sequences by leading them to be falsely considered as safe. This problem of adversarial or accidental failure and bottlenecking the network becomes a larger problem with scale.

4.2.1 Protocol

We propose a scheme where the database server checks the key-holders.

Setup of Database

(a) Since we need a positive hit within the database to ensure that the DOPRF function is working correctly, we need to input a dummy entry into the database server. This can be during the initial setup of the database server (or at some point thereafter, when we can be assured that the DOPRF is correct). Let us call the message and DOPRF hash of the message (that is input into the database server) x_{dummy} and $H_{DOPRF}(x_{dummy})$, respectively.

Initialization of Round

(b) Whenever keys are generated or refreshed, the database server samples random β.

(c) The database server sends $H(x_{dummy})^{\beta}$ to each of the key-holders, where H is a hash function.

(d) The database server receives $H(x_{dummy})^{\beta\alpha_i}$ from each key-holder i. It also stores each $H(x_{dummy})^{\beta\alpha_i}$ for this key period.

(e) The database server confirms that the completed $DOPRF(x_{dummy})$ matches an entry in the database by calculating Equation 2. If not, the server encounters an invalid hash, and jumps to (g).

Recurring Round

(f) The database continually resends $H(x_{dummy})^{\beta}$ to each of the key-holders and checks the result against the stored $H(x_{dummy})^{\beta\alpha_i}$ to ensure that the returned values are consistent. If a value is not consistent, the server jumps to (g).

Encountering an Invalid Hash

(g) If an invalid DOPRF is computed in the Initialization of the Round (e) or during the Recurring Round (f), the first step is to identify the compromised server. If the invalid hash is encountered during the recurring step, then we can determine the compromised server is based on which $H(x_{dummy})^{\beta\alpha_i}$ response doesn't match the cached value. If the invalid hash is encountered during the initialization of the round, then it is unclear which server is compromised. Under the assumption that at most one server may be compromised at a time, we can use Algorithm 1 over the set of t servers in the detected invalid DOPRF operation.

Algorithm 1 Compromised Server Binary Search

```
procedure CompSearch
    $sus\_arr \leftarrow$ array of servers requested in invalid hash
    $valid\_arr \leftarrow$ array of all other servers(presumed valid)
    $low \leftarrow 0$
    $high \leftarrow$ length of $sus\_arr - 1$

    base case:
    if $low \geq high$ then
        return sus_arr[low]

    $mid \leftarrow low + (high - low)/2$.
    Swap sus_arr[low: mid] with valid_arr[0: mid-low]
    if DOPRF(sus_arr) is Valid then
        return CompSearch(valid_arr[0: mid-low])
    else
        return CompSearch(sus_arr[mid: high])
```

(h) Once the compromised serer is detected, the protocol refreshes the keys for all servers except the compromised server.

4.2.2 Tunable Configurations

The choice of t, the number of servers required for a DOPRF calculation and n, the total number of servers are values that will need to change over time with availability and scalability requirements.

There are additional choices regarding timing: how often keys are rotated and how often the recurring round (f) is performed. How often keys are rotated provides tunability to the period of time in which an adversary would need to compromise t servers. While the periodicity of the recurring round enables the worst-case detection of a corrupt server.

One additional option is that in Section 4.2.1, the database may instead use a different $\tilde{\beta}$ in $H(x)^{\tilde{\beta}}$ to all key-holders during Step (f) of the Protocol. This would require an extra aggregation of the $H(x)^{\tilde{\beta}\alpha_i}$ to check against the hash of the message, but this adjustment would protect against potential adversarial keyservers that may detect similar queries. Additionally, this would complicate determining which server is compromised, and would require the use of Algorithm 1 if an invalid hash is detected.

4.2.3 Performance

Setup of Database. The setup has to happen only once and only requires a single insert into the database. Therefore, it has a constant cost.

Initialization of Round. The intention of the Round Initialization step of the protocol is to check that each key-holder server in the distributed system is functional. Thus, we expect to perform the initialization such that every key-holder server is initialized in at least 1 group. This equates to around $\lceil \frac{n}{t} \rceil$ DOPRF calculations, or $O(n)$ messages between the database server and individual key-holder servers.

Recurring Round. The Recurring Round is similar computationally to the Initialization of the Round. If the system is configured to use different β values during the Recurring Round as outlined in Section 4.2.2, then the calculation is equivalent to a single DOPRF calculation per group of t key-holder servers. This equates to around $\lceil \frac{n}{t} \rceil$ DOPRF calculations, or $O(n)$ messages between the database server and individual key-holder servers. Otherwise, if the system uses the same β values during the Recurring Round, then we perform 1 exponentiation less by omitting the calculation in Equation 2.

Recurring Round. As outlined in the SecureDNA system, the Key Share Refreshing Protocol requires every key-holder to communicate with the database server to get their new keys. This requires $O(n)$ messages between the database server and individual key-holder servers.

4.2.4 Further Considerations

The algorithm to detect a compromised server without any valid prior hashes operates under the assumption that only 1 server may be compromised at a time. This assumption may hold in most cases, but an adversary that coordinates the takeover of multiple servers may violate this assumption. There is more work to address this generalized case in an efficient manner. We may draw inspiration from varients of the counterfeit coin weighing problem [2] to detect multiple elements (servers) that are compromised.

The adversary could also try to detect/estimate which queries come from the database server and which queries come from a client. Some further work considering this potential attack may include adding randomization of timing and/or IP address redirection to ensure that the adversary could not differentiate the database server from clients.

Another direction for progress is creating a mechanism to prevent a single adversary from triggering repeated key refreshes. In the current proposed system, when a compromised server is detected, all keys need to be refreshed to ignore the compromised server. Since any subset of t of the n key-holders can constitute a valid DOPRF, it is difficult to selectively deactivate a single compromised server.

5. Future Directions

In addition to our two technical proposals to improve SecureDNA, we have a number of promising extensions.

Customer-client non-repudiation. In the event of a malicious order, we want the the client to be able to prove that the customer did indeed place the order. To do this, we believe a couple of alterations are in order. First is making sure that the customer signs every order to the client. This allows an external party to verify that they did in fact place the order. Next, in the event that the client receives notification that the order was in the database, we will use a zero knowledge circuit to generate the proof. The entire DOPRF will be implemented within a zero knowledge proof, and the client will then generate a zero knowledge proof that the given query is in the database. This proof allows for two things. First, the client cannot falsely accuse customers of malicious orders. Second, the client does not need to reveal the dangerous order itself in order to provide proof that the customer placed a dangerous order. This minimizes the chance for dangerous sequences to be publicly leaked.

Pseudorandom lying. Although the database stores the hashes of a pseudorandom functions whose parameters are kept private, we could imagine a scenario where the key servers and the client are colluding to brute force request as many sequences as possible. While we cannot prevent the querying, we could make it harder for the adversary to guess the sequences in the database by allowing the database to lie with some fixed probability. Specifically, if, for a given true positive query, the database lies with probability p, then the adversary will require $1/p$ more queries to find out the same information. Given that some number of false negative matches in a database is acceptable (since multiple of these sequences are necessary to create a dangerous virus), this would provide yet another barrier against the existing adversaries.

Proof of database consistency. We have provided a means for the database to prove membership or non-membership. However, in the event that the administrator wants to add a new sequence to the database, it would be good to provide a proof that the database is updated in a consistent way. This can be done using using transparency logs (See, e.g., Hu et al. (2021) [3]). This is equivalent to reducing the need to trust the administrator.

References

[1] Carsten Baum, Hongrui Cui, Ivan Damgård, Kevin Esvelt, Mingyu Gao, Dana Gretton, Omer Paneth, Ron Rivest, Vinod Vaikuntanathan, Daniel Wichs, et al. Cryptographic aspects of dna screening. 2020. 1, 2, 4

[2] Richard K. Guy and Richard J. Nowakowski. Coin-weighing problems. *The American Mathematical Monthly*, 102(2):164–167, 1995. 5

[3] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle 2: A low-latency transparency log system. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 285–303. IEEE, 2021. 6

[4] Amy E Smithson and Leslie-Anne Levy. Ataxia: the chemical and biological terrorism threat and the us response. 2000. 1