A Private Packet Filtering Language for Cyber Defense

Michael Oehler, Dhananjay S. Phatak and Alan T. Sherman

***Abstract*—Packet filtering is a central facet of cyber defense used to detect adversarial activity on a network. Detection stems from defensive efforts to discover new attack indicators, and efforts to share indicators with collaborating partners. There are instances where the sensitive nature of an indicator prohibits outright disclosure. Our private packet filtering language adapts the concept of private stream searching, and defines a new capability to filter packet data without revealing the indicator or result. The syntax of the language, the code to generate a private query, search and result, and the semantic constraints enforced by the language are presented. A cyber defender retains control of sensitive indicators, and coordinates a response action without revealing every indicator to the partner or risk disclosure to the adversary.**

***Index Terms*—Cyber Defense, Data Privacy, Oblivious Transfer, Packet Filtering, Private Search, Security Language**

I. Introduction

THE DISCOVERY of new cyber-attack indicators requires significant effort and expense. To ensure the greatest benefit, cyber defenders share new indicators with other collaborating partners (e.g., government and industry, corporations and their international subsidiaries.) However, indicators may be improperly disclosed by a partner, or exposed during an intrusion. This gives the adversary an opportunity to change their Tactics, Techniques, and Procedures (TTPs), reducing the value of the indicator. The defender is faced with a challenge. There is a need to share indicators and a requirement to control their dissemination.

Our contribution recognizes the association between this defensive challenge, and the capability provided by private stream searching. Specifically, we adapt the private search capability presented by Rafail Ostrovsky and William Skeith in 2005 [1], [2], and create a language for private packet filtering. Our high level language preserves the confidentiality of the indicator, and packets returned by the search.

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 of adversarial tactics, controls which attack indicators are revealed, and advises the partner of current threat activity. This is a new scenario for cyber defense and private search.

The design of the language is intuitive and readable. For example, five lines define the cryptographic structure for a private search. Additional indicators and output buffers can also be specified in this structure. A single query can select different types of indicators and filter complex packet streams. This ability to search multiple indicators privately is unique.

Ostrovsky defined private stream searching. Bethencourt [3], [4] and Danezis [5] improved the storage efficiency of the output buffer. Yi constructed a conjunctive search [6], and Finiasz integrated Reed-Solomon codes with private searching [7]. We are also aware of Bethencourt's toolkit for private searching [8]. A high level language for *private stream searching* has not been previously formalized.

We name the language PPF for Private Packet Filtering.

II. Private Stream Search

In this section, we describe the salient features of private stream search. The terms, client, provider, document, and keywords are a generalization. Their use provides clarity, and permits an illustration of private search. In our context, these terms map to defender, partner, packets, and attack indicators, respectively. Last, the term, filter is retained, and a resulting collision in nomenclature is discussed at the end of this section.

A private search system preserves the confidentiality of the search criteria, and involves a client, and one or more information providers. A client generates a query, the provider performs the search, and delivers a response back to the client without gaining knowledge of the query or the result. The naïve approach transfers an entire data set from a provider to the client. Admittedly, this approach conceals the query from the provider. Ignoring bandwidth costs and a required client-side search, few providers would relinquish an entire data set (As an example, a partner is unlikely to divulge all network activity.) Alternatively, if the search criteria were kept secret, but knowledge of the result was evident, the structure of the query could be inferred. This is also unacceptable.

These concepts establish the fundamental properties of a private search system: the provider gains no knowledge of the query, cannot infer information about the query from the result, and client access is limited to results matched by the

Michael Oehler, Dhananjay Phatak, and Alan Sherman are with the Cyber Defense Lab, Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, Baltimore, Maryland 21250 (email: {oehler1, phatak, sherman}@umbc.edu).

TABLE I
AN ILLUSTRATION OF PRIVATE STREAM SEARCHING

Client (Defender): Generates the query:		
Define a public dictionary:	$D = \{gelato, sherbet, snowball, sorbet, zebra\}$	See Fig. 1
Construct an encrypted filter:	$F = \{E(0), E(0), E(0), E(1), E(0)\}$	
Send D and F to the provider.		

Provider (Partners): Performs the search:		
Construct an encrypted buffer:	$B = \{\{E(0), E(0)\}, \{E(0), (0)\}\}$	See Fig. 2
Search using this document:	$d =$ "$unlike\ gelato\ sorbet\ has\ no\ calories$"	
Calculate a product of filter terms:	$s = \prod_{w_i \in d \in D} f_1 = f_1 \times f_4 = E(0) \times \mathrm{E}(1) = \mathrm{E}(1)$	
Calculate the search result as the exponentiation:	$r = s^d = E(1)^d = E(d)$	
Save the result to a buffer position:	$b_2 = \{s, r\} \times b_2$ $= \{\{E(1), E(d)\} \times \{E(0), (0)\}\}$ $= \{E(1), E(d)\}$	
Return the buffer to the client:	$B = \{\{E(0), E(0)\}, \{E(1), E(d)\}\}$	

Client (Defender): Processes the result:		
Decrypt the buffer:	$B = D(\{\{E(0), E(0)\}, \{E(1), E(d)\}\})$ $= \{\{0,0\}, \{1, d\}\}$	See Fig. 3
Recover the matching document:	$d =$ "$unlike\ gelato\ sorbet\ has\ no\ calories$"	

query [9].

Ostrovsky and Skeith created a clever private search system using (partial) homomorphic encryption [1], [2]. The system preserves the confidentiality of the search criteria, the result, and allows the client to match a document on a disjunction of keywords. The system is based on the asymmetric cryptosystem defined by Paillier, and utilizes the additive homomorphic property of the cryptosystem [10]. The cipher text from the Paillier cryptosystem is randomized. Thus, an encrypted value will be indistinguishable from another, even the same value, using the same public key. For instance, the encryption of $E(1)$ is indistinguishable from some other value of $E(1)$.

The client creates a list of encrypted ones and zeroes, corresponding to keywords of interest and non-relevant terms from a public dictionary; an encrypted filter. The client sends the filter and dictionary to the provider.

The provider performs the search by calculating a product of entries taken from the filter that associate with words in a document, an exponentiation, and a second product to save results to an encrypted output buffer. These calculations are performed on encrypted values (in the encrypted domain.) The provider is thus, unaware of the query or search-result. Furthermore, multiple documents may be stored in the buffer, creating a system that streams results, a private stream search (PSS) system.

Table I illustrates a simplified example. Consider 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 buffer B. The search entails a single document d. A product of filter entries f_1 and f_4, corresponding to words existing in the document and dictionary is calculated, and the exponentiation (A product of encrypted terms is equivalent to a summation of plaintext terms.) The provider then randomly selects a buffer position b_2. Results are saved to the buffer as a pairwise multiplication. The client decrypts the buffer, and recovers the document.

We transition to terms related to cyber defense (defender, packet, indicator), but will emphasize that the role and meaning of the encrypted filter is retained. In fact, an encrypted filter is integral to private search; "filter" appears as part of our syntax. A collision occurs when referring to a "packet filter." We define a packet filter as a graph of nodes (variables) used to select certain packets that are then sent to the private search system. For instance, a defender may not be interested in packets destined to common IP addresses. A packet filter could be constructed to disregard these packets, before sending the remaining stream of packets to the encrypted filter.

Our language parallels the three phases of a private search, and this is reflected in Table I with a reference to a code listing. The syntax and role of each listing is discussed next.

III. A PRIVATE PACKET FILTERING LANGUAGE

To introduce the central features of the language, we work through an example that searches for a sensitive indicator, a single Internet Protocol (IP) address.

Fig. 1 contains the code for query generation. This file is constructed by the defender, who defines the cryptographic

structures, indicators, and packet filter. The indicators stored in this code will remain private. The defender uses our parser to transform the query to a public form. This public code defines the private search, and is transferred to the partner. When all packets are searched, the partner transmits an output buffer of encrypted packets to the defender. The search is finished.

The creation of the query, the transformation, transfer, resulting packets, and any response action define the supporting process for an overall system. Our objective focuses on the definition of the language to support this process.

A. Query Generation

Fig. 1 shows the code for query generation. There are three portions: declarations, assignments, and an expression of a packet filter. For clarity, each portion is preceded by a comment, represented by the pound symbol.

```
# Declarations
key public paillier kPub {
    buffer outputBuf {
        filter in_addr dst malSite;
        filter in_addr dst c2IP
    }
};

graph myGraph {
    source file inFile;
    whitelist in_addr dst whList;
    whitelist in_addr src whtLst2;

    whitelist port dst whList3
};

# Assignments
kPub = { include "kPub.key" };
malSite = { 69.25.94.22   };
whList = {  192.168.0.0/16   };
whList2 = { 10.10.10.0/24,   11.11.11.11  };

# A packet filter
myGraph = {
    inFile -> whList -> whtLst2 :: malSite
};
```

Fig. 1. Query generation: sensitive indicators kept private.

Declarations: Variables are declared before use. Each is given a type, and may be followed by a qualifier. Furthermore, variable declarations are structured either to bind the cryptographic relationship of the filter variables, or to establish the function of a packet filter.

There are two declarations in Fig. 1: $key\ public\ paillier\ kPub$ and $graph\ myGraph$. The first declaration expresses a public Paillier key, and is structured such that an output buffer and two encrypted filters are bound with the key. The qualifiers on the filters indicate that destination IP addresses will be searched. This key declaration establishes the cryptographic relationship used for a private search.

We note that the defender and partner use this public key to encrypted the filter and buffer respectively. The partner also uses the public key to perform the search. The defender uses the private key only to decrypt the buffer. This use of keys differs from that in a traditional public-key cryptosystem, which encrypts and decrypts a single document.

The second declaration establishes the packet filter. Specifically, we express a packet filter as a graph of nodes and edges. In this example, the graph variable is named $myGraph$, and includes the definition of four nodes, strictly four variables, one for data input, and three for data reduction. Data input is represented by the source declaration, and in this case input from a file is inferred: $source\ file\ inFile$. The remaining four declarations define whitelist nodes to discard packets of non-interest: $whList$, $whList2$, and $whList3$.

These node (variable) declarations will be bound with a filter variable via edge assignments. Together, all variables form a path, express a specific packet filter, and produce a private search system.

Assignments: The four assignments initialize the public key, a filter variable, and two whitelist variables. The public key is included from a file, and contains the modulus and public random integer, the public parameters of the Paillier cryptosystem. The filter variable, $malSite$ contains a sensitive indicator (A single IP address is used as a demonstration. Filter assignments typically contain a list of many indicators.) The remaining whitelist assignments depict two destination addresses, and a net block of source addresses to exclude from the private search.

Packet Filter: The final portion of Fig. 1 depicts the edge assignments of the packet filter. Variables are interconnected via the "–>" operator. We also introduce a sink operator, "::" to bridge nodes defined in the graph variable and that of the encrypted filter. The operator forms a path from the input source, $inFile$, passes packets through two whitelist variables, and then sinks packets in the filter variable, $malSite$ where the private search is performed. When executed, results are placed into the output buffer, $outputBuf$ that was previously related with the encrypted filter.

The network defender submits the code to the parser, transforms the indicators into a public form, and sends this public code to the partner.

B. The Search

The partner uses the public form of the code for the search as shown in Fig. 2. This listing displays a few alterations: The assignment of the filter variable, $malSite$, references an included file, $malSite.fltr$. This file contains the encrypted values of the indicators, and is also publicly releasable. A default assignment for the input variable, $inFile$, will prompt the partner for the name of a packet capture (PCAP) file.

Two additional lines define processing parameters for the output buffer. These parameters are produced by the parser in

lieu of a buffer assignment. The parameters specify the size of the buffer and that the partner's system constructs the buffer locally (i.e., The partner's system will encrypt a list of 1024 zeroes using the public key associated with the buffer.)

```
# Declarations
key public paillier kPub {
    buffer outputBuf {
        filter in_addr src malSite
    }
};
graph myGraph {
    source file inFile;
    whitelist in_addr dst whList;
    whitelist in_addr src whList2
};

# Assignments
kPub = {
    include "kPub.key"
};
malSite = {
    include "malSite.fltr"
};
inFile = {
    "Enter a PCAP Filename: "
};

whList = 192.168.0.0/16;
whList2 = { 10.10.10.0/24, 11.11.11.11 };

outputBuf.bufferSize = 2048;
outputBuf.production = local;

# Graph Execution
myGraph = {
    inFile -> whList -> whList2  ::  malSite
};
```

Fig. 2. The Search: public code for private packet filtering.

C. The Result

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

```
# The output buffer: A PCAP header
# and an encrypted buffer
outputBuf={
    0xa1b2c3d4…,
    {
        0x112233445566778899aa…
    }
};
```

Fig. 3. The Result: an encrypted output buffer.

Our application of private search exercises a special case. When a packet is searched, a single indicator (the destination address in this example) from a packet is tested against the encrypted filter. No more than one indicator can match for any given packet. Whereas in the general case, multiple words from a document could match, which scales a document by a constant factor. In our case, this scaling factor will be one, and does not have to be transferred to the defender. Our output buffer can thus, be initialized as a simple list of encrypted zeroes, and returned as a list of encrypted results without scaling.

IV. Syntax and Semantics

This section presents the formal syntax, private comments, processing parameters, and describes two principal and semantic tests: the variable and packet filter check.

Definition of the Syntax: The Appendix presents the syntax of the language in a traditional Backus-Naur Form (BNF): nonterminals are represented in a braced form, "<nonterminal>", and productions are presented with a single nonterminal on the left side of the "composed of" operator, "::=". Reserved words are in boldface. The syntax deviates slightly from tradition with the utilization of a regular expression range operator, and a repetition operator, for example "[a-z]" and "[0-9]{1,3}" to represent characters from the alphabet, and one to three digits, respectively.

Optional items (qualifiers) are bounded by square brackets. Curly braces "{" and "}", are terminals used to delimit the declaration of a key, buffer, or graph variable, and additionally when a list of values is required. We also use a state designator to bind the context of an assignment. For instance, the value assignment for an IP address variable is restricted by the "$ipInputState" designator. Finally, the start symbol, $< ppfProgram >$ defines our language as statements of declarations, assignments, and comments.

Private Comments: Within the syntax, there is a notion of public and private comments. When manipulating sensitive indicators, defenders are likely to attribute activity by intelligence source, origin, and by other characteristics of a named intrusion set. These comments need to remain private, are designated by a double pound, "##", and removed by our parser. Public comments, those that can still add clarification and can be sent to the partner, are defined by a single pound. For instance, the private comment in Fig. 4 identifies the indicators and attributes these indicators to a named threat actor. They are removed. The public comment remains in the

parsed version of the code. As a semantic rule, public comments are associated with the next variable, and in their order of appearance.

```
## Malicious hosts attributed to named
##threat actor, Fuzzy Bunny
##
# A filter assignment
malSite.obfuscate=true;
malSite={ 69.25.94.22 };
```

Fig. 3. Public and private comments.

Processing Parameters: Processing parameters define additional options.

The code fragment in Fig. 4 also depicts a processing parameter: the $malSite.obfuscate$ statement indicates that the variable name will be obfuscated in the public code (an obfuscated variable name is expressed as the Base64 string of a cryptographic hash.) This facet reduces a burden on the defender. Variable names can conform to practice, readability, and operational context in the private form, and presented in a non-revealing manner in the public form of the code.

In this instance, the obfuscation was localized to a specific variable. If the variable name had been excluded, the processing parameter is applied globally. For instance, $.obfuscate = true$, obfuscates the names of all variables.

The design includes three parameters that can be used by any type variable, $.obfuscate$, $.cwd$, and $.dataMap$, four parameters for buffer variables, and one for filter variables. The parameters are shown in Fig. 5.

```
# Applies to all variables
. cwd = <aDirectoryPath>
. datamap = [ include | inline ]
. obfuscate = [ true | false]

# Applies to buffer variables
. bufferS ize = <integer >
. production = [ remote | local]
. reuse = [ true | false]
. trigger = <integer >

# Applies to filter variables
. expand = <integer >
```

Fig. 4. Processing parameters for variables

The $.cwd$ parameter sets the working directory for output, and $.datamap$ indicates whether a variable's assigned data will be saved to an include file or inline with the code.

The $.bufferSize$ parameter establishes the number of entries in an output buffer. The $.production$ parameter indicates where the buffer is produced, either locally on the defender's system, or remotely on the partner's system. The $.reuse$ parameter specifies whether an initial copy of the buffer can be used again, after the maximum number of buffer insertions is reached. This threshold is expressed as a multiplicative factor of the buffer size, and specified by the $.trigger$ parameter. The $.expand$ parameter scales the filter, to reduce false positives. These parameters are discussed further in Section V, Facets of Design.

Variable Assignment Checks: After declaration, variables are assigned and then used in a packet filter, via edge assignments. These two states assigned and on a path, are tracked as part of the transformation to the public form of the code. A warning or error is generated if either one of these two states is not met, and as shown in Table II.

TABLE II
VARIABLE ASSIGNMENT CHECKS

Variable	Assigned?	On Path?	Result
filter	No	No	Warning Unused
	No	Yes	Error. Halt
	Yes	No	Warning Unused
	Yes	Yes	OK. Encrypted Filter
node	No	No	Warning Unused
	No	Yes	Warning No-Op
	Yes	No	Warning Unused
	Yes	Yes	OK. Use in Packet Filter

If a variable is not on a path, (the variable is not used in a packet filter), the variable is deemed unused, even if data has been assigned to the variable. A warning will be issued, indicating that the variable must be part of a packet filter, and then removed from the public code. This exclusion has no effect on the private form of the code, minimizes structure in the public code, and does not alter the intent of the search. This was the case for the whitelist variable, $whList3$ and the filter, $c2IP$ from Fig. 1. Neither variable appears in Fig. 2. A greater challenge occurs when a variable is left unassigned, but used in the graph. This condition results in an error or warning, depending on the type of variable.

A design decision was made to produce an error when an unassigned filter variable is used in a graph. The error halts processing. While it is possible to construct an encrypted filter without search criteria, the search would consume resources without producing a result. This seemed inefficient, but in a strict sense, this decision precludes the ability to perform a null or empty search.

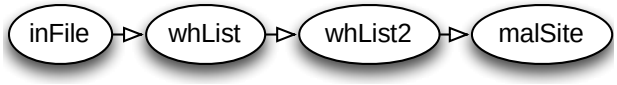
When a node variable is left unassigned, but utilized in a graph, a warning is issued. However, the variable, its declaration, and use in the graph remain. Packets pass through this unassigned node, when the packet filter is executed, and without modification. This is a design decision to support a no-operation (no-op). As packets traverse the packet filter, the no-op imparts little impact.

Packet Filter Validation: The semantic check for graph correctness assures that a path starts at a source variable and sinks to a filter variable. The in-degree of each node must be one, excluding source nodes which have an in-degree of zero. A filter variable cannot connect to another variable. This prohibits feedback loops in the packet filter. These rules are depicted in Figure 1 with three packet filters and their visual representations for clarification. The variables are from Fig. 1. The packet filter in Fig. 6-a for $myGraph$ is correct. The edge assignments start at a source variable, $inFile$, and sink to the

filter, $malSite$. The in-degree of each node is one. This is a valid packet filter. Fig. 6-b also has a correct syntax, specifies a source and sink, and has valid edge assignments. A warning though will be issued, since whList3 was not assigned any data. An error is additionally issued and processing halted since the filter variable, $c2IP$ was not assigned; notice that no indicators for this variable appear in Fig. 1. Fig. 6-c demonstrates an invalid path because the in-degree to the filter variable is two. Processing is halted.

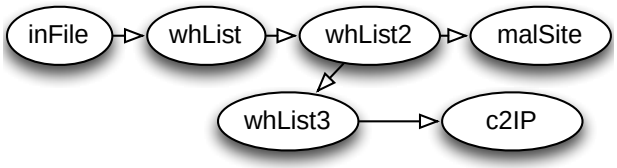
Last, we note that our graph representation is similar to the edge operator (edgeop) for directed graphs in the GraphViz language [11], [12], and the ability to select, direct, and reduce traffic volume is a common data processing paradigm. For example, the rwfilter command selects specific Netflow records in the SiLK tool suite, and the rwsender command creates a tee, directing data to multiple receivers [13], [14]. Our graph processing approach also shares functional similarities with Unix pipes.

```
myGraph ={
 inFile −> whList −> whList2 :: malsite
};
```



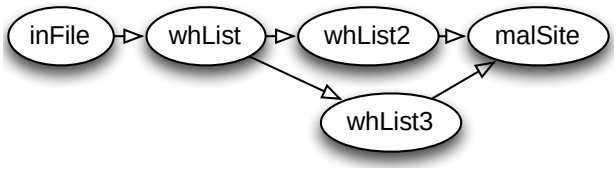
(a) Valid assignment and path

```
myGraph ={
 inFile −> whList −> whList2 :: malsite;
 whList2 −> whList3 :: c21P
};
```



(b) Valid path, but unassigned filter error

```
myGraph ={
    inFile −> whList −> whList2 :: malsite;
            whList −> whList3 :: malsite
};
```



(c) Invalid Path: In degree of 2 is invalid

Fig. 6. Three representations of a packet filter.

V. FACETS OF DESIGN

There are some design decisions that are reflected and apparent in the syntax. In this section, we will address the less apparent decisions, including how filter entries are referenced, a filter size is selected, and how a buffer is managed.

Encrypted Filter References: In the original private search system, entries in an encrypted filter were associated with a public dictionary of words [1]. Consider the contrived public dictionary D from Table I.

This is an example of the inference problem, forming conclusions from premises without authorization [15].

While the associated entries in the filter F are encrypted, precluding exposed interest in the keyword "sorbet", the overall interest in frozen desserts is evident. For this reason, a dictionary is assumed to be diverse, if not unabridged. The solution 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.

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, 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 detailed a method to eliminate the dictionary [4]. The value of a truncated cryptographic hash is utilized as an index into the filter, and the one-way property of the hash assures that the keyword cannot be inferred. A cryptographic hash will also exhibit a uniform response for all inputs, assuring that index values are generated uniformly. Our design utilizes this approach when referencing filter entries.

There is a drawback. That is, for a given hash function $H(x)$ and two words, w and w', hash values may collide $H(w) = H(w')$. This is not (generally) an issue for full-length cryptographic hash values, but the reduction of the hash space will introduce false positives.

Filter Expansion: To counter false positives, the filter size must be increased to an acceptable size. We use an expansion factor relative to the number of indicators. Unfortunately, this leads to a quadratic relationship in the size of the filter. A defender analyzing a thousand IP indicators across multiple intrusion sets, and in an environment which requires little or no spurious access to data, for instance at a rate of one in a thousand, produces a filter of a million entries.

Filter expansion is reflected in the design as the $.expand$ processing parameter for filter variables (Fig. 5). The parameter indicates the proportional expansion of indicators to determine the size of the filter. The parameter is not presented in the public form of the code so that the number of sensitive indicators cannot be immediately deduced.

Buffer Management: Some packets will match on an encrypted indicator in a filter, and will be returned as an encrypted result in the output buffer. Other packets will not match; the result of this non-matching search is an encrypted zero. Informally, an output buffer is not changed when a zero, in the plain text domain, is added to the buffer. However, the partner is unable to detect whether a packet matched and is stored to the buffer, or not. Since the result is encrypted, this leads to a conundrum. The partner does not know what was

stored, which buffer positions are available, or when to stop.

Current storage strategies thus, employ a randomized approach that is fundamentally based on the color survival theorem. Ostrovsky gives a formal presentation of this theorem [1], but intuitively, the strategy saves the result to a few randomly selected buffer positions. The occurrence of multiple copies will perpetuate the survival of at least one copy. A non-match has no effect on the buffer. In Ostrovsky's approach, the client recovers documents from the buffer by decrypting, and then searching for surviving documents. However, at the buffer's limit, some positions may be chosen multiple times, eliminating surviving copies. The recovery algorithm has a non-zero probability that all copies will be overwritten. Large buffers may minimize this condition, but this results in storage inefficiencies.

Subsequent research has sought to improve the document storage algorithm and therefore, document recovery rates. The research recognizes that a collision in a buffer position results in a linear combination of documents, and as a direct result of the homomorphic property of the Paillier cryptosystem. Information is not entirely destroyed, only obscured. Research has thus, qualified external structures, additional processes, and leveraged the redundancy of multiple copies to extract documents that were not recoverable in the original approach.

Bethencourt deconstructs the linear combination through a series of linear equations, but requires a second encrypted buffer [4]. The second buffer acts as a Bloom filter, when decrypted, and validates a document's membership in the output buffer. This knowledge can then be used to establish a system of linear equations to solve.

Danezis presents a (simple) iterative method: identify the singletons, calculate the positions that those documents were stored too, subtract the document value from those buffer positions, and repeat until no further singletons are discovered or the buffer is empty [5]. The use of the term, singleton was defined by Finiasz for this context [7]. The approach does require a function that duplicates document positions by the defender and partner. A (truncated) hash of incremented document values was suggested: $positions = \{H(d_i), H(d_i + 1), H(d_i + 2), \ldots, h(d_i + l)\}$, for each document d_i and for a pre-determined number of copies l. This algorithm replaces Ostrovsky's randomized approach for buffer position selection. Danezis's iterative method achieves full recovery when three document copies are utilized, and the total number of matching-documents inserted into the buffer does not exceed half the buffer size, $m = 0.5\ x\ |B|$.

As a design decision, our prototype utilizes Danezis's iterative method to recover documents from the buffer. The simplicity and acceptable recovery rates justified use. However, this still does not address when the provider should stop inserting results into a buffer. If we abide by the theoretical results, a buffer, twice the size, is returned after every "m" insertions. This is unacceptable when the majority of packets never match.

We resolve this issue with a processing parameter, $.trigger$. The parameter specifies the number of insertions as a multiplicative factor of the buffer size, for example 100, 1000, or 10000 insertions occurs before returning the buffer.

VI. Experimentation

We implemented a working model of the language. Our prototype consists of a lexical analyzer (Flex), a Bison parser, C++ code, and used *Mathematica* for the private search operations.

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.

One challenge remained. We needed an experimental dataset, and one that does not impinge on operational data. We acknowledge the pursuit of a standardize corpora for security research [16]. While Garfinkel's focus is on digital forensics, his scenarios include network datasets, including the "Nitroba University Harassment Scenario" [17]. This data set contains 91,144 IP packets.

Fig. 6 shows the execution of the private search detailed in this paper. The listing depicts a collaborative environment consisting of a defender and partner system, and shows the sequence of commands executed on each system. The search determines if any traffic in the Nitroba data set is destined to 69.25.94.22. A fictional organization with known malicious intent operates a web server, $www.willselfdestruct.com$, at this address. This fictional IP address is a sensitive indicator, and is not initially revealed to the partner.

The dataset consists of inbound and outbound traffic to fifteen private netblocks. Since the intent is to reveal malicious outbound traffic, the query from Fig. 1 includes a whitelist to ignore any inbound traffic; packets sent to 192.168.0.0/16 are dropped. In total, our prototype processed 45,776 IP packets.

The result of this search revealed 101 packets destined to 69.25.94.22. The defender gains situational awareness, and initiates a response action. If deemed appropriate, the defender may reveal the IP indicator and activity to the partner. We emphasize that all other indicators remain private.

Computation is bound by the number of modular exponentiations performed during the search. As depicted in Table III, query generation requires an encryption for each filter entry, and a result is obtained after decrypting the output buffer. The computational cost to create an encrypted filter is $O(|F|)$ encryptions, and the output is recovered in $O(|B|)$ decryptions.

Recall that each exponentiation (a result) is saved to the buffer three times (as per the color survival theorem.) Hiding this constant, the cost to save results to the buffer is $O(|T|)$ modular multiplications. Last, the computational cost for buffer construction can be performed off-line, and is not an overall factor.

The search however, must partition each packet to a set of values with a bit length less than the length of the modulus. The number of exponentiations is thus, hidden by another constant factor, but in general $B \approx F << T$ where T is a data set of packets. The computational cost of $O(|T|)$

exponentiations outweighs that of other operations in this system.

TABLE III
COMPUTATIONAL COST

Query Generation	
Filter Construction	$O(\|F\|)$ Paillier encryptions
The Search	
Buffer Construction	$O(\|B\|)$ Paillier encryptions
Packets Searched	$O(\|T\|)$ Modular exponentiations
Saving the Result	$O(\|T\|)$ Modular multiplications
The Result	
Buffer Decryption	$O(\|B\|)$ Paillier decryptions

VII. CLOSING REMARKS

We developed a high level language for private packet filtering (PPF). The language adapts the concepts of private stream searching, preserves the confidentiality of sensitive indicators, and is highly suited for cyber defense in a collaborative environment. Using our language, a cyber defender maintains situational awareness, controls which indicators are revealed, and shares threat details with collaborating partners.

We designed the language to be user friendly and to bridge the cryptographic constructs of private searching in a form applicable for a cyber defender. This paper presents the syntax of the language and demonstrates a private search for a sensitive IP address.

A greater breadth of searchable indicators is also possible. Additional filter types, such as $filter\ smtp\ subject$, $filter\ http\ user-agent$, $filter\ dns\ a-record$, etc. can be constructed as future work.

The language can also be adapted for new file scanners, anti-virus, and other defensive products that search for malicious content without revealing knowledge of the search.

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

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

defender$ ppf-recover -k kPrivate.key -r buffer.ppf -w partnerActivity.pcap
defender$ tcpdump -n -c 2 -r partnerActivity.pcap
01:03:43.729507 IP 192.168.15.4.35984 > 69.25.94.22.80: Flags [S], seq 3033670331, win 64240, options [mss 1460 …
01:03:43.819342 IP 192.168.15.4.35984 > 69.25.94.22.80: Flags [.], ack 2749676331, win 64296, options [nop,nop,TS val …
01:03:43.825871 IP 192.168.15.4.35984 > 69.25.94.22.80: Flags [P.], seq 0:526, ack 1, %win 64296, options [nop,nop,TS val …

The remaining packets from the tcpdump are not shown for brevity.
```

Fig. 5. A Demonstration of the Private Packet Filtering Prototype

APPENDIX A

The BNF Representation of the Private Packet Filtering (PPF) Language:

<ppfProgram> ::= <statements>
<statements> ::= <declaration> | <assignment> | <comment>

<declaration> ::= <keyDeclaration> | <graphDeclaration>
<keyDeclaration> ::= **key public paillier** <variable> { <bufferDeclarations> };
<bufferDeclarations> ::= <buffer> | <buffer>; <bufferDeclarations>
<buffer> ::= **buffer** <variable> { <filterDeclarations> };
<filterDeclarations> ::= <filter> | <filter>; <filterDeclarations>
<filter> ::= **filter in_addr [src|dst]** <variable>
| **filter port [src|dst]** <variable>

<graphDeclaration> ::= **graph** <variable> {<nodeDeclarations>} ;
<nodeDeclarations> ::= <node> | <node>; <nodeDeclarations>
<node> ::= **source [file|interface]** <variable>
| **whitelist ip [src|dst]** <variable>
| **whitelist port [src|dst]** <variable>

<assignment> ::= <varAssignment> | <parameterAssignment>
<varAssignment> ::= <variable> = <value> | {<values> | **include** "<fileName>"};
<parameterAssignment> ::= <variable>**.**<parameter> = <pValue>; | **.**<parameter> = <pValue>;
<fileName> ::= <text>
<pValue> ::= <decimalValue>
<values> ::= <value> | <value>, <values>
$numInputState<value> ::= **0x**<hexValues> | <decimalValues>
$edgeInputState<value> ::= <variable> -> | <variable> :: | <variable>
$ipInputState<value> ::= <ipAddresses>
<ipAddresses> ::= <ipAddress> | <ipAddress>, <ipAddresses>

<comment> ::= <publicComment> | <privateComment>
<publicComment> ::= #<text>
<privateComment> ::= ##<text>

<parameter> ::= **cwd** | **datamap** | **obfuscate** | **bufferSize** | **production** | **reuse** | **trigger** | **expand**

<variable> ::= <variableID>
<variableID> ::= <letter> | <variableID><letter> | <variableID><digit>
<ipAddress> ::= <netBlock> | <dottedDecimal>
<netBlock> ::= <ddigit>{1, 3}.<ddigit>{1, 3}.<ddigit>{1, 3}.0/24
<dottedDecimal> ::= <ddigit>{1, 3}.<ddigit>{1, 3}.<ddigit>{1, 3}.<ddigit>{1, 3}
<decimalValues> ::= <decimalV alue> | <decimalValue><decimalValues>
<decimalValue> ::= <ddigit>
<hexValues> ::= <hexValue> | <hexValue>, <hexValues>
<hexValue> ::= <hdigit>
<text> ::= <character> | <character><text>
<character> ::= <letter> | <ddigit>
<letter> ::= [a−z] | [A−Z]
<hdigit> ::= [0−9] | [a−f]
<ddigit> ::= [0 − 9]

REFERENCES

[1] R. Ostrovsky and W. Skeith, "Private searching on streaming data," in Annual International Cryptology Conference (CRYPTO'05), vol. 3621, pp. 223–240, 2005.

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

[3] 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, 2006.

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

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

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

[7] M. Finiasz and K. Ramchandran, "Private stream search at the same communication cost as a regular search: Role of LDPC codes," in Proceedings of the 2012 IEEE International Symposium on Information Theory, pp. 2566–2570, IEEE, 2012.

[8] J. Bethencourt and B. Waters, "Private stream searching toolkit." http://acsc.cs.utexas.edu/, 2011.

[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] E. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo, "A technique for drawing directed graphs," IEEE Transaction Software Engineering, vol. 19, no. 3, pp. 214–230, 1993.

[12] E. Gansner, E. Koutsofios, and S. North, "Drawing graphs with dot, the dot user manual," tech. rep., 2006.

[13] C. Gates, M. Collins, M. Duggan, A. Kompanek, and M. Thomas, "More netflow tools: For performance and security," in Large Installation System Administration Conference (LISA '04), the Eighteenth Systems Administration Conference, 2004.

[14] T. Shimeall, S. Faber, M. DeShon, and A. Kompanek, "Using SiLK for network traffic analysis," tech. rep., CERT Network Situational Awareness Group, 2010.

[15] M. Collins, W. Ford, J. O'Keefe, and B. Thuraisingham, "The inference problem in multilevel secure database management systems," in 3rd RADC Database Security Workshop, The MITRE Corporation, 1990.

[16] S. Garfinkel, P. Farrel, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," Digital Investigation, vol. 6, pp. S2–S11, 2009.

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