A Conjunction, Language, and System Facets for Private Packet Filtering

Michael Oehler, Dhananjay S. Phatak, and Alan T. Sherman
Cyber Defense Lab
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, Maryland
Email: {oehler1,phatak,sherman}@umbc.edu

ABSTRACT

Our contribution defines a conjunction operator for private stream searching, integrates this operator into a high level language, and describes the system facets that achieve a realization of private packet filtering. Private stream searching uses an encrypted filter to conceal search terms, processes a search without decrypting the filter, and saves encrypted results to an output buffer. Our conjunction operator is processed as a bitwise summation of hashed keyword values and as a reference into the filter. The operator thus broadens the search capability, and does not increase the complexity of the private search system. When integrated into the language, cyber defenders can filter packets using sensitive attack indicators, and gain situational awareness without revealing those sensitive indicators.

I INTRODUCTION

Cyber defenders engage in collaborative efforts that share threat data, intrusion details, and attack indicators. These indicators are then used within defensive systems to identify subsequent adversarial activity on a network [1]. Through these efforts significant benefits are derived and for all collaborative partners. These indicators are initially derived from analytic processes, and at great cost.

The advantage however, is lost when an adversary discovers the indicator. The adversary can affect subtle changes to the attack infrastructure, delivery, installation, and to the malware. The value of the indicator is rendered ineffective.

The defender is faced with a challenge. There is a need to share indicators without revealing those indicators. To achieve this, many indicators are only shared with trusted partners, and with handling restrictions, like "monitor, but do not block." There are procedural controls too. Security researchers are known to submit newly discovered vulnerabilities directly to the developer, and prior to public announcement. Additionally, corporate policies may prevent indicators from being shared with subsidiaries or with international partners.

Our contribution realizes the association between this challenge, and the capability provided by private stream searching. Specifically, we adapt the private search capability presented by Rafail Ostrovsky and William Skeith [2, 3], while retaining the advancements presented by Danezis and Diaz [4,5], and create a high level language for private packet filtering.

Using our language, the defender constructs a query consisting of sensitive indicators, encrypts the query, and transfers the encrypted query (a filter) to the partner. The partner performs a private search on a stream of packets, and returns encrypted packets. If a matching packet is discovered, the defender notifies the partner of the adversarial activity, and coordinates a response. In this collaborative environment, the defender maintains situational awareness, controls which attack indicators are revealed, and advises the partner of current threat activity without revealing every sensitive indicator [6].

We designed the language to be intuitive, readable, and for the needs of the cyber defender. There is more to our conflation of private search and private packet filtering. We designed a conjunction opera-

tor to broaden the private search capability [7]. The original private search system provides a result based on a logical disjunction; if the indicator (privately) matches "any sensitive indicator in the encrypted filter", then return the packet. Our conjunction returns packets when an indicator and another privately match. We have also analyzed the number of false positives that result from the tagging mechanism used during document recovery [8].

The conjunction, specification of a language, and analysis of private stream searching did not address the system facets that are critical to realizing an implementation of private packet filtering. Much was left unsaid: the elimination of the public dictionary, use of a hashed-index approach for filter entry lookup, a modified document tagging method, a document partitioning method, iterative document recovery, and packet re-assembly.

We provide a full discourse of experiences and insight to guide future implementers and researchers of private stream searching.

II PRIVATE STREAM SEARCH

We describe the salient features of private stream searching in terms of client, provider, document, and keywords. These are a generalization of terms that provide clarity and an illustration of private search. In our context, these terms are synonymous with the cyber defender, partner, packets, and attack indicators respectively.

Private stream searching is a system of cryptographic methods that preserves the confidentiality of the search criteria and results. The naïve solution would transfer an entire data set from an information provider to a client. Admittedly, this would conceal the queries, but would divulge the entire data set without cause. This is unrealistic. Ignoring bandwidth costs and a required client-side search, few would relinquish an entire information asset. Alternatively, if the search criteria could be kept secret, but knowledge of the retrieved documents was revealed, the structure of a query could be inferred. This is also unacceptable.

These concepts thus, establish the fundamental properties of a private search system and introduce the participants: the information provider gains no knowledge of the queries, the provider cannot infer information about the queries from the retrieved documents, and client access to the provider's data set is limited to results matched by legitimate queries [9].

Ostrovsky and Skeith created a clever private search system using (partial) homomorphic encryption to conceal the search criteria, perform a query on this concealed criteria, and return only relevant documents in an encrypted form [2] [3]. The client generates the query, the provider performs a search, and returns a result. The homomorphic property of the cryptosystem assured that no information would be revealed about the search, the results from the provider, and the client would only receive matched documents.

The client constructs a search by selecting a list of search terms from a public dictionary. The client then creates a list of encrypted ones and zeroes that correspond to the keywords and non-relevant words respectively, and sends this filter to the provider. The provider initiates a search by computing a product of entries taken from the filter that associate with words found in a document, calculates an exponentiation, and calculates a second product to save the results to an encrypted output buffer. Documents are retrieved if any keyword existed in the document.

Ostrovsky's system fundamentally provides a private search capability based on a logical disjunction of search terms. Furthermore, multiple documents were stored in an encrypted output buffer, leading to the creation of a system that could stream results, a private stream search system.

The private search system is based on the asymmetric cryptosystem defined by Paillier, and utilizes the additive homomorphic property of the cryptosystem [10]. A brief summary of the cryptosystem is provided: If we denote Paillier encryption as a function from plaintext to ciphertext $E : \mathbb{Z}_N \to \mathbb{Z}_{N^2}$ and the decryption routine as $D : \mathbb{Z}_{N^2} \to \mathbb{Z}_N$, relative to a public and private key, and public modulus N, then the homomorphic property can be expressed as: $D(E(x) \times E(y)) = x + y$, for plaintext messages x and y. Also notice that a constant multiple of an encrypted value produces a scaled plaintext message, $D(\prod_k E(x)) = kx$, and clearly, $D(E(x)^k) = kx$ for some constant number, $k \in \mathbb{Z}_N$.

This relationship is used extensively in the formation of the private search system, but the association of values may be counter intuitive: A fixed value, either one or zero is encrypted for values of x, and document values are used for k, as explained below.

The Paillier cryptosystem is randomized. Thus, an

encrypted value will be indistinguishable from another, even for the same encrypted value, using the same public key. For instance, a particular encryption of $E(1)$ is indistinguishable from some other value of $E(1)$. Last, the Paillier cryptosystem is based on the problem of computing the n-th residue class, which is believed to be computationally difficult.

1 THE QUERY

In Ostrovsky's system, the client creates a query by selecting a public dictionary of words $D = \{w_1, w_2, w_3, \ldots\}$ and a set of private keywords $K \subseteq D$. The client then constructs an encrypted filter $F = \{f_1, f_2, f_3, \ldots, f_{|D|}\}$, where $f_i = E(1)$ for an associated keyword $w_i \in K$. Otherwise, $f_i = E(0)$. This encrypted filter establishes a one-to-one association between words of interest and all other words in a dictionary without exposing the private keywords. The client sends the encrypted filter and dictionary to the information provider.

2 THE SEARCH

The information provider creates an output buffer of tuples $B = \{\{E(0), E(0)\}, \{E(0), E(0)\}, \ldots\}$ that store the encrypted number of matching keywords in the first element, and the document in the second. The provider then performs the following steps for each document d that exists in a data source:

Extract a set of words W from d such that $W \subseteq D$. These are the search terms that determine if the document is a match.

Compute the encrypted value $s = \prod_{|W|} f_i = E(m)$ where every value of f_i (from F) associates with a corresponding search term $w_i \in D$. The value s is the encrypted match value (the client will decrypt this value to recover the number of distinct keywords present in the document $m = |W \cap K|$.) Note that an equal number of filter terms $|W|$ appear in the product of s and correspond to the number of search terms found in the doucment.

Append k bits to the document where each 3-bit triple has a Hamming weight of one, $d' = d << k$. The provider then computes the exponentiation $r = s^{d'}$. The encrypted search result r is either $E(m \times d')$, or $E(0)$ when no keywords occur in the document.

Note that a document d is a numeric representation of a textual document, network packet, database record, file, etc. For long documents, d may be partitioned. Last, a matching document d is *scaled* by the match value m. That is, $m \times d$.

The provider then saves the result $\{s, r\}$ to the output buffer. Specifically, the provider selects random buffer positions $B' \subseteq B$, and performs a (pairwise) modular multiplication $B' = B' \times \{s, r\}$. The new values for B' are then reassigned back to their associated positions in B. This multiplication step in the encrypted domain performs an addition in the plaintext domain; the provider is *adding* the result to the buffer.

When no keywords appear in a document, the result is $\{s, r\} = \{E(0), E(0)\}$. The pairwise multiplication of a non-matching document does not change the plaintext values of the buffer; non-matching documents are accumulated as a summation of plaintext zeroes. The buffer thus, optimizes the communication cost of the search, and assures that non-matching documents are not transmitted back to the client.

The provider returns the encrypted buffer to the client after searching all documents.

3 THE RESULT

To recover documents, the client decrypts the buffer, performs an integer division on each non-zero tuple in the buffer, and detects any surviving documents. Specifically, the client divides the recovered number of matching keywords by the scaled document value. In some instances, a tuple in a buffer position may contain a linear combination of multiple documents; the non-integer results of the division are discarded.

For instance, if a the linear combination at buffer position $b_i = \{m_i, m_i \times d'_i\}$ is divisible $m_i | (m_i \times d'_i)$, the client will attempt document detection. Otherwise, the collision of documents at b_i is discarded.

For divisible documents, the client initiates a document detection routine. In Ostrovsky's approach, the client detects a document when the Hamming weight of each appended 3-bit triple is one. Since multiple copies of a document are saved to the buffer, the client removes any duplicate documents, only one copy will be returned. Thereafter, the client halts processing. Notice that the client makes a single pass through the buffer. There is only one iteration of the document recovery routine in the original system.

Table 1: An Illustration of Private Stream Searching

Client: Generates the query:	
Define a public dictionary:	$D = \{gelato, sherbet, snowball, sorbet, zebra\}$
Select private keywords:	$K = \{sorbet\}$
Construct an encrypted filter:	$F = \{E(0), E(0), E(0), E(1), E(0)\}$
Send D and F to the provider.	
Provider: Performs the search:	
Receive the dictionary D and the filter F.	
Construct an encrypted output buffer of tuples:	$B = \{\{E(0), E(0)\}, \{E(0), E(0)\}\}$
Perform a Search for each document:	$d =$ "unlike gelato sorbet has no calories"
Extract dictionary words from document:	$W = \{$"$gelato''$, "$sorbet''\}$
Calculate a product of filter terms:	$s = \prod_{\|W\|} f_i = f_1 \times f_4 = E(0) \times E(1) = E(1)$
Append k-bits such that each $k/3$ triple has $wt(k/3) = 1$:	$d' = d << k$
Calculate the search result as the exponentiation:	$r = s^{d'} = E(1)^{d'} = E(d')$
Store the result to a randomly seclected buffer position:	$b_1 = \{s, r\} \times b_1$
	$= \{E(1), E(d')\} \times \{E(0), E(0)\}$
	$= \{E(1), E(d')\}$
Return the buffer to the client:	$B = \{\{E(0), E(0)\}, \{E(1), E(d')\}\}$
Client: Processes the result:	
Decrypt the buffer:	$B = D(\{\{E(0), E(0)\}, \{E(1), E(d')\}\})$
	$= \{\{0, 0\}, \{1, d'\}\}$
Identify the document and remove the k appended bits:	$d = d' >> k$
Recover the matching document:	$d =$ "unlike gelato sorbet has no calories"

4 ILLUSTRATION OF PRIVATE SEARCH

Table 1 illustrates a simplified example of the private search system. The client defines a public dictionary D with five words, and a filter F containing five encrypted values. The fourth entry is an encrypted one $E(1)$ and expresses a private keyword that associates with "sorbet".

The provider constructs the output buffer B, a list arranged as tuples of encrypted zeroes. The search entails a single document d. The provider calculates a product of filter entries, f_1 and f_4, corresponding to words that exist in the document and dictionary. The result of this product is the encrypted number of matching keywords $s = E(m)$ in the document. The provider appends k bits to d, and calculates an exponentiation $r = s^{d'} = E(m \times d')$.

Although the number of matching keywords is one in this example, a document d is scaled by the number of matching keywords in practice. The provider *randomly* selects a buffer position b_1 and saves the result $\{s, r\}$ to the buffer as a pairwise multiplication.

The client decrypts the buffer, and recovers the document. Our example saves one copy of the result for brevity.

III RELATED WORK

Ostrovsky's original design saves multiple copies of the result $\{s, r\}$ to randomly selected buffer positions. Intuitively, this random selection perpetuates the survival of at least one document copy (Ostrovsky formally presents this idea as the color survival

theorem.) However, and at a buffer's capacity, the majority of buffer positions are chosen multiple times.

This collision of documents eliminates surviving copies. The private search system thus, has a non-zero probability that some documents will not survive. Large buffers may minimize this condition, but this results in storage inefficiencies that are suboptimal.

Researchers recognized that a collision of multiple documents in a buffer position did not entirely destroy information. In fact, a collision produces a linear combination of documents. For example, if n documents are stored at a buffer position b (a buffer position is a tuple of two values), then $b = \{E(\sum_{i=0}^{n} m_i), E(\sum_{i=0}^{n}(m_i \times d_i)\}$ where the first element of the tuple is the summation of matching keyword values and the second is the summation of scaled document values.

Research has thus, qualified external structures, additional processes, and leveraged the redundancy of multiple copies to improve the document storage algorithm and thus, the number of documents recovered from an output buffer.

1 STORAGE AND RECOVERY

Bethencourt presents a private search scheme in which (encrypted) knowledge of a matching document's index is passed to the client [11, 12]. The client then uses this knowledge to extract matching documents by solving a system of linear equations, a strategy to improve the document storage and recovery rate.

Query construction is identical to Ostrovsky's system. The difference occurs during the search and retrieval phases. Namely, the scheme defines an additional buffer, known as the matching-indices buffer M, and a seeded pseudorandom function $g(i, j) \rightarrow \{0, 1\}$ where i is a document index and j refers to a buffer position.

The provider saves the encrypted match value $s = E(m_i)$ of the i-th document at positions designated by multiple hash values of the index i to buffer M. The provider then saves the result of the search $\{s_i, r_i\}$ to buffer positions in B as designated by the result of the function $g(i, j)$. If the search produces a non-matching result, an encrypted zero is added to both buffers; the plaintext contents of the buffer are unaltered.

To recover documents, the client decrypts both buffers and uses the matching-indices buffer as a Bloom filter to validate a document's membership in the output buffer. The client then uses the result of this membership test and the result of $g(i, j)$ to establish a set of linear equations to solve.

Danezis and Diaz introduced a (simpler) iterative method to improve the document recovery rate [4,5]. Their salient contribution noted that if knowledge of a document's position could be implicitly conveyed, then uncertainty could be removed from the decoding system.

Their resulting algorithm thus, defined a deterministic function for buffer position selection. Specifically, buffer positions were calculated as the hash of the summation of the document and a copy value:

$$positions = \{H(d_i + 1), H(d_i + 2), \ldots, H(d_i + l)$$

for each document d_i, and for a pre-determined number of inserted copies l.

Using these l positions, the provider saves a copy of the result $\{s_i, r_i\}$ to each buffer position via modular multiplication. Their algorithm still stores multiple copies, and relies on the underlying principle that some copies must survive to recover a document. However, this manipulation does not alter the underlying private search system, require transmission of an additional structure, and does not induce a significant computational cost.

To recover documents, the client iterates the following steps after decrypting the output buffer: Identify a document, calculate the positions that the document was stored too, subtract the document value from those buffer positions, and repeat until no further documents are discovered or the buffer is empty.

Due to the simplicity and improved recovery rate over that of the original approach, we use Danezis's iterative method in our prototype. A detailed example of this iterative method as used in our prototype is given in Section IV-5. Recovery rate and the optimal number of document copies is discussed as part of our simulation results in Section VI-1.

2 CONJUNCTION FILTERS

Ostrovsky and Skeith extended their private stream search system to form a conjunction from two lists of (private) keywords [2]. Their resulting "AND" operator returns a matching document if any keywords

from the first list and any from the second appear in the document. We can generalize this with the expression: $A \wedge B$ where A and B are lists of keywords. This extension uses the public-key cryptosystem defined by Boneh, Goh, and Nissim [13]; the cryptosystem is additively homomorphic, and supports at most one multiplication of plaintext values, using a bilinear map.

In this scenario, the client selects two lists of keywords, creates two encrypted filters from each list, and sends both to the provider. For each document, the provider calculates two products of filter terms where terms associate with the words in the document. The result of the search (products s_1 and s_2) are the encrypted number of matching keywords, and serve the same purpose as described by their original work. Ostrovsky then applies the bilinear map to these two ciphertext values $s = e(s_1, s_2)$. The result in the plaintext domain is a product of the number of matching keywords: a zero if no keywords occur in a list, or the product of keywords if matching words occur in both. The matching document is then saved through bitwise encryption.

Yi and Bertino also created a private search system with a conjunction based on the Boneh cryptosystem [14]. In their search, the provider returns a document when any conjunction of two keywords matches. We can generalize this with the expression of conjunctive terms: $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \ldots (a_n \wedge b_n)$ where subscripted a and b are distinct words. This expression provides a greater degree of specificity over Ostrovsky's conjunction, but incurs significant complexity to achieve.

The client computes a (sparse) matrix of filter values (encrypted ones and zeroes.) This *filter matrix* has a row for every conjunctive term, and a column for every word in the public dictionary $|F| = n \times |D|$. In each row, the client inserts an encrypted one corresponding to the words of the conjunctive term. A row will therefore, have two encrypted ones in it. The filter is sent to the provider.

For each dictionary word in a document, the provider will iterate over each row in the filter matrix (test each conjunctive term) by removing entries corresponding to words that are not in the document. The provider then calculates a "sum" of the remaining filter entries, and a "product" with the filter entry associated with the current and examined word. Since a row contains two encrypted ones, the result is either one (the sum of zeroes and a one, then multiplied by one), or zero when the conjunction fails. Of course, the calculations are done in the encrypted domain. The sum is a product of ciphertexts, and the multiplication is computed as a bilinear map. The process must repeat over each row and word, as a means of validating all conditions of the conjunctive expression.

Bethencourt indicates that a limited conjunction operator can be constructed from a hash of concatenated words [11]. Namely, the client would construct a filter containing a concatenation of private keywords, for example "$w_1||w_2$", hash this concatenation, and then store the encrypted value of one in the filter position indexed by the hash value. An information provider would then concatenate adjacent words from the document, hash these adjacencies, and proceed by extracting the filter entries associated with the hash values. Our method for conjunction also utilizes a hashed-index approach, but takes a different direction to form a conjunction operator. See Section V-1.

IV SYSTEM FACETS FOR PRIVATE SEARCHING

On the path to implementation, we discovered a variety of system facets to address. These included: the elimination of the public dictionary, a hashed-index, the development of a document tagging and detection method, a need to partition "large" documents (packets in our scenario), an integration of the iterative document recovery method, and packet re-assembly. Our solution consists of modifications to the original private search system, adaptation of prior research, and a utilization of aspects specific to our problem space, private packet filtering. Each of these facets is discussed in the subsections below.

We transition and use the word, *packets* and network *indicators* since this reflects our application. These terms are synonymous with *document* and private *keyword*.

1 ELIMINATION OF THE DICTIONARY

The original private search system exposes words in the public dictionary. Consider the example from Table 1. Although the encrypted filter precludes interest in the keyword "sorbet", the overall interest in frozen desserts is evident. This is an example of the inference problem [15].

For this reason, the public dictionary is assumed to

be diverse or unabridged. The approach works for common nouns, general terminology, etc. However, proper nouns and domain specific terms are not as easily obfuscated. Exposure in a small set may be sufficient to divulge knowledge, and enumerating the full set may not be possible.

Network indicators are domain specific and cannot be exposed in a public dictionary, even if the indicators were intermingled with a large number of unrelated (chaff) indicators. The adversary need only look for their address, domain name, user-agent, host name, etc. Furthermore, it may not be possible to enumerate every indicator. Our design does not reference filter entries through an association in a public dictionary.

Bethencourt notes that the public dictionary can be replaced by a hash function [11]. The client would use the hash of their private keywords as an index into the encrypted filter, and to create the filter. The provider would then use the hash during the search to calculate the number of matching keywords as a product of filter entries indexed by the hash values of every unique term in the document.

Using this hashed-index approach, the dictionary is eliminated. This modification to the original scheme creates a system in which sensitive terms, geographic locations, proper nouns, and other domain specific information can be searched without having to create a dictionary with every conceivable location, name, or data value. It is even possible that the information provider will not know all of the possible values, but will know how to perform the hash.

There is a downside to this hashed-index approach. A hash collision between a private keyword and an unrelated word could introduce a false positive during the search. While this is not generally an issue for full length hash values, we use a truncated hash. A balance between false positives and the size of the filter is required.

2 HASHED-INDEX TO FILTER ENTRIES

In our implementation, we utilize the hashed-index approach to eliminate the dictionary and to prevent the disclosure of sensitive network indicators (IP addresses and ports.)

Consider a private search for a set of sensitive IP addresses (Figure 1.) To construct a filter, the client hashes each address, truncates the hash to the defined size of the filter F, and sets the filter entry indexed by each hash value to an encrypted one $E(1)$. All other filter entries are set to an encrypted zero $E(0)$.

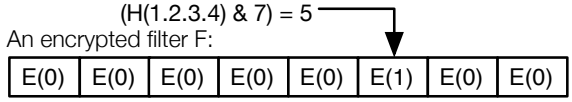

Figure 1: Locating a Filter Entry through a Hash

For the search, the provider hashes the address $h = H(address)$ from the packet, truncates the hash, locates a single entry from the filter $s = f_h$, and calculates the result as an exponentiation $r = s^{packet}$. (Partitioning *large documents* is discussed below.)

We emphasize that a singular datum is used in the search. A single IP address, either source or destination, is tested. In Ostrovsky's original system, multiple words from a document are extracted, and an equal number of associated filter entries are used as the product and in calculation of the encrypted match value s. Our search for a single datum (IP address, port, host name, etc.) imputes a reference to a single filter entry, and eliminates the calculation of a product of filter entries; the value for s is either $E(1)$ or $E(0)$, and eliminates the scaling factor m. The result r is not scaled.

A buffer thus, stores only the result r and is half the size in our implementation. This modification also induces a change to the tagging mechanism.

3 DETECTION TAG

The client must be able to identify a packet saved to the output buffer from a collision of packets in a buffer position. Ostrovsky suggested a method that appended k-bits, divided those bits into three bit triples, and set a bit in each triple to one [2]. The provider embeds this encoding, and the client detects a packet when all triples have a single bit set. Otherwise, the client knows that the buffer position contains a collision of packets.

Our work assessed the number of false positives resulting from $k/3$ appended triples, and offered an alternative based on appended hash values [7]. Both methods were probabilistic and are applicable when performing multiple keyword searches.

Our application for private packet filtering is special case. Since the provider only references a single fil-

ter entry, the result r is not scaled by the number of matching keywords m; s is either $E(1)$ or $E(0)$. The provider can thus, append each packet with a fixed width *detection tag*: 0x0001. The client detects a packet only when the lower sixteen bits equal one in a buffer position. Otherwise, the client knows that the position contains a collision (and the number of collisions in that position.)

Our tagging scheme is used in conjunction with our packet partitioning method.

4 PACKET PARTITIONING

Packet are expressed as a binary string, or more simply, a packet value. Large packets p then, refer to values that exceed the modulus, $p \notin \mathbb{Z}_N$. The *large document problem* refers to methods that enable private searching for large values. One approach equates with a (short) link or reference. The provider would then store the link to the buffer, and the client would then utilize the link to retrieve the document from a trusted third party.

A direct storage of packet data in a buffer was better suited for our needs, but this necessitated a packet partitioning method. Fortunately, working with packets is a special case of the large document problem. Packets sizes could be bound to a Maximum Transmission Unit (MTU) size. We did not need to address arbitrary length documents, and could thus, employ a simple and fixed partitioning strategy.

Our partitioning strategy segments each packet such that each $p' \in \mathbb{Z}_N$, and then appended each packet with a 48-bit *partition designator*. This designator consists of four bit fields as shown in Figure 2: the dection tag, the match identifier (match ID), the partition total, and the current partition number. The provider sets the detection tag to 0x0001, the matchID to the current packet count value, the partition total to the number of partitions used to segment this packet, and the partition number to the reference value of the specific partition.

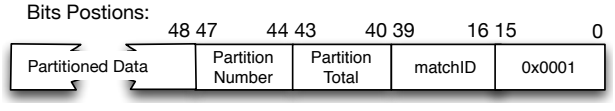

Figure 2: Format of the Partition Designator

The provider partitions each packet value, appends the designator, calculates the search result as the exponentiation over each of these appended partitions, and saves the result r to selected buffer positions

We provide a brief example. Imagine that a packet consisted of 2 bytes, $p = 0xAABB$, that this packet was the fourth *document* to be inserted into the buffer, and the packet had to be partitioned on a byte boundary: $p' = 0xAA$ and $p'' = 0xBB$. The encoded partitions would form a list:

$$p = \{0xAA120000040001, 0xBB220000040001\}$$

During recovery, the client detects to two partitions since both document tags are 0x0001, recognizes that both have the match ID (0x000004), recognizes that this packet consists of two partitions, and that both partitions were recovered. The client reassembles the partitions, and creates the packet. A Packet Capture (PCAP) file is created, and common tools used for further analysis. At the limit of the buffer's capacity, some packets may be lost when all partitions are not available.

Table 2 depicts our modified private search for a packet p, shows three packet partitions, that the filter entry is referenced through a hash of the indicator (an IP address), and that the exponentiation is calculated over each partition.

Table 2: Searching with Partitioned Data

Provider: The Search

Partition a packet:	$p = \{p', p'', p''', ...\}$
Append designators:	$p = \{p'\|\|0x130000990001,$
	$p''\|\|0x230000990001,$
	$p'''\|\|0x330000990001\}$
For convenience, let:	$p = \{p'_1, p''_1, p'''_1\}$
Retrieve filter entry:	$s = f_{H(ip)} = E(1)$
Calculate result:	$r = s^{\{p\}} = E(1)^{\{p\}}$
	$= E(1)^{\{p'_1, p''_1, p'''_1\}}$
	$= \{E(p'_1), E(p''_1), E(p'''_1)\}$
Save to buffer positions:	$B \leftarrow r$

As a design decision, an MTU of 1500 bytes was assumed. Given that a PCAP header is 16 bytes per packet, a packet requires at most 12 partitions when a 1024-bit modulus is used. We additionally need to account for the partition designator in each partition. At most, 13 partitions are required for a 1500 byte packet. A histogram of these partitions appears in Section VI−3.

5 ITERATIVE DOCUMENT RECOVERY

Our prototype utilizes Danezis's iterative method to recover documents from the buffer [5]. The simplicity and acceptable recovery rates justified use. Recovery rates are discussed in Section VI-1.

We present an example of the iterative document recovery method described by Danezis, and offer two modifications. First, the buffer positions are calculated from a hash chain. Hash chains are discussed in [16]. Briefly, a hash chain is a cryptographic primitive based on the successive invocation of a hash function for a given seed value. The values from the hash chain provide a pseudorandom source of buffer positions that can be derived by both the client and provider.

For example, let $H^l(p')$ represent a hash chain, seeded by a partitioned packet p' with length l using hash function $H(x)$. The values of the hash chain are then:

$$positions = H^l(p') = \{H(p'), H(H(p')), \ldots, H^{l-1}(i)\}$$

In practice, hash chains may repeat values. Implementers will need to delete repeated values, and extend the calculation of a hash chain to assure that all values in the chain are unique.

The packet value acts as the seed, and the sequence of (truncated) hash values from the chain reference buffer positions. When the provider and client possess a packet, the same buffer positions can be calculated.

This means that *truncated hash values* are used twice in our system, and for two different purposes: The provider hashes the indicator as a reference into the filter, and the provider uses the hash chain of the packet to designate where copies of the (encrypted) packets are stored too. Upon discovering a packet, the client can calculate the same hash chain, subtract the packet value from all copies in the buffer, and repeat until the buffer is empty or no new packets are discovered.

Assume that the provider creates a buffer B with eight positions, initializes each position to $\{0\}$, encrypts the buffer with the client's public key, and agrees to store *three* copies of a packet in the buffer. In this example, the provider conducts a search of a database containing four packets $T = \{p_1, p_2, p_3, p_4\}$, and does not know that all four matched a filter entry associated with a sensitive network indicator.

The provider appends the partition designator to each packet. Each packet consists of a single partition for simplicity in Table 3:

Table 3: Partitioned Packet Values

Packet partition	*Packet and appended designator*
p'_1	$p_1 \| \| 0x110000010001$
p'_2	$p_2 \| \| 0x110000020001$
p'_3	$p_3 \| \| 0x110000030001$
p'_4	$p_4 \| \| 0x110000040001$

The provider calculates the hash chain on the four partitioned packets in Table 4:

Table 4: Hash Chain Values

Packet	*Hash chain*	*Hash Chain Values*
p'_1	$H^3(p'_1)$	$\{0, 1, 5\}$
p'_2	$H^3(p'_2)$	$\{1, 3, 4\}$
p'_3	$H^3(p'_3)$	$\{2, 3, 5\}$
p'_4	$H^3(p'_4)$	$\{2, 3, 4\}$

The provider saves three copies of each to the buffer, and returns the buffer to the client. The client decrypts the buffer as shown in Table 5:

Table 5: Client Decrypted Buffer

Buffer position	*Buffer values*
0	$\{p_1 \| \| 0x1100010001\}$
1	$\{(p_1 + p_2) \| \| 0x220000030002\}$
2	$\{(p_3 + p_4) \| \| 0x220000070002\}$
3	$\{(p_2 + p_3 + p_4) \| \| 0x330000090003\}$
4	$\{(p_2 + p_4) \| \| 0x220000060002\}$
5	$\{(p_1 + p_3) \| \| 0x220000040002\}$
6	$\{0\}$
7	$\{0\}$

The client detects a single packet p'_1 in buffer position zero (the lower sixteen bits equals 0x0001), removes the full partition designator, calculates the hash chain of p'_1, and subtracts the packet value from the buffer tuple values in positions $\{0, 1, 5\}$. The resulting buffer contains the following entries as shown in Table 6:

Table 6: Client Decrypted Buffer - First Iteration

Buffer position	*Buffer values*
0	$\{0\}$
1	$\{p_2 \| \| 0x110000020001\}$
2	$\{(p_3 + p_4) \| \| 0x220000070002\}$
3	$\{(p_2 + p_3 + p_4) \| \| 0x330000090003\}$
4	$\{(p_2 + p_4) \| \| 0x220000060002\}$
5	$\{p_3 \| \| 0x110000030001\}$
6	$\{0\}$
7	$\{0\}$

In the next iteration, the client identifies two new packets: p'_2 and p'_3 in positions 1 and 5. Calculates their hash chain values, and subtracts their values from the buffer positions, respectively. The result of this second iteration is shown in Table 7:

Table 7: Client Decrypted Buffer - Second Iteration

Buffer Position	*Buffer Tuple Values*
0	$\{0\}$
1	$\{0\}$
2	$\{p'_4 \| \| 0x110000040001\}$
3	$\{p'_4 \| \| 0x110000040001\}$
4	$\{p'_4 \| \| 0x110000040001\}$
5	$\{0\}$
6	$\{0\}$
7	$\{0\}$

In the final iteration, the client detects packet p'_4, subtracts the packet values, and detects that every buffer position contains zero. The client halts processing, Table 8:

Table 8: Client Decrypted Buffer - Final Iteration

Buffer Position	*Buffer Tuple Values*
0	$\{0\}$
1	$\{0\}$
2	$\{0\}$
3	$\{0\}$
4	$\{0\}$
5	$\{0\}$
6	$\{0\}$
7	$\{0\}$

6 PCAP RE-ASSEMBLY

In our prototype, the client performs one final step: The client uses the time stamp from the PCAP packet header, sorts all packets based on this timestamp, and produces a sorted output. The client then prepends the PCAP header to these sorted packets, and writes a PCAP file.

V OUR METHOD FOR CONJUNCTION

Our objective was to broaden the private search capability, and to explore conjunctive methods that could be integrated within the system facets just described. Our conjunction operator thus, uses the structure from Ostrovsky's private search system, uses the Paillier cryptosystem, and uses a hashed-index approach for filter entry lookup.

1 SUMMED HASH CONJUNCTION

The client creates an encrypted filter F by selecting pairs of sensitive indicators $\{w_1, w_2\}$ where each element associates with a field from a packet. The client then produces a hash value of each pair, $h' = H(w_1)$ and $h'' = H(w_2)$, and calculates the bitwise summation (exclusive OR) of these hash values $h = h' \oplus h''$. The client then truncates the summation to a bit length. This length is chosen such that the formation of the encrypted filter has an acceptable probability to minimize a hash collision with unrelated indicators.

The resulting set of summed hash values H represents the conjunction values of interest. The client uses the hash values as an index into the encrypted filter F and encrypts a value of one $E(1)$ at each index. All other positions in F are assigned to $E(0)$. The filter and the names of the fields are sent to the information provider.

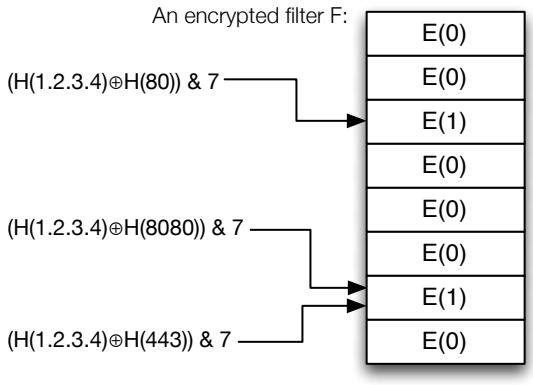

Figure 3: A Summed-Hash Conjunction Filter

Table 9: Private Stream Searching with a Conjunction Filter

Client: Generates the query:	
Select private keywords:	$K = \{\{1.2.3.4, 80\}, \{1.2.3.4, 8080\}, \{1.2.3.4, 443\}\}$
Calculate the hash of keywords:	$H = \{2, 6, 6\}$
Construct an encrypted filter:	$F = \{E(0), E(0), E(1), E(0), E(0), E(0), E(1), E(0)\}$
Send F and record names to the provider.	
Provider: Performs the search:	
Construct an encrypted output buffer:	$B = \{E(0), E(0), E(0), E(0)\}$
Perform a Search on a packet:	$p =$ {1.2.3.4,80,packetData}
Calculate the filter index for the conjunction:	$h = (H(1.2.3.4) \oplus H(80))\&7 = 2$
Retrieve the filter entry:	$s = f_h = f_2 = E(1)$
Append the partition designator:	$p' = packetData \| \| 0x110000010001$
Calculate the exponentiation:	$r = s^{p'} = E(1)^{p'} = E(p')$
Calculate the hash chain:	$C = H^3(p') = \{0, 1, 3\}$
Save copies of the result to buffer positions:	$b_0 = r \times b_0$
	$b_1 = r \times b_1$
	$b_3 = r \times b_3$
Return the buffer to the client:	$B = \{E(p'), E(p'), E(0), E(p')\}$
Client: Processes the result:	
Decrypt the buffer:	$B = D(\{E(p'), E(p'), E(0), E(p')\})$
	$= \{0, p'\}$
Recover the matching packet:	$p = (p' >> 48) = \{packetData\}$

For each packet p in a database, the provider will extract two record entries w_1 and w_2, and calculate their hash values, $h' = H(w_1)$ and $h'' = H(w_2)$. The summation of these hash values $h = h' \oplus h''$ is used as an index into the filter (as before, the hash is truncated to the size of the buffer.) The filter entry $s = f_h$ is extracted from F, and multiple copies of the result $r = s^d$ are saved to the output buffer B.

As an example, lets generate a query for an IP address and port. In Figure 3, the client generates a filter for a conjunction of three private terms: $\{1.2.3.4, 80\}$, $\{1.2.3.4, 8080\}$, and $\{1.2.3.4, 443\}$. The client hashes the words for each term, sums the hash values, truncates the values via a bitwise AND with 7, and calculates the hash indices, 2 and 6. The client then sets these filter entries to an encrypted value of one $E(1)$. In the example, two terms reference the same filter entries. This is acceptable and is not a false positive. The filter and record names (IP address and port) are sent to the provider.

Table 9 continues this example, and highlights the changes induced by our modifications: Notice that the conjunction eliminates the public dictionary. Restricting the search to fields from the packet eliminates the product of filter entries during the search; the provider only references a single filter value, and thus, the encrypted match value is only one or zero (in the plaintext domain.) The result of the provider's exponentiation r is not scaled and the provider only stores copies of r to the buffer (the buffer is half the size.) The example also incorporates Danezis's iterative document recovery method.

The provider constructs an encrypted output buffer as a singular list of encrypted zeroes, and performs a search on a packet p. The provider extracts the

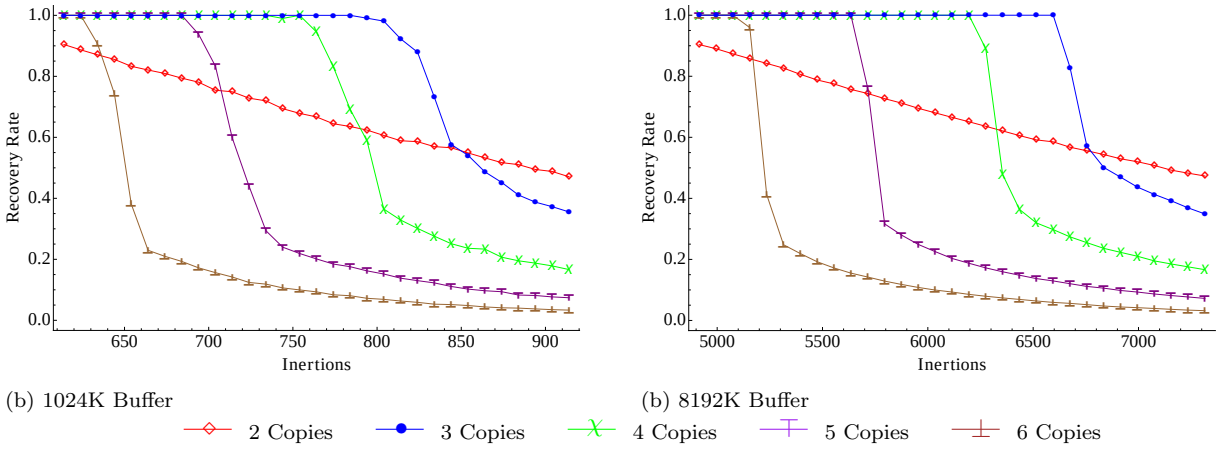

Figure 4: Effects of Multiple Copies on Document Recovery – Three copies is best

IP address and port, calculates the filter index as a summed hash $h = 2$, and retrieves the third filter entry $s = F_2 = E(1)$. After calculating the exponentiation $r = s^{p'} = {F_2}^{p'} = E(p')$, the provider saves this result to the buffer positions as designated by the hash chain $C = \{0, 1, 3\}$, and returns the buffer to the client.

The client decrypts the buffer and extracts one (deduplicated) packet. The example in Table 9 is similar to that of Table 2 emphasizing the integration into the overall implementation.

VI SIMULATION

We conducted two simulations that assessed Danezis's iterative document recovery method: The first measured the optimal number of document copies stored to a buffer. The results are applicable for our private packet filtering system and private stream searching in general; we will use the term *documents*, and recognize that the results apply for packets, files, etc. Second, we measure the number of documents recovered per iteration.

Finally, we measure the number of partitions observed for a given PCAP data set, using our packet partitioning method.

1 NUMBER OF DOCUMENT COPIES

Our simulation examined the effects of multiple document copies in an output buffer. The effects are measured in terms of a "recovery rate", the fraction of documents retrieved from a buffer for a given number of insertions. Ideally, the number of matching documents saved to a buffer should match the number retrieved. The recovery rate should be 1.0 for any number of insertions into the buffer.

However, as the number of insertions increases, some documents will be overwritten with a multiplicity greater than can be recovered from the buffer. At some point, full recovery will not be possible, and the recovery rate will decrease. The simulation thus, determines the operational capacity of a buffer, where full, acceptable, and unacceptable recovery rates can be expected.

The simulation assumes that *all searches are successful*, that each document is inserted into a buffer. This permits analysis. Recognize that document selection would be driven by the number of private keywords and their frequency of occurrence in an actual corpus.

Figure 4 presents the recovery rates of our implementation of the iterative document recovery method. The simulations used MD5 for the construction of the hash chain, varied the buffer length, the number of copies, and the number of insertions. Two buffer lengths were exercised: 1024 and 8192 positions. The number of document copies was varied from 2 to 6. Insertions ranged from 0.6 to 0.9 of the buffer size.

The buffer with 1024 positions was exercised with 614 to 921 document insertions, and 4915 to 7372 insertions were used for the buffer with 8192 positions. The documents were big integers and selected at random. Lastly, each simulation was executed over fifty trials with these parameters over the range of insertions. The recovery rate of each trial was then averaged.

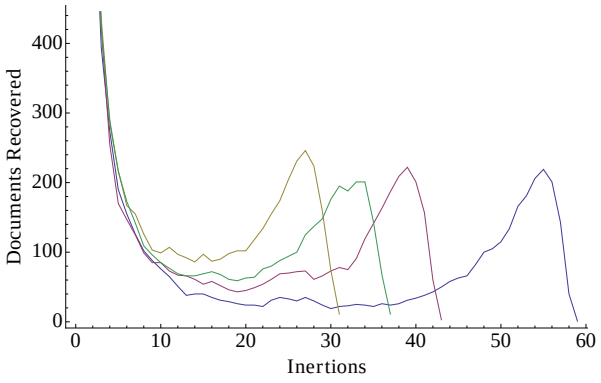

Figure 5: Documents Recovered per Iteration

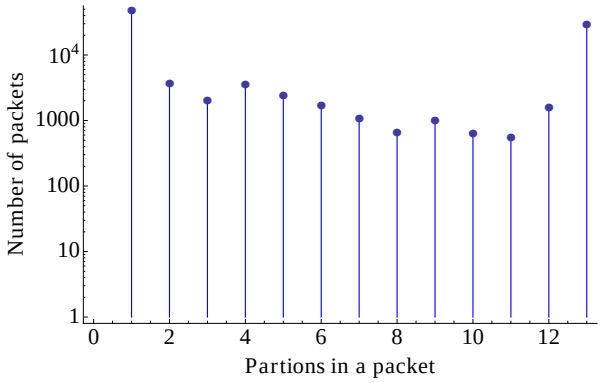

Figure 6: Histogram of Packet Partitions

We seek a line from Figure 4 that provides full recovery over the greatest range of insertions. We observe that the line, representing three document copies, offers this feature and for both buffers (drawn as a thick blue line.)

Consider the buffer with 1024 positions, full recovery is possible to 800 insertions when three copies is used. At that point, the recovery rate decreases. In some scenarios, this may be initially acceptable, but as the upper bound of tested insertions is reached, recovery is less than half. A similar result occurs in the buffer with 8192 positions. Full recovery is possible to 6700 document insertions when three copies is used.

The extent that documents could be recovered for other copy values did not perform as well. Notice that when 4, 5, and 6 document copies were used for storage, full recovery could only be achieved when fewer documents were inserted into the buffer. If 4 copies are used, full recovery is possible for 775 documents for the 1024 buffer. Whereas, when 6 copies are used, recovery is possible to 625 documents. Too many copies and too many insertions leads to an increased number of summations in a buffer position that cannot be resolved by the client. The opposite is also true. In the case of 2 copies, full recovery is never achieved over the range of tested insertions. When too few copies are used, no single copy remains for recovery.

Our results confirm those presented by Danezis [5]. Since both used an iterative method to remove newly discovered documents from the buffer (our algorithm used a different positioning method), similar observations should be expected: Both efforts dispel the notion that implementers should use a large number of document copies as the buffer size increases, and expect a high recovery rate. This notion does not hold for finite and practically sized buffers.

Implementers should use three document copies as the optimal value, and certainly for buffers sized to a few thousand. Last, Danezis indicates that full recovery can be achieved when the number of insertions is less than half the size of the buffer.

Our experiments show that a greater capacity is possible. Full recovery was achieved at 800 and 6700 insertions for our two buffer sizes. This is approximately 0.80 of a buffer's size.

2 DOCUMENTS PER ITERATION

Figure 5 depicts the number of documents returned per iteration from the iterative recovery method. In this simulation, the buffer size was 8192 and the the buffer held 6625 insertions (near capacity.) The figure shows four unique trials (lines), that a high number of documents are initially discovered, followed by a bottom, and then a peak number of documents are recovered before the process halts. All documents were recovered in these four simulations.

The intent was to discover the linear relationship between the size of the buffer and the number of iterations needed to recover all documents. Some implementers may be tempted to establish loop bounds relative to a buffer's size, and as a means of stopping an errant recovery. For example, a buffer with 8192 positions seems to require no more than 60 iterations to process (approximately 1% of the buffer size.)

We recommend a more effective measure: Only permit positive buffer values. When the client subtracts

a document value from the assigned buffer positions, the result should always be positive. A negative value in a buffer position identifies an errant condition, and halts processing.

3 COUNTING PACKET PARTITIONS

Figure 6 shows the number of packet partitions after processing the "Nitroba University Harassment Scenario" [17]. This digital forensic exercise includes a network dataset, the "Nitroba" corpus. This data set contains 91,144 IP packets.

In this corpus, there were approximately thirty thousand packets that fit in a single partition, and nearly as many that fix into 13. These partitions introduce a constant factor in the computation of the search (specifically the exponentiation), and incur the greatest computational cost of the system. Fortunately, the exponentiation of r across multiple packet partitions can be parallelized. Each exponentiation is independent; an area for future consideration.

VII THE LANGUAGE

In this section, we provide a brief summarization of our high level language for *private packet filtering*, and present the integration of our summed hash conjunction. The intent is to convey the salient aspects of the language, show how our conjunction can be used, and present an application for cyber defense. A full discourse, definition of the grammar, and details of our prototype appear in [6]. We switch from client and provider, to terms familiar with cyber defense: a network defender and partner.

The language is designed for use in a *collaborative environment* where a network defender and partner agree to the notion of a private search. The defender has access to sensitive attack indicators that if revealed, could damage the source. The partner is willing to share packet data that matches a query. If any of the sensitive indicators are discovered on the partner's network, the defender can notify the partner of the malicious activity (exposing only that indicator.)

Using our language, the client enters sensitive attack indicators, uses our parser to transform those indicators into an encrypted filter, and transfers the resulting code to the partner. The partner conducts the search and returns an encrypted buffer. In total, there are three files: one for the query which the defender keeps secret, one for the search, and one for the output buffer. After extracting packets, our prototype produces a PCAP file for use with other tools.

1 QUERY GENERATION

Listing 1 depicts the code for query generation. This file consists of three sections: declarations, assignments, and an expression of a packet filter. There is a collision in nomenclature. Our "packet filter" selects packets that are then processed by the private search engine. Since the "encrypted filter" is an integral aspect of private search, "filter" is used as a keyword. A packet filter uses an "encrypted filter" to privately search packets.

```
# Declarations
key public paillier kPub {
  buffer counjunctionOutput {
    conjunction aConjunction {
      filter in_addr dst _x;
      filter port dst _y
    };
  };
  buffer malsiteOutput {
    filter in_addr dst malSites;
    filter port dst dPort
  }
};

graph myGraph {
  source file  inFile;
  whitelist in_addr dst  whList
};

# Assignments
kPub = { include "kPub.key" };

malSites = { 5.6.7.8, 5.6.7.9  };
dPort = {80, 8080, 443};
aConjunction = {  {1.2.3.4, 80},   {1.2.3.4, 8080},
  {1.2.3.4, 443}, {malSites, dPort}
};

malSites = { include "listMaliousIPs.txt"  };

whList = { 192.168.0.0/16 };

# A packet filter
myGraph = {
  inFile -> whList :: aConjunction;
  whList :: malSites
};
```

Listing 1: The Query: Indicators Kept Private

There are two declarations: `key public paillier kPub` and `graph myGraph`. The first declares a variable for a public key, two output buffers, a conjunction, and two filters. We emphasize that the hierarchical structure establishes the cryptographic relationships necessary for a private search. Last, the conjunction variable is independent of the two filter variables, but since the data type of these filters match that of the conjunction, the variables can be assigned.

The second declaration establishes a packet filter as a graph of nodes and edges. The `graph` variable defines two nodes, strictly variables, one for data input and data reduction.

The code then displays four assignments to initialize the public key, two filter variables, and the conjunction. The parameters for the public key are included from a file. The filter variables are assigned to two IP addresses, and a list of port values respectively.

The conjunction variable is then assigned as a list of four terms. The first three are scalar assignments. The fourth term is a short hand notation that creates conjunctive terms from an element-wise pairing of the filter variables. Here, the matching types from the two previously assigned filters are used. In total, the conjunction is assigned to nine conjunctive terms.

The final assignment creates a white list, and a net block of destination addresses to exclude from the private search. The white list dis-regards packets to common addresses. Last, a list of IP addresses is included from a file and assigned to the IP filter.

The final section depicts the edge assignments of the packet filter. Variables are interconnected via the "`->`" operator and sink to a filter variable, via the "`::`" operator. This interconnection forms a path from the input source, `inFile`, to a white list variable, and then processes packets in filters.

The defender submits the code in Listing 1 to our parser. The parser removes the sensitive indicators, computes, and writes the lists of encrypted values. The defender sends this public file to the partner.

2 THE SEARCH

Listing 2 depicts the public form of the query, and has a syntax similar to that seen in Listing 1. The significant difference occurs in the assignment of `aConjunction` and `malSites`. The values for both encrypted filters are included from a file.

```
# Declarations
key public paillier kPub {
  buffer counjunctionOutput {
    conjunction aConjunction {
      filter in_addr dst _x;
      filter port dst _y
    }
  }

  buffer malsiteOutput {
    filter in_addr dst malSites;
  }
};

graph myGraph {
  source file  inFile;
  whitelist in_addr dst  whList;
};

# Assignments
kPub = {  include "kPub.key" };

aConjunction = { include "aConjunction.fltr" };

malSites = { include "malSites.fltr" };

inFile = {
  "  Enter a PCAP Filename:  "
};

whList = 192.168.0.0/16;

# Graph Execution
myGraph = {
  inFile -> whList :: conjunctionOutput;
  whList :: malsiteOutput
};
```

Listing 2: The Search: Public Code for Private Packet Filtering

When the partner executes the packet filter, packets flow through the graph and into both filters. We represent this action in Figure 7, and the "tee" that directs traffic into `aConjunction`, and `malSites`. Packets that match in either the conjunction, or the (standard disjunction) filter are stored to the output buffer associated with that filter. Specifically, the results from `aConjunction` are saved to `conjunctionOutput`, and the results from `malSites` are stored in `malsiteOutput`.

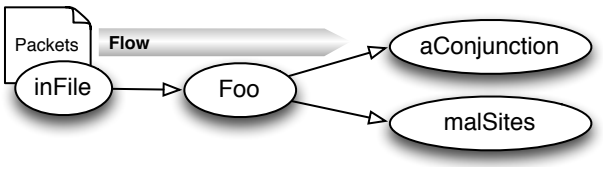

Figure 7: A Graph Representation of a Packet Filter

3 THE RESULT

After completing the search, the partner's system will create an output file consisting of a PCAP header, identified by magic number `0xa1b2c3d4...`, and the encrypted data from the buffers. The partner sends this file to the defender, who decrypts the buffers and assembles the matching packets into PCAP files. The defender can then use additional packet processing tools. For brevity, only one ASCII hex value from the buffer is displayed in Listing 3:

```
# The output buffer:
#   A PCAP header and encrypted buffer
conjunctionOutput = {
  0xa1b2c3d4...,
  {
    0x112233445566778899aa...,
    ...
   }
};
malsiteOutput = {
  0xa1b2c3d4...,
  {
    0xffeeddccbbaa99887766...,
    ...
   }
};
```

Listing 3: The Result: An Encrypted Output Buffer

Our language supports private searches for conjunctions and disjunctions of IP addresses and ports. The language thus, provides access to the low level construct of private searching, and in a form that is familiar to network defenders.

VIII PROTOTYPE

A prototype was constructed in order to demonstrate a working model of the language. The prototype consists of a lexical analyzer (Flex), a Bison parser, C++ code, and used *Mathematica* for the private search operations.

There are two principal C++ classes, an engine and a base variable class. Each instance of a variable is derived from the base variable class, and implements specific functionality: a key class provides access to the cryptographic parameters, a buffer class manages the encrypted output, node variables select specific packets, a filter class, and a conjunction class. The engine interfaces with the Bison parser, instantiates variables, and enforces semantic constraints in conjunction with each variable.

The result is three programs, `ppf-generate`, `ppf-search`, and `ppf-recover`, that derive from a single code base to generate the query, perform the search, and retrieve results. Listing 4 illustrates how our private packet filter language and the examples described would be used by the defender and partner.

IX CONCLUSION

We defined a summed hash conjunction for private stream searching. The conjunctive operator integrates into the private search system without any additional data structures, complex calculations, and removes the requirement for a public dictionary. This broadens the capability provided by the private search system, and permits the use of sensitive indicators that cannot be exposed.

The insight and the theory from prior research has been adapted, and a practical realization of private stream searching has been realized (we have not achieved full packet capture at line rate.) Our language and prototype for private packet filtering has been extended and includes our conjunction. To achieve these goals, several system modifications were made: Our conjunction references a single filter entry through a hash which removes a calculation of filter entries, does not scale the result, and reduces the buffer size by one half. We also described how to tag and partition packets to solve the *large documents* problem, that three document copies is optimum, and a buffer's capacity is approximately 0.8 the size of the buffer.

The experiences and insights are intended for future implementers and research, and open new venues for research. The conjunction and language can be adapted for further cyber research in private antivirus scanning, file scanners, intrusion detection, and the discovery of malicious content without revealing knowledge of the search.

```
defender$ ppf-generate -r privateIndicators.ppf -w public.ppf
defender$ echo "Send public.ppf to the partner."

partner$ ppf-search -r public.ppf -w buffer.ppf
  Enter PCAP filename:    partner.pcap
partner$ echo "Return the buffer file, buffer.ppf"

defender$ ppf-recover -k kPrivate.key -r buffer.pss -w partnerActivity.pcap
```

Listing 4: A Demonstration of Private Packet Filtering

References

[1] E. Hutchins, M. Cloppert, and R. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," in *Conference on Information Warfare and Security*, pp. 113–125, 2011.

[2] R. Ostrovsky and W. E. Skeith, "Private searching on streaming data," in *Advances in Cryptology (CRYPTO 2005)* (V. Shoup, ed.), vol. 3621 of *Lecture Notes in Computer Science*, Springer, 2005.

[3] R. Ostrovsky and W. E. Skeith, "Private searching on streaming data," *Journal of Cryptology*, vol. 20, no. 4, pp. 397–430, 2007.

[4] G. Danezis and C. Diaz, "Improving the decoding efficiency of private search," in *the Dagstuhl seminar on anonymity and its applications*, 2005.

[5] G. Danezis and C. Diaz, "Space-efficient private search with applications to rateless codes," in *Financial cryptography (FC'07)*, vol. 4886, pp. 148–162, Springer-Verlag, 2007.

[6] M. Oehler, D. Phatak, and A. Sherman, "A private packet filtering language for cyber defense," in *the 8th Annual Symposium on Information Assurance (ASIA'13)*, (Albany, New York), pp. 46–55, 2013.

[7] M. Oehler and D. Phatak, "A conjunction filter for private stream search," in *the 8th ASE/IEEE International Conference on Privacy, Security, Risk, and Trust (PASSAT'13)*, (Washington, DC), 2013.

[8] M. Oehler and D. Phatak, "A simulation of document detection methods and reducing false positive for private stream searching," in *the 8th International Workshop on Data Privacy Management (DPM'13)*, (Egham, England), 2013.

[9] S. M. Bellovin and W. R. Cheswick, "Privacy-enhanced searches using encrypted bloom filters," Tech. Rep. CUCS-034-07, 2007.

[10] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International conference on the theory and applications of cryptographic techniques (EUROCRYPT'99)*, vol. 1592, pp. 223–238, 1999.

[11] J. Bethencourt, D. Song, and B. Waters, "New construction and practical applications for private stream searching (extended abstract)," in *IEEE Symposium on Security and Privacy (SP'06)*, pp. 132–139, IEEE Computer Society Press, Los Alamitos, 2006.

[12] J. Bethencourt, D. Song, and B. Waters, "New techniques for private stream searching," *ACM Transactions on Information and System Security*, vol. 12, no. 3, 2009.

[13] D. Boneh, E. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertext," in *Theory of Cryptography (TCC'05)*, vol. 3378, pp. 325–341, 2005.

[14] X. Yi and E. Bertino, "Private searching for single and conjunctive keywords on streaming data," in *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society (WPES '11)*, pp. 153–158, 2011.

[15] D. E. Denning and P. J. Denning, "Data security," *Computing Surveys*, vol. 11, no. 3, pp. 227–249, 1979.

[16] N. M. Haller, "The S/Key one-time password system," in *the Symposium on Network Distributed Systems, Security*, (San Diego, California), pp. 151–157, 1994.

[17] S. Garfinkel, "Digital corpora producing the digital body – nitroba university harassment scenario." NSF DUE-0919593, http://digitalcorpora.org/, 2011.