A Novel Approach to Detecting Covert DNS Tunnels Using Throughput Estimation

Michael Himbeault

University of Manitoba

Abstract. DNS tunnels represent a clear and common threat to network security by bypassing existing security and administrative controls. The ability for DNS tunnels to transmit arbitrary data over conforming DNS packets makes them difficult to detect. This work proposes a novel approach to detecting DNS tunnels that is built on entropy and operates on commodity hardware. The proposed approach is compared to several others from literature and outperforms both in computational speed and detection accuracy.

Keywords: intrusion detection, APT, DNS, entropy, NIDS, network analysis

1 Introduction

Control of the data that moves between hosts on a computer network is vitally important in order to be able to enforce security policies and protect sensitive information. To this end systems such as firewalls, proxies, and content filters are put into place in order to monitor and control network traffic. Covert channels are utilized to circumvent these control mechanisms for purposes that range from benign[22] to malicious. This work focuses specifically on DNS tunnels, however other types of covert channels do exist.

Detecting a DNS tunnel effectively on a busy network link becomes an exercise in discrimination. Since there is such a wide variety of network traffic that is generated on a busy link, there is generally no simple definition of *normal* for a particular class of traffic, including DNS.

A highly sensitive and specific detection of DNS tunnels on a busy network link is an important problem in the arena of network security as it enables administrators to block or otherwise control these potential sources of compromise. Even if an enterprise does not have sensitive information that needs protecting, DNS tunnels should still be blocked in order to prevent malware from communicating[16]. Because DNS tunnels can be used for arbitrary communication, they can be used as command-and-control channels for botnets or any other malicious system that relies on data transmission.

In addition to the malicious uses for DNS tunnels, there are other uses that could be classed as benign. One use of this is the DNS-interface to the NIST National Software Reference Library[22]. Queries to a specific

zone can include hashes of files on a system in order to determine the validity or purpose of said file.
Because signature based schemes are not effective in detecting DNS tunnels in a zero-day situation, another method is required. Analysis based on character frequency is proposed in[6] by Kenton Born with similar approaches proposed by several other authors (see Sect. 3.3 for more details) which uses character frequency analysis under some assumed properties.
The proposed method operates under the assumption that DNS tunnels move more data than a normal domain but do not necessarily do it by moving more bytes than a benign domain. This distinction is important since, on a busy network, a large content or service provider such as Google, Amazon, Facebook or Twitter may make up orders of magnitude more DNS traffic by byte count than a DNS tunnel. A brief treatment of the effects of DNS caching on the number of bytes observed in repeated query strings is given in Sect. 5.1. The proposed method uses the information measuring properties of entropy in order to estimate the amount of information transmitted.
This proposed detection methodology is shown to detect DNS tunnels in as few as ten packets (as shown in Sect. 6.1) and continues to be robust in highly hostile detection environments such as those that contain a great deal of non-tunnel traffic as well as benign uses of DNS for transmitting arbitrary data. Detection performance on commodity hardware is shown to scale to greater than two gigabits of UDP port 53 throughput per second in a performance-oriented C++ implementation. This indicates that this methodology does not sacrifice performance for detection accuracy and remains practical for monitoring very large networks.
Section 6.3 compares the processing performance of various approaches from the literature to the proposed method, demonstrating that the proposed method has better performance in a great many cases. Section 7 compares the detection performance of the proposed approach to its peers, demonstrating that in addition to improved computational performance, the proposed approach also provides superior detection performance.
By improving upon both areas over existing methods from the literature, the proposed method constitutes a worthwhile contribution to the field of DNS tunnel detection.

2 Background

2.1 Covert Channels

Covert channels are methods of communication that use non-standard means of communication typically for the purpose of evading detection and/or blocking by the existing security infrastructure. Covert channels may utilize portions of an existing protocol[5] or communication channel, or they may find ways of transporting information utilizing a completely new medium. An example of the latter is called a *timing channel*[34], which can utilize the timing between packets to convey information. A

timing channel carefully controls the timing between packets sent to a remote server to encode information, thereby utilizing a method of communication that is not utilized by any standardized protocol or communication method.

2.2 DNS Tunnels

DNS tunnelling is the method by which arbitrary data is transferred over the same channels as DNS. DNS tunnels come in one of two primary types: raw, or conforming.

Raw DNS Tunnels Raw DNS tunnels do not attempt to mimic or conform to the DNS specifications, and simply attempt to utilize the fact that UDP port 53 is often left relatively uncontrolled in firewalls. Raw tunnels attempt to exploit this by transmitting arbitrary traffic using UDP port 53 packets with arbitrary payload [1]. This is the most efficient exploitation of the ubiquity of DNS as it incurs the lowest amount of overhead, both computationally and in terms of network throughput. The trade off for this efficiency is that it is the least conforming and the most likely to get stopped by either a firewall or a proxy. In the situation where all DNS queries are forced to be proxied through a dedicated DNS server, raw DNS tunnels will fail to operate as expected. This is because when the UDP port 53 traffic is redirected to the proxy, the DNS server will attempt to interpret the arbitrary payload as a DNS packet and will likely fail. When it fails, it will drop the packet thereby preventing all raw UDP port 53 communication. Because these types of tunnels are effectively blocked by standard firewall and proxy practises, detection of these tunnels is not considered in this work.

Conforming DNS Tunnels Conforming DNS tunnels produce DNS packets that conform to all appropriate specification and RFC documents and, as far as any DNS server is concerned, the traffic generated is valid DNS traffic. These tunnels incur the highest computational and throughput overhead, but have the advantage that detecting and blocking them is a very difficult process. The detection of this type of DNS tunnels is the topic of this work. This type of tunnel is capable of operating in almost any environment, even those with very strict firewall and proxy policies. Because this type of tunnel operates in very hostile (to the operation of the tunnel) environments, detection of this type of DNS tunnel is of interest to all levels of government and industry.
Conforming DNS tunnels operate by embedding the data for transmission into the query string and response, requiring a modified, non-conforming, server on one end of the connection and a piece of software on the client end. Typically these types of DNS tunnels have one endpoint that is controlled by the tunnel user, with that controlled endpoint running dedicated server software. The client and server software

[1] Iodine demonstrates this behaviour when operating in its raw transport mode

are responsible for transforming arbitrary information to and from DNS queries and responses. The precise details of how the translation is done between DNS and the raw data depends entirely on the implementation.

DNS Tunnel Software Some existing DNS tunneling software currently available is OzymanDNS[30], Iodine[15], Dns2tcp[13], DNScat[29] and DeNiSe[17], and PSUDP[4]. Each of these have slightly different operational characteristics, but they all aim to do the same thing which is transmission of arbitrary data over DNS.

3 Review of the State of the Art

The solutions that exist to date to detect DNS tunnels generally make very little use of complex and static signatures, but rather attempt to exploit a characteristic trait or property that the DNS tunnel will exhibit. If a tunnel can be crafted to not exhibit that feature, then those detection strategies will normally fail in their detection.

3.1 General Covert Channel or Anomaly Detection Research

All of the work in this section is aimed at detection of general covert channels, and does not specifically focus on DNS tunnel detection.
(Browne, 1994)[8] establishes an entropy conservation based approach for testing the completeness of general (that is, not specific to DNS) covert channel analysis and detection methodologies. (Shaffer, 2008)[35] Proposes a Security Domain model for assessing the surface of a piece of software for exploitable covert channels. (Ray, 2008)[32] proposes a protocol for use in a covert channel that incorporates stealth, low overhead, data integrity, data confidentiality, and data reliability. This protocol can be used on top of any other covert channel transport method (ICMP, IP, HTTP, DNS, etc. . .)
(Horenbeeck, 2006)[21] discusses, briefly, DNS tunnels and their implications and includes a mention of proxying DNS requests is given as a potential solution but without examining the other options. The rest of the paper discusses the risk management and policy based mitigations that can be applied to covert channels in general. (Moskowitz, 2003)[25] investigates the link between anonymity and covert channels. (Newman, 2007)[26] discusses covert channels in a broad sense, examining the various types of covert channels along with the relationship between covert communication, cryptography, steganography and secrets. (Okamura, 2010)[27] discusses a fascinating type of covert channel for communication between virtual machines that share a physical host involving the manipulation of host CPU load. Tunnel Hunter[18] is an application that aims for general covert channel detection over a variety of tunnelling communication channels.

3.2 Non-DNS Related Research

(Bauer, 2003)[1] discusses a new type of HTTP-based covert channel that adds the unwitting web browser application to the anonymity set. (Borders, 2004)[2] discusses a method of detecting data egress using HTTP-based covert channels. (Cabuk, 2004)[10] and (Cabuk, 2009)[11] discuss the design and detection of IP (Internet Protocol) based covert timing channels. (Gianvecchio, 2007)[20] discusses an entropy-based approach to detecting covert timing channels on the Internet based on their effect on the original process' entropy properties.

3.3 DNS Covert Channel Research

The SANS Institutes's InfoSec Reading Room published a report on the design and detection of DNS tunnels[19]. The report covers a very wide variety of topics including background information, tunnel-specific information, technical information, existing applications, detection techniques, detection implementations, and a sample detection scenario. This report is exceptionally good reading as a primer on the topic.
The sample detection scenario employs an analysis technique very similar to the technique that will be outlined in Sect. 5.
(Karasaridis, 2006)[24] proposes and evaluates mechanisms that use network flow data[2] to detect DNS anomalies including cache poisoning and tunnels. The authors are able to observe considerable changes in their cross-distribution entropy measurement during the onset of the Sinit virus in their real-world data. This approach is discussed in additional detail in (Roolvink, 2008)[33].
(Born, 2010)[3] discusses a way of using javascript in a web browser to exfiltrate data from a network, while [5] discusses a novel way of crafting a DNS tunnel that exploits the nature of a DNS packet and the ability to create unused space in the packet in which arbitrary data can be stored. [6] discusses a method of detecting DNS tunnels by examining character and n-gram frequencies in the names that are being queried for. [7] demonstrates the effectives of data visualization when attempting to detect a DNS tunnel using a custom visualization engine using the character frequency analysis proposed in [6]. If a DNS tunnel can be crafted such that its character frequencies are distributed sufficiently close to those of legitimate DNS names, then it is possible to hide a DNS tunnel from this type of analysis.
(Butler, 2011)[9] demonstrates a way of quantitatively analyzing covert communication channels with particular focus on DNS covert channels. It proposes a *codeword mode* of communication over DNS where a specific lexicon is chosen that allows the two endpoints to communicate with each other. Each word in the dictionary has a particular meaning[3] that is understood by both endpoints.

[2] Flow data is a way of digesting network packet data into information per communication, stream, or (in the case of UDP since there is no inherent concept of a stream of interrelation of packets) temporally contiguous collection of packets.

[3] The words can represent binary information, or can represent higher level constructs such as commands in the context of a botnet or piece of malware.

(Romana, 2007)[14] discusses their analysis of DNS data on a large campus network using the output of a DNS resolver's query logging as their input. Digestion of the large query log file is done with standard Unix utilities and logic available on almost all Unix-based systems. The authors estimate the entropy of the source IP address (of the DNS query) and the queries themselves, and perform analysis based on that digestion.

(Thomas, 2011)[37] proposes and evaluates the efficacy of a Field Programmable Gate Array (FPGA) based solution for detecting malicious DNS packets on a high throughput network link. The analysis performed on the DNS packets in order to determine their validity is done via a signature-based system where the DNS query is hashed, and the hash is compared to a blacklist of domains that are disallowed based on the network policies.

(Dietrich, 2011)[16] examines the use of DNS for command and control of botnets based on the reverse engineering of the *Feederbot* botnet application. Based on the lessons learnt from Feederbot, the authors applied their methods to other real-world traffic and detected other botnets that also use DNS as their command and control medium. The first class of approach used by the authors is very similar in theory to the character frequency analysis proposed by Born[6]. The authors also propose the use of behavioural analysis on data and statistics gathered from the aggregate of several packets to estimate the persistence of DNS queries as well as the amount of data moved over DNS by each host on the network.

(Paxson, 2011)[28] is a slide deck that discusses the author's searches through large campus networks for DNS tunnels in the wild. The author proposes an approach for detecting DNS tunnels that is very similar to the method proposed in this work in that it examines the approximate amount of data transferred per domain and/or subdomain. The author, instead of utilizing entropy measures, makes use of the utility *gzip*[4] to estimate the amount of data moved under a domain in a given collection of queries.

jhind[23] gave a presentation at DefCon 17 that discusses the use of artificial neural networks to identify DNS tunnel traffic. The author proposed that the neural network operate on the euclidean distance between the various queries to a particular subdomain, treating the queries as vectors in higher dimensional Euclidean space. The author successfully detected DNS tunnels as produced by several software packages (Iodine, Ozymandns and Dns2tcp) using the described approach.

Static signatures exist for at least three common network anomaly detection engines (Snort[12], Proventia[31], and TippingPoint[5]) engines, with others likely offering similar functionality.

[4] *gzip* is a compression utility that is used to compress input streams such as archives or other files.

[5] TippingPoint does not make information about its filters available publicly, however a personal correspondence with a TippingPoint user revealed that filters 9932 and 9938 trigger on the application data contained in DNS packets generated by Ozymandns.

4 Problem Statement and Evaluation Criteria

4.1 Detailed Problem Description

DNS tunnel detection is a complicated task made more difficult by the fact that DNS tunnel traffic can appear to be completely legitimate network traffic that conforms to all standards and restrictions. It need not violate any established standards or conventions, which makes it difficult to detect against the background of normal DNS traffic based on testing for violations.
This property of DNS tunnels makes them a particularly effective transport mechanism when data exfiltration or network control circumvention is the end goal. For this reason an efficient method of detecting DNS tunnels is required that can effectively detect a DNS tunnel against normal DNS traffic with a low false-positive rate and that must not be susceptible to existing methods of circumvention.

4.2 Solution Evaluation Criteria

The objectives that must be met for an approach to have successfully solved the problem posed are:

- Successfully discern tunnel traffic generated from existing tunnel applications and theoretical tunnel traffic (built using additional parameters to attempt to hide from known detection methods) from a baseline of normal traffic.
- Be resistant to known obfuscation methods compared to existing detection methods.
- Be able to operate at high speed on general purpose, easily obtainable hardware.

The proposed approach will be evaluated against these criteria to determine whether or not it can be considered an improvement on the state of the art for this type of detection.
Item 1 will be validated by comparing the chosen approaches against the proposed approach in a relative scoring fashion. Methods will be compared to their peers for relative detection performance, and improvement therein, in the various test scenarios. Methods will be scored based on false positive rate, with lower rates being more desirable. Successful approaches must be highly specific with a very low false positive rate in order to prevent overloading alerting systems with unhelpful information.
Item 2 will be tested using a next-generation tunnel and referred to as `next-gen`, that simulates what DNS tunnelling applications may look like in the future. The primary difference is that output of this tunnel is set to match the character frequency distributions of normal DNS queries. Due to the implementation details of the next-gen tunnel, there is no server-to-client transfer direction for that tunnelling application.
Item 3 will be tested by comparing the approaches when implemented on a common Python framework to produce a level playing field of performance.

5 Proposed Detection Method

The method proposed in this work examines the information theoretical properties of the DNS queries to each domain, thus retaining the flexibility to filter and alert per domain as opposed to more generally on the set of all DNS queries. The tools developed to test this approach utilize full packet data for its analysis, but can be modified to use name server query logs (as were used in [14]) or other sources of query information. The prototype C++ software is easily capable of running at greater than gigabit speed on inexpensive, off the shelf hardware making this approach practical and uncomplicated to deploy on smaller networks or in resource constrained situations. Deployment in large environments is similarly straight forward.

5.1 Theoretical Basis

Assumptions The detection approach proposed in this work makes certain assumptions about the nature of DNS tunnels in order to effectively detect them. The primary assumption made is that DNS tunnels move more data than a normal DNS subdomain, with a very particular meaning of *data* that goes beyond simply counting bytes or the number of queries. The concept of the amount of data moved under a DNS domain involves considering the entropy of the queries as a whole, and not the characters that make up a query. The list of assumptions follows:

- DNS tunnel applications use the queries themselves to transport data from the client to server.
- This mechanism will force there to be more unique queries per domain (or subdomain) proportional to the amount of unique data transferred from the client to the server.
- In a server-to-client transfer of data, there will still need to be acknowledgements sent from the client to the server, with the acknowledgement data encoded in the query string.

The primary assumption, in the language of DNS queries, is that DNS tunnels will cause more unique DNS queries to a domain (or subdomain) than benign traffic. If a DNS tunnel is able to construct its network traffic in such a way that this assumption is no longer true, then the proposed approach will be ineffective in detecting it.

Theory In a large Internet provider network, it is possible that there could be many copies of the same DNS query - say *google.com* - each of which would count towards the total number of bytes or queries transferred to/from that domain. If a naive approach to detection is used, such as counting bytes or queries to/from a domain, then this kind of repetition will have a detrimental effect on the metric calculated for some popular domains.

In order to work around this, the proposed approach takes a different stance on unique data and instead uses entropy to measure the amount of data moved in the queries to a domain or subdomain. By considering queries as atomic objects, and maintaining a tally of the queries to a

domain, and their counts over an interval, a probability distribution function (PDF) is generated. By computing the entropy of this PDF, a basic measure of throughput is achieved. However, since there is value in the capturing the length of the queries that were sent (since longer queries are moving more bytes than shorter ones), the entropy is multipled by the average query length (in bytes) over that interval. This metric, which will be referred as the Domain Length-Weighted Entropy (DLWE), is the primary mechanism through which the proposed approach detects DNS tunnels.

With this new measure of data, the approach considers intervals of time and computes the amount of data estimated to be moved by each domain over that interval. By sorting all domains by their data throughput, the heavy-hitters can be examined in each time interval with white-listing preventing many of the top benign contenders from causing alerts.

As will be shown in Sect. 6.3, this approach is capable of processing packets nearly as fast as a naive approach with equivalent or better detection performance which is shown in Sect. 7.

The Effect of DNS Caching on Detection Effectiveness Because the packet capture was done in an environment where a large portion of the clients use one of only a few different DNS servers, the effects of caching will cause the naive approach to have far better detection performance than if this were not the case. If this were an environment such as a large ISP or Internet backbone where such DNS caching were not present then the naive approach would have far different detection performance and characteristics.

In order to grant some possible context to the effects that DNS caching has had on the naive method's performance, a simple comparison is offered. Data was taken from a home network serving five computers and smartphones, with DNS traffic logged over a twenty-two day period to match the time frame of the capture for real world data. Over that twenty two day period, *www.google.com* was queried 5645 times compared to the 268842 times the same query was seen in the real world traffic capture. It is important to modulate these values by the number of hosts that the real world traffic represents, which is on the order of approximately thirty thousand, or six thousand times the number of hosts the home network was supporting.

Approximately scaling the home network by a factor of six thousand results in an estimated three million queries to *google.com* occurring in the real world traffic, of which only a twelfth actually appeared in the capture due to DNS caching. This is only one common domain name, so applying similar logic to other domains, it is easy to see that in these scenarios, the normal curves would take on a very different shape, easily obscuring tunnel traffic for low throughputs.

A small sample of data was collected from Merlin's[6] caching DNS servers which represents *every* DNS query made of them, regardless of whether those queries were served from the cache or not. This type of DNS sampling presents a high load on their instrumentation infrastructure, and

[6] Merlin is a small ISP that supplied the primary data set for the testing later.

so is only reasonable for this type of comparison. The logs obtained from their servers span four hours from 1200 to 1600 on a Thursday afternoon. Figure 1 shows the effects of query caching on the repetition counts of queries in networks. The home network, mentioned above, as well as Merlin's network are represented in order to demonstrate scale. The horizontal axes represents the count of queries, normalized as a proportion of the maximum count. Because the compared networks and captures have vastly different scales, with the packet capture spanning weeks and the query log capture only spanning hours, it was important to normalize the data sets in order for a direct comparison to be possible.
Each plot was built by tallying the DNS queries in each capture, sorting by count, and then dividing by the largest count. The y-value on the plot then represents the proportion of unique queries that had a count greater than xM_d where x is the horizontal value and M_d is the maximum count for that particular dataset.

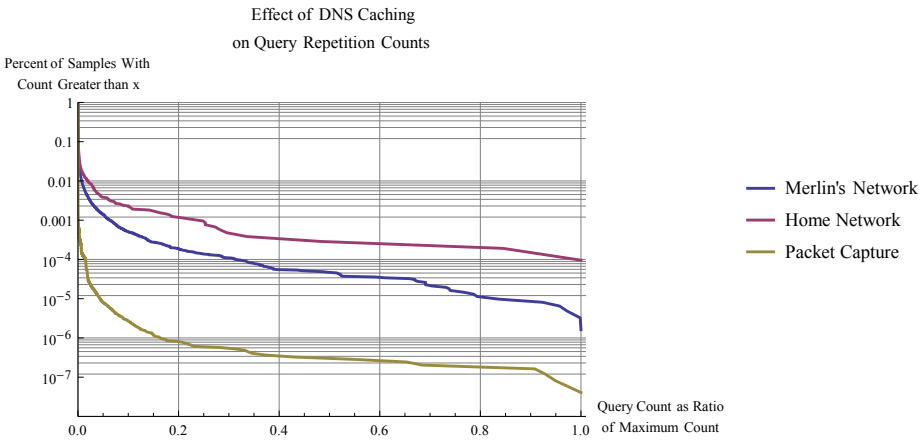


Fig. 1. Shows the trends for DNS query counts in networks with and without caching. The axes are normalized to account for the fact that the real world data has much more data than the home network. By normalizing the query counts, it is possible to perform a direct comparison.

As can be seen from the plot, which is logarithmic in the vertical axis, it is evident that, on average, the queries in an uncached environment occur over two orders of magnitude more frequently than in a cached environment. This increased occurrence in an uncached environment would have a strongly visible impact on the naive metric's ability accurately detect DNS tunnels. The additional increase in the home network over the query logs from Merlin's network is due to the fact that there may be on-premise customer caching DNS servers on Merlin's network that are not, and can not be, accounted for in the logging obtained from Merlin. As can be seen from the proposed method's underlying mechanisms, the DNS caching actually provides a *more pessimistic* detection environment

compared to the uncached environment. Unlike the naive method, whose detection performance results will not be applicable in an uncached environment, the proposed method can be expected to perform better in an uncached environment than in the testing in Sect. 8.
Born's method will also improve in detection performance in an uncached network, since it will cause normal traffic to have a character distribution more heavily skewed away from uniform, resulting in larger metrics and better certainty. Paxson's metric, however, will suffer in an uncached environment due to the fact that additional data, even if it is highly repetitive, will still increase the metrics of normal traffic and may obscure some low throughput DNS tunnels in the process.

6 Detailed Testing Methodology and Processing Performance

A collection of tests were run on several detection methods, demonstrating the performance of this work's proposed method when compared to existing methods from the literature. The detection methods chosen for comparison are

- The n-gram detection proposed by Born[6] because it is well defined and was the most prevalent approach found during the literature search. Additionally, the approaches built on this technique claim reasonable success in detecting DNS tunnels.
- The use of *gzip* on domain and subdomain packet data as proposed by Paxson[28] because it involves looking at data that is very similar to the proposed approach, but makes use of different methods for measuring the data throughput.
- A naive approach that simply measures the volume (in number of characters in the query strings) of packets per domain/subdomain in an attempt to illustrate that simple volume of queries is a highly inadequate approach, and that more sophisticated approaches can perform considerably better.

The collection of methods was put through several tests in an attempt to demonstrate their performance in average (using existing implementations) and worst case scenarios. All of the tests involving traffic generation were performed in a virtual environment of two Linux-based virtual guests directly connected via a virtual network on a single physical host. The tunnelling applications were communicating between the virtual hosts, transmitting content from the high entropy source `/dev/urandom` under Linux.

6.1 Situational Performance Goals

Determining a Baseline Through cooperation with Merlin, an educational Internet Service Provider (ISP) in Manitoba, DNS traffic was collected over a period from Thursday November 4 2010 until Friday November 26 2010. The hosts responsible for the DNS traffic observed

include several dozen school divisions totalling tens of thousands of individual computers. The capture includes just over one billion packets destined to, or sourced form, UDP port 53 (the standard DNS port).
This captured traffic will be used to determine a baseline distribution to which the metrics produced on isolated tunnel traffic can be compared. This baseline will provide context in order to determine if a method is able to detect a tunnel with sufficiently high certainty.
It is assumed that the incidence of tunnels in this baseline traffic is sufficiently low that it can be discounted. This assumption may not be perfectly accurate due to to reasons indicated in the introduction. Because some security vendors use DNS to transmit some of their information, these transmissions are in essence a DNS tunnel and so will represent a certain portion of the real world traffic. The effect of tunnels present in the real world traffic given the assumption that there are none will result in a more pessimistic environment for testing. Since some of the traffic that lies further out than the synthesized traffic may in fact be a tunnel, a portion of the false positive rate that the detection approaches will suffer may actually be due to the classification of existing traffic as a tunnel, and not due to misclassification. By making the assumption that no tunnels exist in the real world, however, the false-positive rates given here represent an lower bound and so should transfer well to other real-world scenarios.

Existing Implementation Detection This test will involve the two hosts communicating at varying throughput rates using the chosen existing DNS tunnel implementations. The throughput rates will scale from as little as several bytes per second, to several megabytes (or as high as the tunnel applications can support) per second. The wide range of throughputs used is done to give an indication of how the detection methods scale with tunnelled throughput.
The existing implementations chosen for testing in this section are Iodine[15], DNScat[29], and DNS2TCP[13]. Iodine is chosen due to the fact that it provides a full VPN solution without additional work by the user. DNScat is chosen due to being written in Java and so runs on multiple platforms[7] without the need for a compiler or other complex dependencies that the user must obtain. DNS2TCP is chosen since it does not require root access, and is written in C indicating potentially better throughput than other mechanisms.
The detection results of each of the approaches on the existing tunnelling applications is of value to those wishing to identify uses of DNS tunnelling currently in the wild. Instead of generating a large number of distinct events that will be detected (or not) resulting in a ROC plot, rather a much smaller number of prototypical events were produced. Details are given in Sect. 7 regarding the statistics of the metrics from these tests and how the detection methods will be evaluated. In essence, detection methods will be scored for false-positive rate and ranked against each other in order to determine a relative rating and ranking. The relative

[7] DNScat will run on any platform that supports Java 1.4 or later[29]

performance comparisons allow for a more contextual performance analysis to be done between two methods that perform very similarly.

6.2 Tunneling Application Throughput

The tunnelling applications used during the evaluation were subjected to different rates of traffic in both client-to-server and server-to-client directions. For each tunnelling application, sixteen captures were performed at each of the following target throughput rates (in bytes per second) in each direction (client-to-server and server-to-client where applicable):
10, 25, 50, 100, 250, 500, 1000, 2500, 5000, 10000, 25000, 50000, 100000, 250000, 500000, 1000000
The rates stop at ten bytes per second for practical reasons. It is considered reasonable to assume that tunnels with a throughput rate lower than this are ineffective at transmitting sufficient data to be practically useful in many situations. As will be seen in Sect. 7, throughput rates lower than ten bytes per second would quickly become lost in normal DNS traffic for even the most discerning detection methods.
Due to implementation details of the applications, and of the next-gen tunnel, not all applications were able to transmit traffic at the target rate. Figure 2 shows how the various tunnelling applications responded to various input rates, plotting their actual rate of ingestion of input and the observed number of characters of query output they generated on the network. Note that for the lighter coloured graphs showing the observed output, the vertical axis is representing a value equivalent to the naive metric. The next-gen tunnel was not tested in a server-to-client direction since it does not implement that functionality.

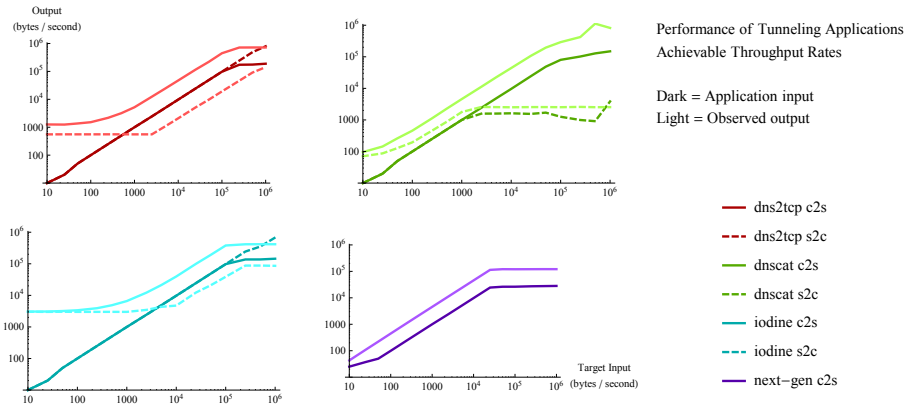


Fig. 2. Shows the scaling behaviour of DNS tunnelling applications as the input rate is scaled up. Note that not all applications are capable of transmitting data at the rate they are given data, which is visible as a plateau on the right-hand-side of each plot.

6.3 Detection Method and Python Interpreter Processing Performance

Python has several interpreters available freely in addition to the standard interpreter (for this discussion, the standard interpreter will be referred to as Cython). A notable alternative, called PyPy, is a Python interpreter written in Python itself that contains just-in-time compilation (JIT) mechanisms that Cython does not have. PyPy can offer an order of magnitude or better speedup[36] in some workloads. Figures 3, 4, and 5 show the performance of the various detection methods over aggregate tunnel and real-world data on both PyPy and Cython.

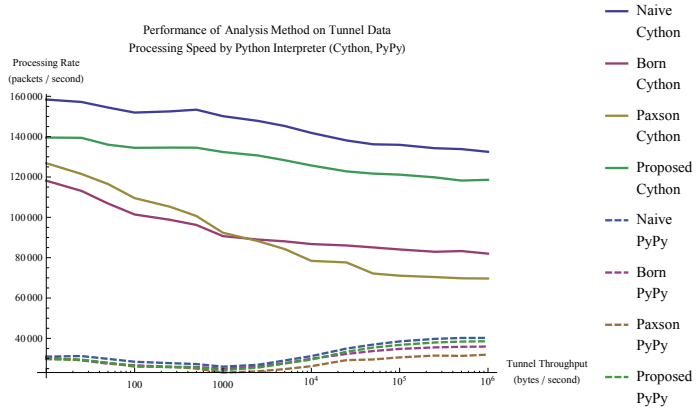


Fig. 3. This plot shows the performance of the detection methods and Python interpreters over the aggregate tunnel data. The output is packet processing rate as a function of the target input rate (rate at which the tunnels are transmitting traffic). As the target input rate increases, the structures involved in the methods get larger in a non-linear way, resulting in longer operation times and slower performance.

Figure 4 shows the performance over time of the different detection methods and Python interpreters as more packets are ingested from real world data. Observe that as time progresses, the methods get progressively slower, likely due to inefficiencies in the interpreter and/or method.
Also unlike the aggregate tunnel detection performance, the methods when run under PyPy perform *far* better, even surpassing the Cython counterparts in at least one case. Note that the Paxson's approach when run under PyPy shows decreasing performance over time while no such decrease for larger times is observed when run under Cython.
Figure 5 shows the same data, but with a limited scale allowing early-time behaviour to be examined. Note that the ramp-up of PyPy's JIT components is observed in the very early time followed by a very consistent processing rate.
Figure 6 attempts to represent the performance of the various detection methods and Python interpreters as more tunnel data is moved through

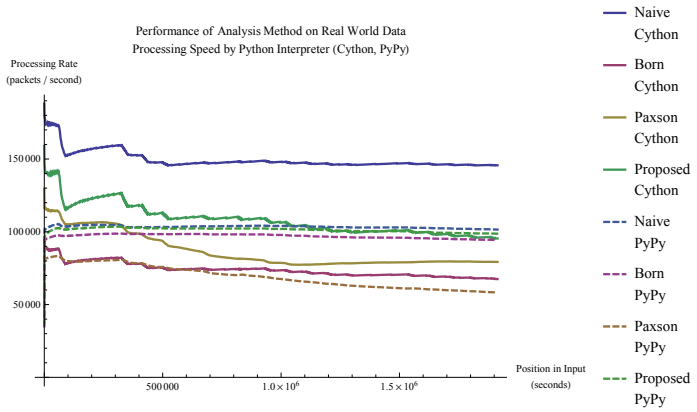


Fig. 4. This plot shows the performance of the detection methods and Python interpreters on real world DNS traffic. The output is packet processing performance as time progresses and more packets are processed by the script. As more packets are fed into the script, inefficiencies in the methods and/or the Python interpreters themselves become observable in the degradation of performance.

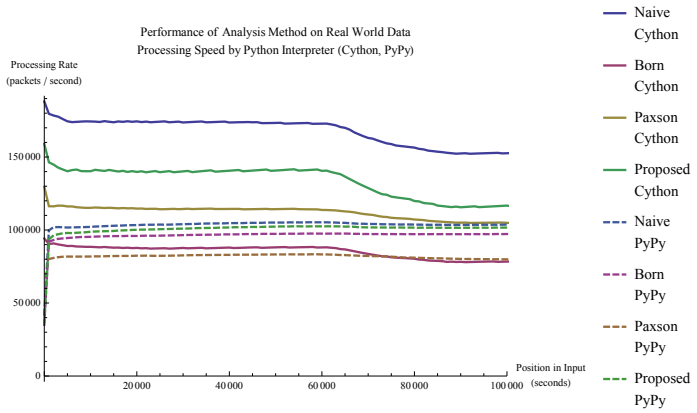


Fig. 5. This plot is identical to Fig. 4 but restricts the time displayed to the first one hundred thousand seconds. The 'spool up' of the JIT portion of PyPy is noticeable in the very early time-scales.

them per interval. Their horizontal axes are the actual data input rate (see 6.2), and the vertical axes indicate the processing rate (in packets per second).

In the above mentioned figures the legend requires some additional context. The plot legends contain labels of the form *dns2tcp c2s Cython* which contains three distinct pieces of information. The first word indicates which tunnelling application being one of DNS2TCP, DNSCat, Iodine, or the next-generation simulated application which is indicated by a name of `next-gen`. The second word indicates whether the data being moved over the tunnel is being transferred from the client to the server (`c2s`) or from the server to the client (`s2c`). The final word indicates which Python interpreter is being used.

There are fourteen lines on each figure, each corresponding to a Python interpreter, tunnel application, and data transfer direction triple. The solid lines correspond to runs made under the Cython interpreter and dashed lines indicate the use of the PyPy interpreter.

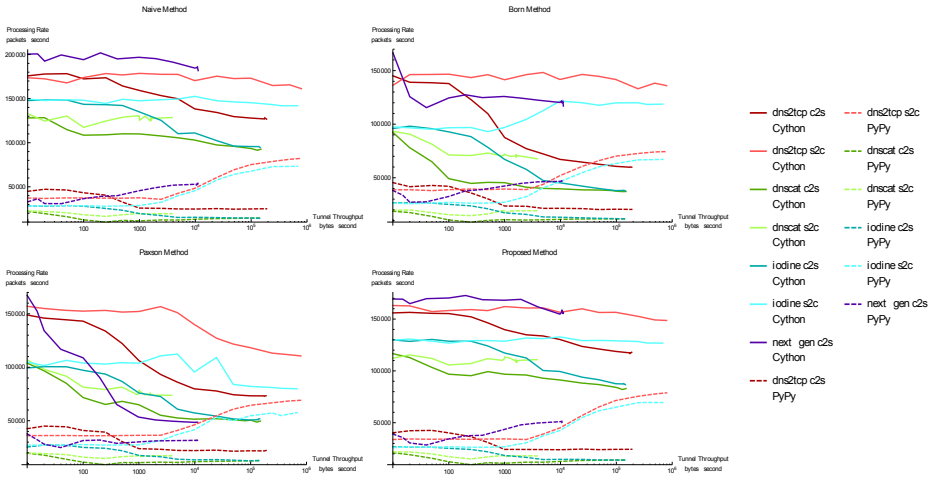


Fig. 6. Performance of all methods on separated tunnelling application data, showing processing rate as a function of input rate.

The proposed approach shows performance characteristics and trends that match the naive approach far more closely than either of the other two approaches. There is minimal degradation in performance for most of the tunnelling applications, and almost all of the samples under the Cython interpreter are above one hundred thousand packets per second. It is instructive to observe that PyPy's performance overall is considerably lower than Cython on tunnel data, but as is shown in Fig. 4, this is not the case on real-world data.

6.4 Processing Performance Conclusion

As has been shown, the detection methods were tested on both real world and purpose-generated tunnel application traffic as well as on two different Python interpreters and were instrumented for their processing performance.
When evaluating the performance of the detection methods on real-world data, in all cases the naive method is the fastest, followed by the proposed approach, Born's method, and Paxson's method in order.
When operating on tunnelling application traffic, the average performance of the methods (averaged over all tunnels and cases) can be seen in Fig. 3 where the naive and proposed methods both perform well, maintaining processing rates in excess of one hundred twenty thousands packets per second. Born's and Paxson's approaches both suffer severe degradation of performance as throughput increases, resulting in final processing rates well below one hundred thousand packets per second.

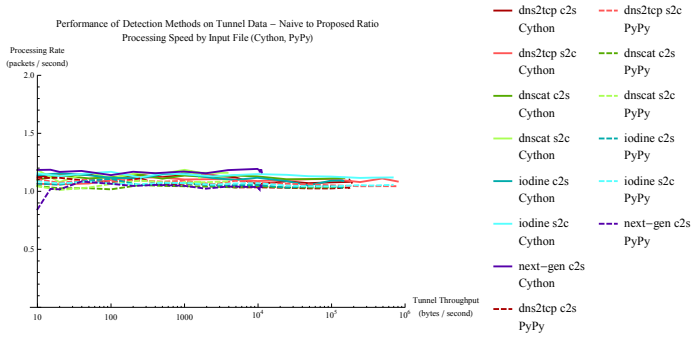


Fig. 7. The ratio of the performance of the naive method to the proposed method on separated tunnelling application data, with the vertical axis showing the speedup of the naive method over the proposed method.

When the ratio of the performance of the naive method to the proposed method is examined, a clustering very close to a fixed value is observed as in figure 7. This indicates that much of the performance degradation, and potentially other performance characteristics, of the two methods are dominated by the common scaffolding and/or the Python interpreter as opposed to the underlying methods or their implementation.
Through this examination, it has been shown that the proposed method out-performs both methods from the literature by a considerable margin and comes very close to matching the naive method in performance in many cases.

7 Tunnel Detection Evaluation

In order to obtain the metrics used in this section, tunnel application and real world data was separated into adjacent ten second windows for

processing. The distribution of metrics across these windows is computed and used to produce the plots for real world data as well as indicative representative values for tunnel applications.
When examining the distribution of metrics produced by tunnel applications, it was discovered that the metrics were clustered extremely tightly around the mean. The relative standard deviation for the various tunnel applications, transfer directions (server-to-client or client-to-server) are given as a function of input rate for each of the detection methods in Fig. 8 . These plots show that the clustering around the mean for these metrics is so tight that a standard receiver operating characteristic (ROC) plot would be of little additional value since the true and false positive rates are dependant on the value ranges that the metrics take. Since the range of values that the metrics take on is so limited, the differences between a low and high percentile in true and false positive rates would be minimal.

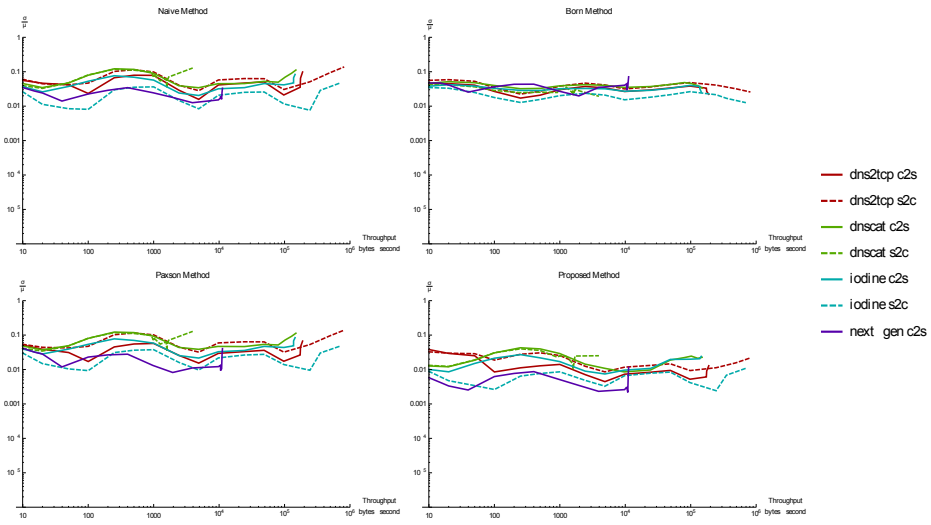


Fig. 8. Relative Standard Deviation of Metrics - All Metrics

Because of the extremely tight distributions and the insensitivity on input data distribution (described in Sect. 6), the mean value will be taken as representative of the tunnel metric for a given throughput, direction, detection method, and tunnel application. This choice of a single representative value simplifies discussion and makes presentation of the salient characteristics of the detection methods more straight forward.
Figure 9 shows several log-log plots (in order to be able to provide adequate resolution for both very small and very large throughputs), one for each detection method, that demonstrates how the mean metric generated by the methods scale as the throughput of the tunnel is increased. As is visible in Fig. 9, tunnels with lower throughput produce categorically smaller (or in the case of Born's approach, larger) metrics. This

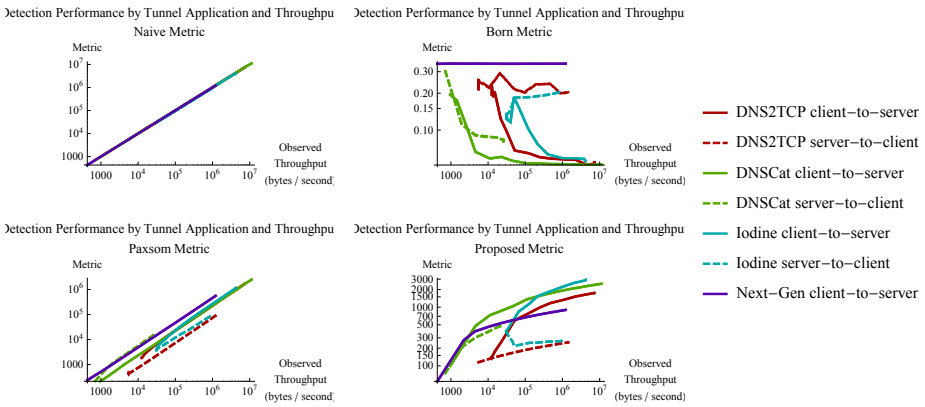


Fig. 9. These plots show how the metrics computed by the various detection methods scale per tunnelling application as a function of the tunnel throughput rates. Note that since not all tunnels are capable of a full range of throughputs (some generate a minimum amount of traffic, regardless of how low the throughput is), resulting in some of the shorter plots.

property of the lower throughput tunnels makes them necessarily harder to detect when laid over top of normal traffic. Because of this, the methods will be tested on their ability to detect the most hidden tunnel of each application, which in practice is one of the tunnels that transmits only ten bytes per second, against background normal DNS traffic. By testing the methods to ensure they detect low-rate tunnels, their ability to detect high-rate tunnels is implicitly demonstrated.

It is instructive to observe the relative standard deviations for the various tunnel applications and detection methods at the lowest throughput level, ten bytes per second. This is given in table 1.

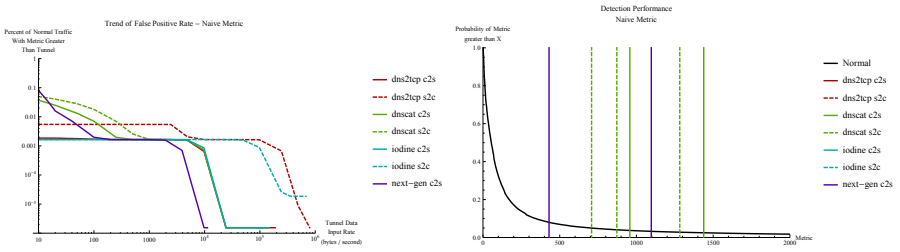
	Naive	Born	Paxson	Proposed
DNS2TCP c2s	0.0559644	0.0497584	0.0450118	0.0371504
DNS2TCP s2c	0.0477585	0.0675144	0.0495964	0.0392231
DNSCat c2s	0.0155828	0.0526164	0.018919	0.00980975
DNSCat s2c	0.0152077	0.0452746	0.0172972	0.0101241
Iodine c2s	0.00720872	0.0439898	0.0106345	0.00500239
Iodine s2c	0.00687431	0.0377363	0.0119256	0.00375623
Next-Gen	0.0309484	0.0581522	0.0399604	0.00414985

Table 1. The relative standard deviation given by $\frac{\sigma}{\mu}$ of the multiple samples for each tunnel application and detection method pairing for the lowest throughput rate.

8 Detection Performance Against Real World Data

Since tunnel metrics are represented by a single value, the mean of their samples for a given scenario, the detection methods will be ranked based on how they partition the metrics produced by real world data. For simplicity a thresholding approach will be considered as the classification mechanism, with anything below the tunnel's metric classified as legitimate, and anything above the tunnel's metric classified as a tunnel. Detection methods will be scored based on the number of false positives that could be expected to occur given the distribution obtained for normal traffic. The ordering is reversed for Born's method.

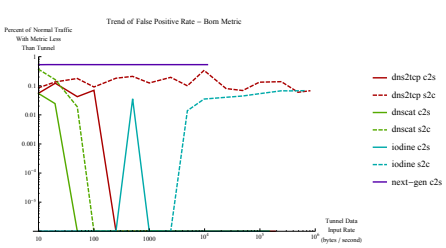
For each detection method, two plots are given that show how the tunnel applications compare to real world data. The first plot demonstrates how the estimated false-positive rate behaves as a function of throughput rate, direction, and tunnel application. The second plot shows the distribution of metrics of real world data with indicators represented by vertical coloured bars placed to mark the mean metrics of various tunnel application and direction pairs as given in the corresponding legend. For each marker, the false-positive rate is the y value of the normal traffic curve at intersection of the normal curve with the vertical marker. The second plot only shows the tunnelling combinations that produce the highest false-positive rates, since those are the scenarios of greatest interest.



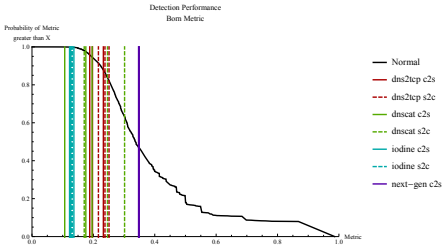
(a) Trend of False Positive Rate for Tunnels by Throughput - Naive Metric

(b) Tunnel Detection Performance - Naive Metric

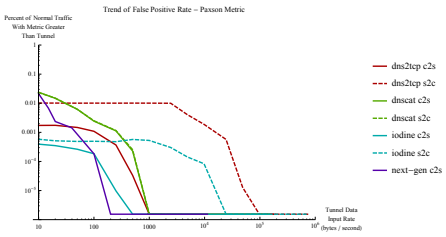
The naive method, due to the nature of the the capture involving (relatively) very few duplicate queries, performs quite well overall even on the next-gen tunnel traffic. Figure 10(a) shows the trends of the false positive rate for the tunnelling applications as a function of the data throughput. The lack of duplication in the DNS queries and its impacts were discussed in section 5.1 in greater detail.



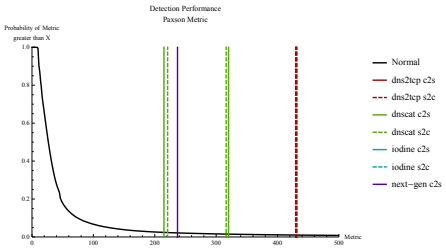
(c) Trend of False Positive Rate for Tunnels by Throughput - Born's Metric



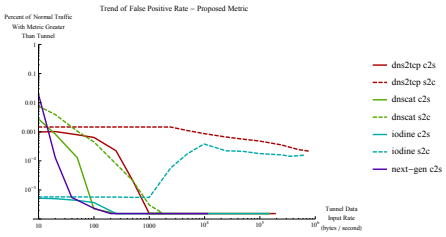
(d) Tunnel Detection Performance - Born's Metric



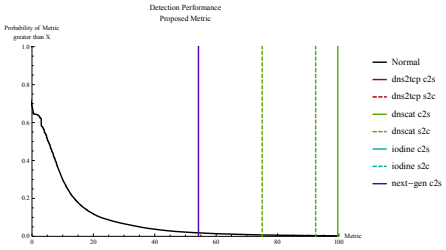
(e) Trend of False Positive Rate for Tunnels by Throughput - Paxson's Metric



(f) Tunnel Detection Performance - Paxson's Metric



(g) Trend of False Positive Rate for Tunnels by Throughput - Proposed Metric



(h) Tunnel Detection Performance - Proposed Metric

9 Specificity and Ambiguity of Tunnel Classification

It is possible to simplify the above plots and figures into a single chart that plots the minimum detection specificity observed for each method and each tunnelling application. The following charts only consider the certainty of detecting the tunnel in which the method is least certain. By comparing the methods in their most hostile scenarios more substantial distinctions can be observed with clearer separation between the best two methods.

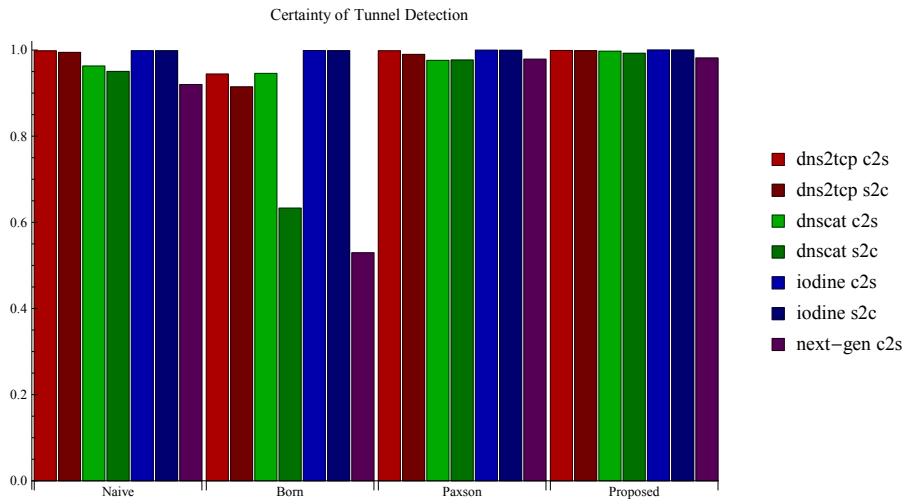
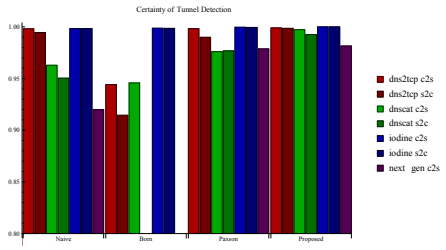


Fig. 10. Comparison of the specificity of classification of a tunnel against real-world traffic for the least certain tunnel in each detection scenario (method/application pair).

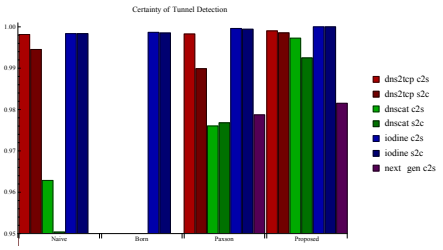
Figure 11(a) shows the same data as in Fig. 10 with a restricted range spanning the interval $[0.80, 1.00]$ as opposed to $[0, 1]$. This restricted charting range makes the the differences between the top performing methods more easily visible.
The differences between Paxson's method and the proposed method are visible, but an additional chart further accentuating them is instructive and is shown in figure 11(b). In this final chart which shows a range of certainties in the interval $[0.95, 1.00]$, the differences between the two methods are clearly visible, with the proposed approach achieving a higher certainty in every detection scenario.

10 Tunnel Detection Performance Conclusion

It is visible from the detailed plots in Sect. 8 and 9 that the proposed method is superior to its peers in its ability to detect tunnels with cer-



(a) hart of Certainty of Detection by Tunnel Application and Detection Method - 0.80 to 1.00



(b) Chart of Certainty of Detection by Tunnel Application and Detection Method - 0.95 to 1.00

tainty in excess of ninety eight percent. This extremely high detection rate is achieved at a very short time scale and with very low tunnel throughput.

11 Conclusion

In this work, a new method of detecting DNS tunnels was proposed, described, and evaluated.

The existing landscape of detection methods was summarized (section 3) and a gap identified indicating a need for a new method (section 4.2) that has both high processing performance on commodity hardware, and robust tunnel detection. A prototypical next-gen tunnel application was postulated and simulated in order to present a more difficult detection task during evaluation alongside existing tunnel applications (section 6.1). Several detection methods from the literature as well as the proposed method were selected (section 6) and implemented on a common framework. The implemented methods were tested for processing performance (section 6.3) and tunnel detection performance on a large sample of real world DNS traffic as well as existing and next-gen tunnelling application data (section 7).

As is shown in the relevant sections, the proposed method outperforms its peers in both processing performance and tunnel detection in almost every situation by a measurable and often considerable margin.

When compared to its peers from literature, the performance improvement over Paxson's approach on real-world data is approximately 25% while a speedup of almost 100% is observed when processing tunnel data under the Cython interpreter. The margins are similar under PyPy, with the primary difference being that the proposed method is nearly double the performance of Paxson's approach on both real world and tunnel application data. Born's approach is able to narrow the gap on tunnel data to only approximately a 50% improvement, while the gap on real-world data increases to about 30%. Under PyPy however, the performance of the proposed method and Born's method become very similar with gaps less than 10%.

Examining tunnel detection performance shows additional benefits to the proposed method. Born's method shows very poor performance, as expected, when faced with the next-gen tunnel, with a less drastic performance hit shown on other tunnelling software. In all cases the proposed, as well as Paxson's, approach out-perform Born's method by a considerable margin in false-positive rates. The proposed approach reduces the false positive rate by almost 98% when compared to Born's approach. When comparing Paxson's approach with the proposed approach, the proposed approach reduces the number of false positives by up to nearly 90% in the best case, with an average reduction of 70%.
The exception is the naive method which outperforms the proposed method in processing performance by approximately fifty percent at the expense of detection performance, with the proposed approach reducing the false positive rate by 82%. As was mentioned in Sect. 5.1 however, the naive method's detection performance is only as good as it is in this case due to the lack of duplication of DNS queries.
Recall that the improvements demonstrated by the proposed approach are on measured in a highly pessimistic detection scenario, with false-positive rates rapidly dropping below 10^{-6} as throughput increases.
The end product of these results is a contribution to the field comprising a new detection method with superior processing and detection performance. The new detection method falls short of matching the naive method's processing performance in many situations with the trade off of far superior detection performance in the general case.

12 Future Work

The proposed method was shown to be highly effective in detecting the targeted network traffic with superior performance, outperforming existing methods in almost every scenario. In order for this to be of value, however, an adoption of this method into an existing commercial products or an implementation as a plugin for an existing security framework would be necessary.
Future work could include implementing the proposed method for Bro, Snort, Suricata or other existing intrusion detection systems in order to improve the ability of organizations to observe DNS tunnels in their network. Partnership with, and adoption by, an existing industry partner would aid in the spread and deployment of this technique in enterprise and corporate environments. Due to the nature of computer security research, regardless of how valuable a contribution may be, unless there is deployment of the approach into real world environments it is unable to aid in securing the Internet.

References

1. M. Bauer. New covert channels in HTTP: adding unwitting Web browsers to anonymity sets. In S. Jajodia, P. Samarati, and P. F. Syverson, editors, *WPES*, pages 72–78. ACM, 2003.

2. K. Borders and A. Prakash. Web tap: detecting covert web traffic. In V. Atluri, B. Pfitzmann, and P. D. McDaniel, editors, *ACM Conference on Computer and Communications Security*, pages 110–120. ACM, 2004.
3. K. Born. Browser-Based Covert Data Exfiltration. *CoRR*, abs/1004.4357, 2010.
4. K. Born. PSUDP — Kenton Born. http://www.kentonborn.com/psudp, jul 2010. PSUDP source code and implementation.
5. K. Born. PSUDP: A Passive Approach to Network-Wide Covert Communication. 2010.
6. K. Born and D. Gustafson. Detecting DNS Tunnels Using Character Frequency Analysis. *CoRR*, abs/1004.4358, 2010.
7. K. Born and D. Gustafson. NgViz: Detecting DNS Tunnels through N-Gram Visualization and Quantitative Analysis. *CoRR*, abs/1004.4359, 2010.
8. R. Browne. An Entropy Conservation Law for Testing the Completeness of Convert Channel Analysis. In D. E. Denning, R. Pyle, R. Ganesan, and R. S. Sandhu, editors, *ACM Conference on Computer and Communications Security*, pages 270–281. ACM, 1994.
9. P. Butler, K. Xu, and D. D. Yao. Quantitatively Analyzing Stealthy Communication Channels. In J. Lopez and G. Tsudik, editors, *ACNS*, volume 6715 of *Lecture Notes in Computer Science*, pages 238–254, 2011.
10. S. Cabuk, C. E. Brodley, and C. Shields. IP covert timing channels: design and detection. In V. Atluri, B. Pfitzmann, and P. D. McDaniel, editors, *ACM Conference on Computer and Communications Security*, pages 178–187. ACM, 2004.
11. S. Cabuk, C. E. Brodley, and C. Shields. IP Covert Channel Detection. *ACM Trans. Inf. Syst. Secur.*, 12(4), 2009.
12. M. Chamberland. Snort rules for Iodine Covert DNS Tunnel Detection. http://www.securitywire.com/2009/07/snort-rules-for-iodine-covert-dns-tunnel-detection, jul 2009.
13. H. S. Consultants. HSC - Tools - Dns2tcp. http://hsc.fr/ressources/outils/dns2tcp/index.html.en, may 2012.
14. Dennis Arturo Ludeña Romaña and Yasuo Musashi. Entropy Based Analysis of DNS Query Traffic in the Campus Network. 2007.
15. L. P. Deutsch. kryo.se: iodine (IP-over-DNS, IPv4 over DNS tunnel). http://code.kryo.se/iodine, feb 2010.
16. C. J. Dietrich, C. Rossow, F. C. Freiling, H. Bos, M. v. Steen, and N. Pohlmann. On Botnets That Use DNS for Command and Control. In *Proceedings of the 2011 Seventh European Conference on Computer Network Defense*, EC2ND '11, pages 9–16, Washington, DC, USA, 2011. IEEE Computer Society.
17. M. Dornseif. mdornseif/DeNiSe GitHub. https://github.com/mdornseif/DeNiSe, jan 2006.
18. M. Dusi, M. Crotti, F. Gringoli, and L. Salgarelli. Tunnel Hunter: Detecting application-layer tunnels with statistical fingerprinting. *Computer Networks*, 53(1):81–97, 2009.
19. G. Farnham. Detecting DNS Tunneling. *InfoSec Reading Room*, February 2013.

20. S. Gianvecchio and H. Wang. Detecting covert timing channels: an entropy-based approach. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 307–316. ACM, 2007.
21. M. V. Horenbeeck. Deception on the network: thinking differently about covert channels, 2006.
22. Internet Storm Centre - SANS Institute. Hash Database — SANS Internet Storm Center; Cooperative Network Security Community - Internet Security. https://isc.sans.edu/tools/hashsearch.html#dns_interface, oct 2013.
23. jhind. Catching DNS tunnels with A.I. http://www.meanypants.com/meanypants, jul 2009.
24. A. Karasaridis, K. S. Meier-Hellstern, and D. A. Hoeflin. Detection of DNS Anomalies using Flow Data Analysis. In *GLOBECOM*. IEEE, 2006.
25. I. S. Moskowitz, R. E. Newman, D. P. Crepeau, and A. R. Miller. Covert channels and anonymizing networks. In S. Jajodia, P. Samarati, and P. F. Syverson, editors, *WPES*, pages 79–88. ACM, 2003.
26. R. C. Newman. Covert computer and network communications. In *Proceedings of the 4th annual conference on Information security curriculum development*, InfoSecCD '07, pages 12:1–12:8, New York, NY, USA, 2007. ACM.
27. K. Okamura and Y. Oyama. Load-based covert channels between Xen virtual machines. In S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, editors, *SAC*, pages 173–180. ACM, 2010.
28. V. Paxson. Behavioral Detection of Stealthy Intruders. https://seclab.cs.ucsb.edu/academic/projects/projects/cybaware/2011, sep 2011.
29. T. Pietraszek. DNScat. http://tadek.pietraszek.org/projects/DNScat, sep 2005.
30. J. Plenz. DNStunnel.de - free DNS tunneling service. http://dnstunnel.de, jun 2011.
31. Proventia. Proventia Server IPS - DNS tunnel traffic detected. http://www.iss.net/security_center/reference/vuln/DNS_Tunnel_Detected.htm, jan 2013.
32. B. Ray and S. Mishra. A Protocol for Building Secure and Reliable Covert Channel. In *Proceedings of the 2008 Sixth Annual Conference on Privacy, Security and Trust*, PST '08, pages 246–253, Washington, DC, USA, 2008. IEEE Computer Society.
33. S. Roolvink. Detecting attacks involving DNS servers. December 2008.
34. S. H. Sellke, C.-C. Wang, S. Bagchi, and N. Shroff. Tcp/ip timing channels: Theory to implementation. In *INFOCOM 2009, IEEE*, pages 2204–2212. IEEE, 2009.
35. A. B. Shaffer, M. Auguston, C. E. Irvine, and T. E. Levin. A security domain model to assess software for exploitable covert channels. In Ã. Erlingsson and M. Pistoia, editors, *PLAS*, pages 45–56. ACM, 2008.

36. P. Team. PyPy Status Blog: PyPy is faster than C, again: string formatting. http://morepypy.blogspot.com/2011/08/pypy-is-faster-than-c-again-string.html, oct 2013.
37. B. Thomas, B. E. Mullins, G. L. Peterson, and R. F. Mills. An FPGA System for Detecting Malicious DNS Network Traffic. In G. L. Peterson and S. Shenoi, editors, *IFIP Int. Conf. Digital Forensics*, volume 361 of *IFIP Advances in Information and Communication Technology*, pages 195–207. Springer, 2011.