Building an Encrypted and Searchable Audit Log

Waters, Balfanz, Durfee & Smetters

Presenter: Matthew Green
Talk Outline

- Searches on Encrypted Data: Background & Previous Work
- Secure Audit Logs, The Scheme
- Extensions and Recent Work
- Implementation: Where is it?
- Open Problems
Searching Encrypted Data?

- Search ciphertexts based on contents
- Maintain confidentiality, allow searchers to detect certain elements, e.g. keyword
- Notions of security, Dictionary attacks?

\[ E_k(\text{“3100 Wyman Park Drive, Baltimore”}) \]
Delegated Searching

Contact the Keyholder for authorization to search on a particular term

Let me search for “Water”? 

Searcher  Authorization  Keyholder  Secret Keys
Delegating: Motivation

Motivation is twofold:

- Efficiency: keyholder can offload search workloads to somebody else, reduce bandwidth
- Reduce size of Trusted Computing Base

Keyholder
Trusted Computing Base

- **DB**
- **DB**
- **DB**
- **DB**
- **Creator**
- **Search Device**

= Fully Trusted
Reducing a Trusted Computing Base

- **DB** = Fully Trusted
- **DB** = Semi-Trusted

**Creator**

**Search Device**

**Keyholder (online?)**

**DB**

Symbols:
- Red diamond = Fully Trusted
- Blue diamond = Semi-Trusted
Schemes
Song, Wagner & Perrig

- Plaintext is divided into words, \( w_1 \ldots w_n \)
- Encrypted with a symmetric-key stream cipher

```
<table>
<thead>
<tr>
<th>&quot;now&quot;</th>
<th>&quot;is&quot;</th>
<th>&quot;the&quot;</th>
<th>&quot;time&quot;</th>
<th>&quot;for&quot;</th>
<th>&quot;all&quot;</th>
<th>...</th>
</tr>
</thead>
</table>

<keystream> + 
= 

\[ C_1 \quad C_2 \quad C_3 \quad C_4 \quad C_5 \quad C_6 \quad C_7 \]
Search delegation: keyholder reveals $k$, to allow tests on $<S_i, f_k(S_i)>$
Secure Indexes (Goh)

- Goh introduces IND-CKA, IND2-CKA model for ciphertexts
- IND-CKA: A ciphertext reveals no information unless you search for the precise keyword
- IND-CKA2: As above, reveals no information about the # of keywords
Audit Logs

- Record activity that takes place on a server/device.
  - Log attacks/unauthorized usage
  - Should be efficiently searchable by authorized users (e.g., searches by username or activity type)
Audit Log Attacks

- Attacker gains total control of machine and all of its secrets. There are three primary threats to the audit log:
  - Destruction (total or selective)
  - Modification, e.g. to cover attack trail
  - Examination, e.g. to recover usage data & other potentially useful information
Protecting Log Integrity

- Schneier & Kelsey: Cryptographic Protection for Audit Logs
- Ensures integrity & privacy of log entries written before compromise
- (can’t save entries written afterwards!)
$W_{j-1}$ \quad $E_{K_{j-1}}(D_{j-1})$ \quad $Y_{j-1}$ \quad $Z_{j-1}$

$W_j$ \quad $E_{K_j}(D_j)$ \quad $Y_j$ \quad $Z_j$

$A_j$

$K_j = H(W_j, A_j)$

$Y_j = H(Y_{j-1}, E_{K_j}(D_j), W_j)$

$A_{j+1} = H(A_j)$

$E_{K_j}(D_j)$

$MAC_{A_j}(Y_j)$
Integrity & Privacy

- S&K use a hash-chain to guarantee security/integrity of older log entries
- Forward Secure

\[ k_n = f(A_n) \]
\[ k m_n = f'(A_n) \]
Decryption requires the original secret (or some intermediate version)

Search requires full decryption

Must be absolutely sure $A_{n-1}$ is eradicated

\[
k_n = f(A_n)\]
\[
k_m n = f'(A_n)\]
We can limit which records a user can decrypt, by deriving keys based on public record types.

$$k_n = f(\text{Type}, A_n)$$
$$km_n = f'(\text{Type}, A_n)$$
Decrypting a Log

- Contact the Trusted Manager for a decryption key on any log entries you want.
- Specify entry types (or keys won't work).

Might I decrypt entries 40-800 of types {....}?

Auditor

\[ k_{40}, \ldots, k_{800} \]

Manager

Secret Keys
Schneier/Kelsey can provide time-based decryptions (or search)

\[ k_n = f(A_n) \]
\[ km_n = f'(A_n) \]
Identity Based Encryption

- First proposed by Shamir in 1984, actual schemes by Cox, then Boneh & Franklin
- Anyone can compute a Public Key from some public Info + a string
- PKG can generate a Secret Key from the string + some secret Info

```
PK = “mgreen@cs.jhu.edu”
    + PK_M

SK = “mgreen@cs.jhu.edu”
    + SK_M
```
Elliptic Curves

- **Based on Curve Points** (e.g., P, Q.)

- **Point Addition**, similar to integer multiplication:
  \[(P + Q) = (Q + P), \quad (Q + \langle{\text{unity}\rangle}) = Q\]

- **Scalar Multiplication**, similar to exponentiation:
  - e.g.: \[5 \times P = (P + P + P + P + P)\]
  - \[1 \times P = P\]
  - \[q \times P = P \text{ (where q is the order)}\]
Cryptographic Assumptions

- **Discrete Logarithm Problem:**
  Given $g^a \mod p$, find $a$

- **Computational Diffie-Hellman Problem:**
  Given $g^a$ & $g^b$, find $g^{ab} \mod p$
Elliptic Curve Assumptions

- **EC-Discrete Logarithm Problem:**
  Given $aP$, find $a$

- **EC-Computational Diffie-Hellman Problem:**
  Given $aP$ & $bP$, find $abP$
Bilinear Pairings

A Bilinear Pairing is a function $e(G_1, G_1) \rightarrow G_2$ with the following properties:

- **Non-degeneracy.** For generator points $<P, Q>$ in $G_1$, $e(P, Q)$ is a generator of $G_2$

- **Bilinearity.** $e(aP, bQ) = e(P, Q)^{ab}$

- **One Way.** No way to map back from $G_2$ to $G_1$
Pairings \neq \text{CDH}

\[
\begin{align*}
G_1 & \quad \text{e}(aP, bP) \\
G_2 & \quad \text{e}(P, P)^{ab}
\end{align*}
\]
Fun With Pairings

Public Key = sP

Hash_to_Point(“foobar”) = zP
Boneh & Franklin’s IBE

- A pairing $e(P, Q) \rightarrow \mathbb{Z}_q$
- Two hash functions: \( \text{Hash}_\text{to}_\text{Point}(), H() \)
- Public Parameters: (curve params, $p, q, P$)
- $SK_M = s$, $PK_M = sP$
B & F’s IBE Encryption

GET_PK(PK_M = sP, "<keystring>"):  
PK = e(Hash_to_Point("<keystring>", sP)  
    = e(zP, sP)  
    = e(P, P)^{sz}

GET_SK(SK_M = s, "<keystring>"):  
SK = s * Hash_to_Point("<keystring>")  
    = s * zP  
    = szP
B & F’s IBE Decryption

IBE_ENC(M, PK = e(P, P)^sz):
  r = random int from Z_q
  C = <rP, M XOR H(PK^r)>

IBE_DEC(C, SK = szP):
  e(rP, szP) = e(P, P)^szr
  Hash e(P, P)^szr, then XOR to recover M
Boneh, Crescenzo, Ostrovsky & Persiano

- Same scheme as Waters (independently discovered)
- Provides a real security model
Creating a Log Entry

\[ E_K("\text{mgreen searched for ... \text{\textquote{Gas}}, \text{\textquote{Electricity}}, \text{\textquote{Water}} ... \text{\textquote{}}") \]

\[ \text{IBE-ENC}(\text{PK("Gas"), <flag | K>}) \]

\[ \text{IBE-ENC}(\text{PK("Electricity"), <flag | K>}) \]

\[ \text{IBE-ENC}(\text{PK("Water"), <flag | K>}) \]

\[ E_{PK}(K), \ H(\text{this record } \| \ H(\text{last record})) \]
Searching, Step 1

Contact the Trusted Manager for a search key on a particular term

Let me search for “Water”?

Searcher

SK("Water")

Manager

SK_{M}
### Searching, Step 2

| IBE-ENC(PK("Gas"), <flag | K>) | IBE-ENC(PK("Electricity"), <flag | K>) | IBE-ENC(PK("Water"), <flag | K>) | E<sub>K</sub>("mgreen searched for ... 'Gas', 'Electricity', 'Water' ... ") |
|---------------------------------|----------------------------------------|---------------------------------|----------------------------------|

**IBE_DEC** SK("Water") →

**IBE-ENC** (PK("Gas"), <flag | K>) →

**IBE-ENC** (PK("Electricity"), <flag | K>) →

**IBE-ENC** (PK("Water"), <flag | K>) →

**EPK(K)** ...
Adding Time

Simple approach: append a Time period to IBE keystrings, e.g.:

\[\text{IBE-ENC(PK(“Gas || 9-14-04”), <flag | K>)}\]

Searcher indicates time period when requesting IBE Secret Key

Must still try all records
Caching IBE Public Keys

- To produce an IBE ciphertext, we generate an IBE Public Key.

- Key Gen is the most expensive operation, requiring up to 175ms (that’s per keyword!)

- To save time, we could cache these keys for later reuse

- The downside: If an adversary captures this cache, they learn which keywords have been active recently
Batching Keywords

- $n \times m$ Keyword Ciphertexts
  - $n =$ total log entries
  - $m =$ average # of keywords per entry

- Log generation & Search time proportional

- Many common keywords will be repeated, can we be more efficient than?
Does Batching Help?

- Batching reduces the number of ciphertexts from \((m)n\) to \(t\), where \(t\) is total # of unique keywords in the block.
- Batching reduces waste for the most common keywords, but what about the uncommon ones?
- Who searches on common words, anyway?
## Block Batching Example

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Entries</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;water&quot;</td>
<td>1, 2, 4</td>
<td>$k_1, k_2, k_4, k_{19}$</td>
</tr>
<tr>
<td>&quot;gas&quot;</td>
<td>14, 20, 27</td>
<td>$k_{14}, k_{20}, k_{27}$</td>
</tr>
<tr>
<td>&quot;electricity&quot;</td>
<td>3, 49</td>
<td>$k_3, k_{49}$</td>
</tr>
<tr>
<td>&quot;snorkles&quot;</td>
<td>24</td>
<td>$k_{24}$</td>
</tr>
<tr>
<td>&quot;petunia&quot;</td>
<td>4</td>
<td>$k_4$</td>
</tr>
<tr>
<td>&quot;spork&quot;</td>
<td>33</td>
<td>$k_{33}$</td>
</tr>
</tbody>
</table>
Uses “backpointers” to link groups of keywords within a time period

Advantages of batching, but doesn’t keep the log open (unwritten) for long periods
Randomness Re-use

To search a block of n keywords requires n pairing computations

\[ C = \langle rP, M \ XOR \ h(e(P, P)^{szr}) \rangle \]

\[ e(rP, SK("keyword")) = e(P, P)^{szr} \]

We can reduce this if we re-use the same value r for each keyword in a batch
Randomness Re-use

We can use \( rP \) for a group of ciphertexts, and only store the second term:

\[
\begin{align*}
   c_1 &= <\text{flag} | k> \ XOR \ h(e(P, P)^{\text{rsz}}) \\
   c_2 &= <\text{flag} | k> \ XOR \ h(e(P, P)^{\text{rsz}'}) \\
   c_3 &= <\text{flag} | k> \ XOR \ h(e(P, P)^{\text{rsz}''})
\end{align*}
\]

Only one pairing, but still have to XOR with many ciphertexts
A Slightly Better Approach

PK("water") = e(sP, Hash_to_Point("water"))
= e(P, P)^sz

SK("water") = s * Hash_to_Point("water") = szP
Waters’ Implementation

- Waters et al. implemented the IBE-based scheme to log SQL queries (MySQL Proxy)
- Used Stanford IBE Library, 1024-bit supersingular curves (q=160); AES 128-bit
  2.8GHz Pentium IV
- Hash-chain integrity checking
Implementation:
Optimizations Used

- IBE Public Key Caching:
  PK generation + encryption = 180ms
  encryption only (cached key) = 5ms
  100MB Cache -> ~800,000 Public Keys

- Webster's Dictionary: 300,000 words

- Randomness Re-use
Implementation: Ok, and...?

- Implementation reveals the pairing computation time, encryption time-- and not much else.

- Is it practical? Where are your performance numbers and graphs? What data are you storing? Can we have the source code?
Open Problems

- Reducing storage & computational costs
- Better security models, reduced involvement of keyholder
- New approaches, or incremental improvements?
Other Problems

- In the Song scheme, all keywords in the document are searchable.

- In the Goh scheme (and many others), relevant keywords chosen by data creator.

- Subtler concerns: What if keywords are not chosen correctly? What if data creator is malicious?
END
Revoking Search Keys

- We might want to revoke a search key after we’ve given it out.

- A possible approach:
  - Re-encrypt all keywords under new IBE keys.
  - e.g.: “Gas” -> “Gas || 2”
Revoking through Dumb Re-encryption

Keyholder

(Decrypt)

plaintext

(Encrypt)

DB

IBE-ENC(PK("Gas"), ...)

IBE-ENC(PK("Gas||2"), ...)

IBE-ENC(PK("Gas"), ...)

IBE-ENC(PK("Gas||2"), ...)
Revoking through Proxy
Re-encryption?

Keyholder \(\rightarrow\) RK \(\rightarrow\) DB

\[
\text{IBE-ENC(PK("Gas"), ...)} \\
\downarrow \\
(\text{Re-Encrypt}) \\
\downarrow \\
\text{IBE-ENC(PK("Gas||2"), ...)}
\]
Trusted Computing Base

= Fully Trusted
Waters et al.
Symmetric-Key Scheme

\[ E_K(\text{"mgreen searched for … ‘Gas’, ‘Electricity’, ‘Water’ …"}) \]

\[ h_S(\text{"Gas"}) \text{ XOR } \langle \text{flag | K} \rangle \]

\[ h_S(\text{"Electricity"}) \text{ XOR } \langle \text{flag | K} \rangle \]

\[ h_S(\text{"Water"}) \text{ XOR } \langle \text{flag | K} \rangle \]

Secret Key = S
Waters et al. Symmetric-Key Scheme

$$E_K(\text{"mgreen searched for ... 'Gas', 'Electricity', 'Water' ... "}), r$$

$c_1 = h_{a_1}(r) \ XOR \ <\text{flag} \ | \ K>$

$c_2 = h_{a_2}(r) \ XOR \ <\text{flag} \ | \ K>$

$c_3 = h_{a_3}(r) \ XOR \ <\text{flag} \ | \ K>$

Master Secret = S

$a_1 = h_S(\text{"Gas")}$

$a_2 = h_S(\text{"Food")}$

$a_3 = h_S(\text{"Water")}$
Symmetric, Searching

Let me search for "Water"?

\[ a = h_s("Water") \]

\[ c_1 \ XOR \ a = "???" \]

\[ c_2 \ XOR \ a = \text{<flag | key>} \]

\[ c_3 \ XOR \ a = "???" \]
Reducing a Trusted Computing Base

Keyholder
Reducing a Trusted Computing Base

DB

DB

Keyholder

DB

DB

Search Device

SK("Water")