Evading Stepping Stone Detection Under the Cloak of Streaming Media with SNEAK

Jaideep D. Padhye\textsuperscript{a,}\textsuperscript{*}, Kush Kothari\textsuperscript{b}, Madhu Venkateshaiah\textsuperscript{c}, Matthew Wright\textsuperscript{b,*}

\textsuperscript{a} Cisco, Inc., Mail Stop SJCF/1/4, 210 West Tasman Drive, San Jose, CA 95134, USA
\textsuperscript{b} Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019, USA
\textsuperscript{c} clearAvenue, LLC, 939 Elkridge Landing Road #195, Linthicum, MD 21090, USA

Abstract

Network-based intrusions have become a serious threat to the users of the Internet. To help cover their tracks, attackers launch attacks from a series of previously compromised systems called stepping stones. Timing correlations on incoming and outgoing packets can lead to detection of the stepping stone and can be used to trace the attacker through each link. Prior work has sought to counter the possibility of the attacker employing chaff packets and randomized delays. To date, however, researchers have not accounted for the full range of techniques that a sophisticated attacker could apply. In this work, we show that such an attacker could avoid detection by the best known stepping stone detection methods. We propose a simple buffering technique that could be used by an attacker on a stepping stone to evade detection. This technique makes the timing of packets in the output flow of the stepping stone entirely independent of the timing of packets from the input flow, thereby eliminating the timing link that makes existing stepping stone detection methods possible. To accomplish this, we only require buffering at the stepping stone and enough chaff packets to generate a constant rate flow. This traffic has the characteristics of a multimedia stream, such as Voice over IP (VoIP), which is quite common on the Internet to-
day. To test the effectiveness of our technique, we implemented it in a prototype stepping stone application and tested its performance on the DETER testbed and the PlanetLab testbed. Our prototype successfully evades watermark-based detection and provides reasonable performance for shell commands over at least three stepping stones.

Key words: stepping stone detection, traffic analysis, time-based watermarking

1. Introduction

Hackers who seek to break into systems and steal information or create disruptions of service also hope to cover their tracks. To increase the difficulty of detection and forensics, hackers typically launch attacks indirectly by relaying their attack through a chain of intermediate (previously compromised) systems called stepping stones (also known as hop points). The attacker does this by constructing a chain of interactive connections using protocols like Telnet or SSH. The commands that the attacker types on his local terminal are relayed through the stepping stones until they reach the victim. This gives the attackers a certain level of anonymity and makes it hard for an investigator to trace the attack back to its origin.

Since the attacker is sending attack traffic to a stepping stone and the stepping stone is just forwarding it to an external host, the stepping-stone detection problem comes down to finding an outgoing connection with the same characteristics as a given incoming connection. An intuitive approach to solve this problem would be to compare the contents of the incoming and outgoing packets in a network to find packets with the same content. However, the use of encrypted communication protocols like SSH have made this approach ineffective. We thus need to use other aspects of the traffic like timing characteristics to detect stepping stones. One of the more promising approaches to solve the stepping-stone detection problem is to actively perturb the timings of packets in incoming streams, making it easy to find outgoing streams with the same tim-
ing patterns. This approach can be made robust to random changes in packet timing introduced by the attacker [1].

Stepping-stone detection is closely related to the problem of timing analysis to de-anonymize connections in systems like Tor [2]. A variety of sophisticated techniques have been proposed for timing analysis, including passive attacks [3, 4, 5] and active attacks [6, 7, 8]. Active attacks are particularly powerful, as they can deanonymize streams with a constant rate of packets (such as multimedia traffic [6]), and a wide variety of flow transformations like dropping packets, adding chaff or dummy packets, splitting the stream, and merging different streams [7]. These attacks could be applied to the stepping-stone detection problem with great effectiveness.

Contributions. In this paper, we explore a simple approach to defeat all known approaches to timing analysis and thereby evade stepping-stone detection techniques. This paper is an extension of our previous work [9, 10, 11]. Our system, SNEAK, employs link padding between stepping stones to stop timing analysis. Simply put, SNEAK forwards packets in such a way that the outgoing traffic is independent from incoming traffic. While link padding [12] and other ways to make outgoing traffic independent from incoming traffic [13, 14] have previously been proposed and studied, we are the first to show that link padding is practical and effective for hackers seeking to evade stepping-stone detection.

We believe that this represents an important result for three reasons. First, prior work in stepping-stone detection showed that the hacker could be discovered if the amount of chaff is bounded [14]. SNEAK employs a practical mechanism that the attacker can use to exceed those bounds without creating suspicion. Second, SNEAK evades advanced watermarking-based active timing analysis without attempting to detect the watermark. This means that it is not subject to further advances in the invisibility of the watermark. This is in contrast with the recent work of Kiyavash et al. [15], in which they attempt to detect the presence of a watermark, recover watermarking parameters, and remove the watermark from the flow. If parameters such as interval length are modified in this scheme from the original values, watermarking will be invisible.
to attacker. Instead, SNEAK removes all watermarks and other timing information so advancements in watermarking or modifications in the parameters will not be effective. Third, by defeating existing methods of timing analysis, SNEAK demonstrates the need for complementary methods of detecting stepping stones.

Our primary assumption is that constant-rate traffic will not be flagged as unusual. Since multimedia streams are becoming more and more common on the Internet, particularly with the growing use of Internet telephony (VoIP) and video content, we feel that this is a reasonable assumption.\footnote{For multimedia streams that are not constant rate but send packets at a high rate without very long gaps, our technique can be adapted with some cost to the system’s response time.} According to Ellacoya Networks, Inc., the usage data of one million broadband subscribers shows that streaming video and audio combine to make up about 22% of all traffic, with another 1% for VoIP, even though it is not a high bandwidth media [16]. The popular Skype P2P VoIP service, which automatically encrypts calls, had over 8 billion minutes of computer-to-telephone calls in 2008 [17]. These trends may not hold in certain corporate and high-security environments, which are likely to be desirable targets for hackers. However, stepping stones are intermediate hosts and can be located in more open environments like broadband-connected home PCs and college campuses, even while the hacker aims for more secure targets. Additionally, many of the currently used VoIP codecs have a fixed rate, meaning constant packet flows. For example, the website http://www.voip-info.org/wiki-Codes includes a list of codecs used in VoIP. Most of the codecs listed are fixed rate. In our lab, we have tested the popular VoIP applications Skype and Freecall using Netmeter. Though Skype uses variable rate connections for computer-to-computer calls, Skype and Freecall use 3.8 Kbps and 10.4 Kbps constant rate traffic, respectively, for computer-to-telephone calls.

Given the assumption of constant rate traffic, our proposed technique is fairly simple. The attacker, using a path of stepping stones, disguises his traffic
as encrypted VoIP or some other multimedia stream. At each stepping stone between the attacker and the target, a receiving process takes the packets of this stream are re-encrypted and placed in a small buffer. A different process running on the stepping stone, the sending process, takes these payloads from the buffer and sends them at a constant rate to the next node, also disguised as an encrypted VoIP stream. Whenever the buffer is empty, a chaff packet is sent instead. The key principle is that the sending process is independent of the receiving process. The timing characteristics of the packets generated by the sending process have no dependence on the timing of packets from the incoming stream, so all timing information is removed.

This technique should stop any stepping stone detection technique based on the timings of packets. In this paper, we explore the effectiveness and practicality of this idea in more detail. In particular, we use experiments with a simple prototype to show that the costs of using this technique, in terms of buffering delay and dropped packets, are reasonable for interactive stepping stones. When sending shell commands over three stepping stones, we experience acceptable delays for interactive sessions. We also discuss the possibilities for countering this technique and the implications for current and ongoing research in this area.

We now describe the organization of the rest of the paper. In Section 2, we discuss the background concepts and related work to justify our design decisions. In Section 3, we describe our model and present our proposed buffering technique in detail. Section 4 describes the design of the prototype we built, while Section 5 describes our experiences using it on the DETER and PlanetLab testbeds. We discuss practical considerations in Section 6, and Section 7 concludes.

2. Background

We now describe some of the techniques that have been used to detect stepping stones and related work in timing analysis.

Let us first define some terminology. When a person logs into one computer and from there logs into another computer and so on, we refer to the sequence of logins as a connection chain [18]. Any intermediate host on a connection
chain is called a *stepping stone*. Typically, connection chains are formed by using terminal emulation programs like Telnet and SSH. The stepping stones thus formed are called *interactive stepping stones*, since the attacker types in a command and waits for a response. Though it is possible for a computer program to create a connection chain, e.g. for file transfer, we limit the scope of our research to interactive stepping stones.

Stepping stone detection is a process of observing all incoming and outgoing connections in a network and determining which ones are parts of a connection chain. This problem is closely related to the problem of *traceback*, tracing intruders through the Internet by following the connection chain. In both of the above applications, the fundamental underlying problem is to compare and analyze two connections and determine if there is any correlation between them. Researchers have proposed two main approaches: *passive monitoring* and *active perturbation*.

2.1. Passive Monitoring

Passive monitoring is an approach in which traffic flows are observed and analyzed to find correlations. The interactive stepping stone problem was first formulated and studied by Staniford and Heberlein [18]. They proposed a content-based algorithm that creates thumbprints of streams and compares them, looking for good matches. The problem with this approach is that it requires that the traffic be in unencrypted plain text. Zhang and Paxson [19] were the first to propose a scheme to correlate traffic across stepping stones even if the traffic is encrypted by the stepping stone. The method is based on classifying traffic into on and off periods, e.g. active typing versus thinking periods in interactive SSH sessions, and correlating the timings of these periods.

Yoda and Etoh [20] define the minimum average delay gap between the packet streams of two TCP connections as the *deviation*. They then propose to correlate streams using deviations, which does not require clock synchronization and is able to correlate connections observed at different points of network. Wang, Reeves, and Wu [21] address the problem of correlation by a scheme
based on inter-packet timing characteristics. While timing-based correlation approaches have the advantage that they are simple and do not disturb normal traffic, they are vulnerable to countermeasures by the attacker. The attacker can perturb the timing characteristics of connections by selectively or randomly delaying packets at the stepping stone [13]. This kind of perturbation adversely affects timing-based correlation.

Blum et al. describe a passive timing analysis technique that is robust to moderate timing perturbations and chaff [14]. They model the timing analysis problem using poisson-distributed flows and provide bounds on the number of packets required for detection and provable guarantees on the false positive and false negative rates. Their assumptions for proving these bounds are that the amount of delay perturbation and the amount of chaff are both limited. In our work, we show a practical technique for the attacker to use that greatly increases the amount of chaff beyond these limits by hiding in apparent multimedia streams.

Donoho et al. first described how, with output timings independent of the input timings, the stepping stone could evade detection completely [13]. In our work, we show how this could be made practical for interactive stepping stones and demonstrate this in experiments. Blum et al. extend this notion with theoretical analysis of the amount of chaff required for evasion given poisson-distributed flows [14]. They show that it is possible to use lower amounts of chaff than we propose using and still defeat passive timing analysis. However, encapsulating the attack traffic in multimedia protocols that have high traffic rates allows us to use simpler techniques that are also effective against active perturbation.

2.2. Active perturbation

A promising defense against the problem of random timing perturbation by an attacker is active perturbation by the observer. In this approach, an incoming connection is perturbed by inducing a packet loss or delay and the outgoing connections are checked to see if the perturbation is echoed in them. Since the
attacker does not know what the perturbation is, he will not be able to use random perturbation to precisely affect the results. Wang and Reeves [1] proposed the first active correlation method designed to be robust against random timing perturbation. In this scheme, a unique delay-based watermark is embedded into a traffic flow by slightly adjusting the timing of selected packets in the flow. The watermarked flow can be uniquely identified and thus correlated with other flows in the connection chain.

Watermarking consists of two complementary processes: embedding the watermark and decoding the watermark. A watermark is simply a unique binary string. The process of embedding one bit of this string consists of changing some property of a traffic flow such that the change represents a bit. Decoding the watermark involves capturing candidate flows that might match the watermarked flow and looking for the bits in the flow characteristics. The bits of the watermark should have enough redundancy to ensure that they are decoded correctly with high probability.

In the technique proposed by Wang and Reeves [1], packets are randomly chosen and paired to obtain inter-packet delays (IPDs). A watermark is embedded by manipulation of these IPDs. To make the scheme robust against random perturbations, multiple IPDs are manipulated to embed one bit. In their paper the authors make the following assumptions:

1. While the attacker can add extra delay to any or all packets of an outgoing flow of the stepping stone, the maximum delay that he can introduce is bounded
2. The attacker does not know which packets are being watermarked
3. The component that decodes watermarks in traffic flows knows which packets have been watermarked

The first assumption is reasonable for an interactive session, as very high delays will disrupt the attacker’s ability to use the connection. The second assumption is reasonable, due to random packet selection, although detection and removal may be possible [22, 15]. The third assumption, however, can be broken with
the addition of just a few chaff packets. While this is a problem for earlier works that assume perfect packet matching [1, 6], recent work by Pyun, et al. handles this problem [23]. Our attacker approach defeats all of these methods.

Also relevant to the present work is the use of timing-based watermarks to track anonymized Voice over IP (VoIP). Anonymous VoIP would have different traffic patterns from typical Web browsing, similar to a constant rate flow. As an attack on this anonymized service, Wang, Chen, and Jajodia propose a watermark-based technique to track anonymized VoIP calls [6]. This technique adapts the technique of [1], which cannot be used directly due to the more stringent real-time constraints for VoIP streams as compared with SSH. In the VoIP-tracking technique, the packets that are delayed for embedding the watermark are chosen randomly and delayed by a fixed amount. This delay is called the \textit{watermarking delay}. Since the anonymity system has no way of knowing the watermarking delay and which packets are delayed, random perturbations are not an effective defense.

More recently, Wang et al. extended their work by introducing a \textit{centroid-based} watermarking scheme that provides robustness against chaff and the mixing and merging of different flows [7]. The time duration for embedding the watermark is divided into equal intervals and, using the timings at which the packets appear in the interval, the centroid of the interval is calculated. The difference between centroid values is used to encode and decode bits. Although their work considers many types of \textit{inter-flow transformations}, such as adding chaff and merging flows, it does not investigate our more robust technique. In particular, they say that they exploit a “fundamental limitation of low-latency anonymizing systems” in that they do not eliminate the packet timing correlation between the flows. This is precisely what our technique is designed to do.

Recent work has also studied ways to hide watermarks effectively from the attacker. A technique based on DSSS encoding was proposed to help in hiding the watermarks [8]. More recently, Houmansadr et al. proposed a \textit{non-blind} watermarking scheme called RAINBOW [24]. RAINBOW is non-blind in that
it stores the IPD values of incoming traffic and these are used by the decoder to analyze the flow; prior schemes ignore this information. This allows the watermarking delays to be very small, helping to make it invisible to the attacker and prevent him from modifying his flow to remove watermarks. In SNEAK, however, we do not look for the presence of watermarking. We instead eliminate all the timing characteristics in the flow, both the IPD variation and watermarks, thus evading detection regardless of watermark invisibility.

In our work, we specifically demonstrate that SNEAK defeats the interval centroid-based watermarking scheme of Wang et al. [7]. We believe that, as an active attack against arbitrary traffic types with substantial flow modifications, this scheme is the most powerful detection method in the literature. In particular, it should be more powerful than passive techniques, since it takes advantage of modifications to the timing of packets in the stream. Also, it is more suitable for attempting to detect our approach than RAINBOW [24], as it uses larger watermarks to obtain greater robustness with less emphasis on invisibility; SNEAK modifies timings without attempting to detect the watermark.

2.3. Anonymity

Another related area is timing analysis attacks against systems for low-latency anonymous communications. Active perturbation attacks against anonymity systems are discussed above; we now describe passive attacks in anonymity systems. Danezis [4] presents an attack based on traffic analysis on connection-based mixed networks functioning in continuous mode. It uses signal detection techniques to compare a traffic pattern extracted from the stream that is being tracked with all the links in the network. Levine et al. [3] show that simple statistical correlation is an effective timing analysis technique, even if all users have the same constant rate of traffic. They propose a defense called defensive dropping, in which intermediate nodes on the path drop selected chaff packets, and show that it is effective in reducing attacker effectiveness. Shmatikov and Wang propose adaptive padding, in which packets are added at intermediate nodes on the path in response to gaps between packets [25]. Both defensive
dropping and adaptive padding are inappropriate for evading stepping stone detection, as the resulting traffic flows are evidently not typical and could be flagged as abnormal.

More directly relevant to the present work is the study of Fu et al. on the effectiveness of link padding [12]. Link padding is the use of cover traffic between nodes on an anonymity path. This idea was proposed for use in the Tarzan P2P anonymity system by Freedman and Morris [26] and has its origins in Chaum’s seminal work on mixes [27]. A key to making link padding successful is to make the timing of the output stream independent from the input stream, theoretically removing the chance of attacks based on timing correlation. Fu et al. describe two types of link padding, Constant Interarrival Time (CIT), in which packets are sent at a fixed rate and Variable Interarrival Time (VIT), in which packets are sent at random times within a specified range. They also present the results of experiments conducted on a small network testbed and found that VIT provides better protection against their proposed statistical pattern recognition technique.

Any success against CIT may seem surprising, as the output stream should be independent from any inputs. Fu et al., however, point out that the system itself can have delays due to the workload of receiving packets [12]. The experiments conducted to demonstrate this consider two streams passing through one node. The two streams both randomly alternate between two states: high, with an average of 40 packets per second, and low, with an average of 10 packets per second. With such dramatic changes in the state of each stream, we believe that it is possible for two such streams to be distinguished.

However, the observer in the stepping stone model faces a different set of assumptions. In stepping-stone detection, the detector can already see that there are streams entering and exiting a single node. The challenge is to link these as being the same, rather than comparing two pairs of streams. Since fluctuations in the incoming stream could affect system load and impact timings in unrelated outgoing streams, it is much harder to determine whether the outgoing stream is related to the incoming stream. Additionally, we employ
constant rate streams as input. This means that the fluctuations at the stepping stone will be much less those due to alternating between high and low states as in Fu et al.’s experiments. Finally, we note that active perturbation should be more powerful than passive observation, since it creates more information to observe. By demonstrating effectiveness against active perturbation we show that link padding, when used carefully (e.g. with constant rate streams), is relevant as a means of hiding stepping stones.

3. System Model

In this section, we describe a model for the use and detection of stepping stones. We also propose a technique by which an attacker can evade sophisticated stepping stone detection methods, including all methods that have been proposed to date.

Consider an attacker, using node A as a client, who has compromised system S in a network N_1 and uses it as a stepping stone. He uses node S to launch an attack on the target system T in another network N_2. For simplicity, let us assume that there is only one stepping stone, i.e. the attacker is relaying the attack through only one compromised system. Let us call the person who is trying to detect the stepping stone the observer. The observer’s objective is to detect that an incoming connection is being relayed through node S to a host outside the network. The observer does this by embedding a timing-based watermark into all the connections coming into N_1 and tries to detect if any of the outgoing connections contain the watermark that he embedded. The attacker’s objective is to evade detection by the observer by distorting the
watermark to an extent that it is hard to detect.

We propose that the attacker do this by splitting the stepping stone into two independent processes: a receiver and a sender. The receiver accepts packets as they arrive and adds them to a FIFO queue. The sender sends packets at a constant rate to mimic a multimedia stream. If there is a packet in the queue when it is time to send, it removes it from the queue and sends it. If not, it sends a chaff packet. Since these two processes are independent, the timing information is lost and the observer will not be able to detect the watermark.

We note that any outgoing traffic pattern that is independent of the input traffic pattern would be effective for this purpose. We use constant rate traffic because it is used in multimedia streams and for its consistent flow, which provides an upper bound on the buffering delay. Non-constant rate flows, e.g. generated using a Poisson distribution, might be considered suspicious, as they would likely not match the traffic patterns of other flows in the network. Finally, we note that some multimedia streams do not use a constant flow [28]; such flows can be emulated based on their specific characteristics. Essentially any flow can be used as long as the packet rate is high enough to limit buffering delays in most cases. Exploring this extension is beyond the scope of this paper.

For the purpose of our study, we make the following assumptions:

- The attacker has complete control of $A$ and $S$. Thus he can modify the system or use rogue applications on any of these systems.

- The attacker does not know which packets are watermarked

- The attacker does not know the parameters of the watermark algorithm, including the watermark delay and the distance between packets.

3.1. Evading detection

We now describe in detail a technique that the attacker can use to evade detection by the observer. The attacker first optionally profiles the connection from $A$ to $S$, as described below in Section 3.2. He then establishes the connection chain through the stepping stone $S$ to the target host $T$ and starts the
attack. When the attack packets arrive at the gateway to \( N_1 \), the observer embeds a watermark.

As packets arrive at the stepping stone, the receiver accepts the packets, decrypts them, and puts them in a FIFO queue. The sender encrypts and sends packets out at a constant rate from \( S \) to \( T \). It takes packets from the queue if they are available, otherwise generating a random chaff packet to send. To generate a constant-rate traffic stream, the attacker needs to buffer packets that arrive early and drop packets when they arrive late. To achieve this, the attacker divides the time into fixed-length slots, beginning with the arrival of the first packet at \( S \). The length of each slot is \( \text{ipd} = 1/\tau \) which is the mean inter-packet delay (IPD) at the source of the traffic. Each packet is expected to arrive in its respective slot, but packets may arrive earlier or later than expected due to variability of network delays and watermarking delays encountered by packets. This is illustrated in Figure 2.

If a packet arrives early, the packet is delayed until the end of the slot. If a packet arrives late, i.e. after the end of its slot, a chaff packet with a random payload is sent in place of the actual packet. When the delayed packet finally arrives, it is buffered until the end of the next available slot and then sent. Some packets may arrive more than one slot late, followed by packets that arrive on time. This could lead to multiple packets being buffered in the queue and have a cascading effect on the buffering delays. As increasingly longer delays would affect the quality of the connection, the stepping stone should drop packets to
We have designed two approaches for dropping packets. In the first approach (Algorithm 1), the sender drops packets, meaning that it decides reactively whether to drop a packet after having buffered it. If the sender finds more than two packets in the queue, it drops the first one (which is older) and sends the next one. In the second approach (Algorithm 2), the receiver drops packets, meaning that the system drops packets proactively to limit the buffer. Algorithm 2 uses connection profiling, described below, to choose a time limit $\tau$; packets that arrive more than $\tau$ past their expected arrival time are dropped. The key difference in the effect of these approaches is that Algorithm 1 has fewer dropped packets, while Algorithm 2 has lower buffering delay, as we show in Section 5.

The entire process can be summarized by the following algorithm:

1. From the time the first packet was sent, trigger a Send event at intervals of duration $ipd$.
2. Buffer incoming packets.
3. Drop excessively late packets (Algorithm 2).
4. When a Send event is triggered:
   a. If there is no packet in the buffer, send a chaff packet and continue.
   b. Else If there are more than two packets in the buffer, drop the first packet (Algorithm 1).
   c. Send one packet.

More detailed descriptions of Algorithm 1 and Algorithm 2 are given in Appendix A.

3.2. Connection profiling

To maximize the system’s performance for Algorithm 2, it is useful for the attacker to learn the delay characteristics of the network. To this end, the attacker can optionally profile the network connection between his host and the stepping stone. Before establishing the connection chain to launch his attack, the attacker sends a stream of packets from his system at a specific rate to the...
stepping stone. On the stepping stone, the attacker records the arrival time of these packets and calculates the IPDs and standard deviation $\sigma$ of the IPDs. This allows him to profile the connection’s delay characteristics. The attacker combines this information with his knowledge of the traffic rate to derive the expected arrival time of packets.

Given a packet stream $P_1, ..., P_n$ being sent at rate $r$ and received on the stepping stone with time stamps $t_1, ..., t_n$ respectively ($t_i < t_j$ for $1 \leq i < j \leq n$), we define the IPD between $P_{k+1}$ and $P_k$ as:

$$ipd_k = t_{k+1} - t_k, \ (k = 1, ..., n - 1)$$

Since the attacker knows the traffic rate $r$ at the source, he knows the mean inter-packet delay $\overline{ipd} = 1/r$. He calculates the IPD standard deviation as:

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n-1} (ipd_i - \overline{ipd})^2}$$

Since the attacker sends packets at a constant rate, he can expect that the IPDs are normally distributed around the mean when the packets arrive at the stepping stone [1]. According to the empirical rule for a normal distribution, the attacker can deduce that:

- 68% of the packets will arrive within 1 standard deviation (of the mean)
- 95% of the packets will arrive within 2 standard deviations
- 99.8% of the packets will arrive within 3 standard deviations

Other research has suggested that jitter should be exponentially distributed [29] or gamma distributed [30]. The use of a normal distribution helps us to design the system for good performance, but our approach can be modified for various network conditions. In Section 5, we show that our approach works well in experiments over the Internet.

Algorithm 2, described above in Section 3.1, relies on this profiling to estimate the standard deviation of the IPD to calculate values for deciding whether
to drop packets that arrive late. Specifically, any packet that arrives more than $3\sigma$ past its expected arrival time is dropped. This ensures that few packets are dropped, thereby minimizing retransmissions. At the same time, this limits the number of buffered packets, as long as the network jitter is not substantially larger than the mean IPD. In high-jitter networks, our approach could be adapted by reducing the allowed delay at the cost of more retransmissions.

### 3.3. Observer countermeasures

Since the sending process generates a constant rate traffic stream that is independent of the receiving stream, all timing information of the traffic flow is lost. In effect, this totally removes the watermark that was previously embedded and any other timing characteristics. To make it harder for the attacker to evade detection, the observer can increase the watermarking delays. Increasing the watermarking delay makes the detection scheme more robust. This would also increase the jitter in the stepping stone connection. To compensate for this, the attacker has to buffer packets for a longer duration and would have to drop more packets, thus degrading the quality of his own connection. The observer could attempt to delay relatively few packets with much higher delays, e.g. 100 ms. In this case, the connection profiler would create a traffic profile similar to the un-watermarked case. The delayed packets would be treated as drops at $S$, somewhat degrading the attacker’s usability.

Note that increasing the watermarking delay affects connection quality. Since the observer needs to watermark all incoming connections, an increase in watermarking delay would not only affect the connection being used by the attacker, but all other incoming connections in the network. This substantially limits the options for active perturbation. In Section 6, we discuss some other countermeasures that the observer could use.

### 4. A Stepping-Stone Prototype

In this section, we describe the design of the SNEAK experimental prototype application that employs our algorithms to evade detection.
4.1. Prototype Design

Our goal is to create a pseudo-shell application that allows the attacker to run commands on the victim’s machine. Although this software lacks the superior shell functionality provided by Telnet and SSH, it achieves its intended goals. The software has been written in C and tested on Linux kernel 2.6.22. We send packets using the Real-time Transport Protocol (RTP) [31], thus creating a packet stream that resembles VoIP or some other multimedia. RTP provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video data. Applications typically run RTP on top of UDP to make use of its multiplexing and checksum services. Many of the popular VoIP protocols such as H.323 and SIP use RTP as their transport level protocol [32].

We also need to encrypt the payload to prevent the contents of our packet streams from being inspected by monitoring software. Hence, we use the Secure Real-time Transport Protocol (SRTP) [33], a profile of RTP that provides confidentiality and message authentication. We use libsrtp-1.4.2, a freely available implementation of SRTP. We also use a fixed packet size of 64 bytes to match existing VoIP packets (packet sizes vary depending on the codec). Our application consists of three components: the Client, the Server, and the Agent. The Client resides on the attacker’s machine, while the Server resides on the victim’s machine. Note that, for attacking hosts that are yet to be compromised, the victim could be used as a stepping stone. The incoming SRTP stream and any outgoing packets caused by shell commands should be difficult to correlate, but the Server can add random delays to make it even more challenging. Finally, the Agent resides on the stepping stone and it relays the packets between the Client and the Server. For testing with multiple stepping stones in a connection chain, we need to install the Agent on each of the nodes involved.

We now describe each of the three components of our architecture, as well as our mechanism for ensuring connection robustness without excessive delay.
4.1.1. The Client

The Client program issues the attacker’s commands that are to be run on the victim’s machine. It uses separate threads for sending (the sender) and receiving (the receiver). The sender continuously sends packets at a constant rate over a UDP socket, while the receiver simultaneously receives packets on the same socket and prints them out on the screen. The sender monitors the attacker host’s standard input and prepares any typed commands to be sent to the Agent in the next time slot. If any commands or replies require more data than is available in our fixed packet size, we simply break the data into multiple packets and queue them all for sending one at a time in the constant rate stream. Whenever there is no command available, it keeps sending chaff packets. To create chaff packets, we simply fill the payload with “NUL” characters. Thus, the chaff packets are filtered out by the recipient by testing for the presence of all “NUL” characters in the payload. The use of AES counter mode prevents the observer from determining which packets are chaff and which are real.

4.1.2. The Server

The Server program responds to each received command by executing the command on the victim’s machine and sending back the response to the Client. It has a two-thread design, similar to the Client. The sender forks a child process that creates a shell and ties its standard input, output, and error descriptors to input and output pipes. When the receiver receives a command on its UDP socket, it writes to one end of the input pipe. On execution of the command, the shell writes the output on the output file descriptor. The sender listens on the other end of the output pipe and sends the data back on the socket. Similar to the Client, the sender sends at a constant rate, sending data only at designated sending times and sending chaff packets when no data is available.

4.1.3. The Agent

The Agent program receives data from the previous host in the connection chain and sends it to the next host. It acts as a relay, employing one of the algorithms described in Section 3.1 to selectively forward or drop packets. It
consists of four threads that do the work of sending and receiving in each direction and we call them client-side or server-side threads. The prefix client-side or server-side indicates that the data is coming from the direction of the Client or Server respectively but it may come from an Agent on another host.

Also important are the FIFO queues for each direction, which we call the upstream buffer (client-side to server-side) and the downstream buffer (server-side to client-side). Whenever any data appears on the client-side socket, the client-side receiver puts the packet in the upstream buffer. The server-side sender picks up a packet from the buffer in the next available slot and sends the packet towards the Server. Similarly, the server-side receiver collects responses from the server-side socket and puts them in the downstream buffer, from which the client-side sender gets them and sends them towards the Client. The architecture of Agent is depicted in Figure 3.

4.1.4. A Low-cost Robustness Mechanism

RTP and SRTP do not provide any quality of service guarantees. They don’t guarantee timely delivery or prevent out-of-order delivery. To run terminal commands over the network, however, TCP-like guarantees must be provided. To this end, the attacker would be smart to implement a retransmission protocol.
based on acknowledgments and time-outs, as in TCP. For an overlay network, however, with non-trivial drop rates between stepping stones, the latency cost in waiting for a reactive retransmission (i.e. based on a missing acknowledgment) would be quite high.

To remedy this, we observe that a large number of packets in this system are chaff. Thus, we can use those packets to help with the retransmission problem through pro-active retransmission of all packets in place of chaff. By simply sending enough copies of each packet, we substantially reduce the chances of needing a reactive retransmission and thereby reduce the latency between typing a command and receiving the response.

For our simple proof-of-concept implementation, we do not implement reactive retransmissions. We do, however, implement and test pro-active retransmission in the following way. For each packet containing a real message, we send \( R \) copies of the same packet over the network in succession while maintaining the constant packet rate. This increases the probability of the message reaching the destination without a reactive retransmission. We call \( R \), the number of copies of a packet sent over the network, the redundancy number. For any network conditions, we can achieve the desired limit on delays due to reactive retransmissions by increasing the redundancy number. Sending additional copies of the same packet doesn’t increase the work for the system, as the system will continually send packets to maintain a constant rate stream.

So as to not execute commands repeatedly on the Server, we must remove duplicate packets. We track the packets by storing a unique internal sequence number (ISN) in each packet sent (including chaff packets). For a 64-byte payload, the first two bytes are used for storing the ISN as demonstrated in Figure 4. The ISN is different from the RTP sequence number, which allows the receiver to reconstruct the sender’s packet sequence. We mark the packet number as received in a bit vector, according to its ISN, and we drop any duplicate packets that arrive.

Reactive retransmissions will certainly be needed in some cases, such as when passing large amounts of data. However, for applications like transfer of small
files, error correcting codes (ECC) could be used to ensure complete data transfer without as much redundancy and without much reactive retransmission [34]. In general, nearly any application could be supported if the bandwidth requirements are low relative to the mimicked multimedia stream.

To facilitate pro-active retransmissions, we slightly modify Algorithm 2. The receiver, which handles packet dropping in Algorithm 2, drops all redundant packets (step 3). The sender then creates the redundancy number worth of send events for all real packets to reestablish redundancy for the next network hop (step 4). This simple step ensures full redundancy between each connection.

5. Experimental setup and Results

In this section, we describe our experimental setup for testing the SNEAK stepping stone prototype and the result of those tests. We tested our prototype on both the DETER testbed and the PlanetLab testbed. DETER provides a controlled environment and tools to emulate different network conditions, whereas PlanetLab provides a real Internet-based environment with nodes in different geographical locations. We first describe the experimental setup and results on the DETER testbed and then we describe the experimental results on the PlanetLab testbed. Our main focus was on testing the robustness and the usability of SNEAK for attackers, in terms of the delay caused by buffering, as well as the effect of redundancy in cross traffic environment.

5.1. Experimental design and result on DETER testbed

We first describe the experimental design and the results we obtained on the DETER testbed. In the Internet, there is cross traffic with different characteristics such as constant- or variable-rate streams, variable packet sizes, and a
variety of protocols including DNS, FTP and HTTP. To test our prototype in such an environment, we used the SEER traffic generator tool, which is capable of generating 12 different traffic types with different IPDs, i.e. think times. For our experiments, we chose three categories of traffic, constant bit rate, FTP, and HTTP, with think times as Max/Min or exponential. We used average cross traffic rates of 10 Mbps, 40 Mbps, or 80 Mbps on links with 100 Mbps capacity. The packet transmission rate of our stepping stone prototype was fixed at 40 packets per second, a rate of 20.5 Kbps, and the capacity of the upstream and downstream buffers was fixed at 10 packets per buffer.

For our measurements, we report results from 15 sessions, each with 20 commands, for a total of 300 commands. We did this for $R = 1$ to $R = 6$.

5.1.1. Robustness

To test whether the scheme described in Section 4.1.4 provides robustness to minimize the need for reactive retransmissions, we tested the effect of the redundancy number on the success rate of the queries. We include results for both Algorithm 1 and Algorithm 2, as described in Section 3.1. We define the success rate as the percentage of commands that are executed on the victim and
for which a response is returned correctly without any reactive retransmissions needed. To evaluate this, we issued a number of commands that would result in a single reply packet being sent, such as `hostname` and `date`. We have also estimated the success rates for commands resulting in multiple reply packets such as `ls` and `dir`, which generate 80 packets on average in our experiments. We emphasize that this robustness is only designed to minimize delay due to retransmissions; a complete SNEAK Client would include other robustness measures to ensure packet arrival.

When there is no cross traffic, we achieved 95% to 100% success rates, as shown in Figure 6. With 80 Mbps cross traffic, the success rate for redundancy $R = 1$ and Algorithm 2 drops to 20% and increases gradually for higher redundancy numbers, reaching 98% for $R = 6$ as shown in Figure 7. Thus, even for busy networks, a redundancy number of six limits reactive retransmissions to just 2%. For Algorithm 1, we found that we could get close to 100% success rates for $R = 6$. The cost of this approach is a discernable increase in response.
Our findings indicate that no more than two packets have been present in any of the buffers when there is no cross traffic. During adverse cross traffic of 80 Mbps, up to nine packets have been reported in a buffer at any one time. In all cases, the memory footprint of the system remains very low. If there are more packets for each command or each reply from the Server, then increased buffering may be a suitable trade-off to ensure fewer reactive retransmissions.

5.1.2. Usability

We tested the usability of the system by first using it to execute a series of commands. We found the response times to be reasonable for interactive use in typing commands and getting responses from the Server. We also measured the response time of the system from the time the command is sent to the time the response is received, subtracting out the time to perform the commands on the Server. We observed during the experiments that the response time of the system is largely dependent on the cross traffic present in the system. The
average response times for a session of 10 commands ranged from 95 ms to 121 ms when there is no cross traffic and between 167 ms to 250 ms with 80 Mbps cross traffic.

5.1.3. Watermark evasion

We now show the results of attempting watermark-based detection against our system. We use the watermarking technique of Wang et al., with 16-bit watermarks [7]. The key metric is the bit difference, effectively the Hamming distance between the injected watermark code and the observed code; a bit difference of zero means that the watermark was decoded from the stream without error. To give detection results, we set a maximum allowed bit difference for detection and determine the rate with which watermarks are correctly decoded from the stream; this follows the procedure of Wang et al. [7]. False positives cannot be tested directly; unlike the experiments of Wang et al. [7], we do not have multiple watermarked streams to distinguish, as only stepping stone traffic passes watermarks through the network. Thus, we assume that unwatermarked outgoing streams would appear to have random watermarks. Note that the decoding process of Wang et al. will find a watermark bit string in any stream, whether it is present or not. We thus compare the detection results against the expected detection rates on random bit strings. For a bit difference of $b$ out of 16, the probability of detection is given by $P_d = C(16, b)/2^{16}$, where $C(n, k)$ denotes “n choose k.”

First, for reference, we give results for watermarking without our defenses. Our results show a 100% detection rate for a bit difference of two when there is no buffering and dropping of packets at the Agent. A bit difference of two would only result in an expected 0.2% false positive rate. With a bit difference of zero, we get a 98% detection rate and an expected 0.001% false positive rate. Clearly, Wang et al.’s watermarking technique would work well against a version of SNEAK that appeared to be like a multimedia stream but without buffering.

When buffering is added, watermarking fails. Our plots show the frequency counts for bit differences between the watermark embedded in the stream and
the watermark detected after passing through the Agent. We only show results for watermarks sent across a single Agent; multiple Agents and network hops would increase the changes in traffic and reduce the chance of detection. For comparison with expected false positives, we also plot the expected bit difference for a random process; the closer the results are to random, the harder it would be for the observer to track the session.

Figure 8(a) shows the results of watermark-based detection tests when there is no cross traffic, compared with random watermarks, while Figure 8(b) shows the bit difference with 80 Mbps cross traffic. The bit difference frequency in both cases is very similar to random. In particular, for no cross traffic and maximum bit differences of four or five, the respective detection rates are 2% and 8%. The detection rate for random with no cross traffic is actually slightly higher (3.5% and 9.5% respectively). Thus, detection is no more effective than random guessing. Put another way, allowing a maximum bit difference of five for detection would result in only a 8% detection rate while generating false positives for over 9.5% of all streams.

We discuss the possibility of detection further in Section 6.

5.2. Experimental design and result on PlanetLab testbed

We now describe the results we obtained by testing our prototype on the PlanetLab testbed. A real stepping-stone attack would likely involve nodes at a considerable geographical distance from each other, and hence we would expect significant latency and jitter. To test our software against such an environment, we chose nodes that were spread across the United States and we used two hops to attack the victim. For our experiments, our Client was placed on isec.uta.edu (Dallas, TX area) and we used two Agents which were placed at ricepl-1.cs.rice.edu (Houston, TX) and planetlab1.uc.edu (Cincinnati, OH), respectively. The Server was placed at ricepl-3.cs.rice.edu (Houston, TX). We selected these nodes for their relatively high performance among PlanetLab nodes we tested. As PlanetLab uses virtualization, many nodes introduce high delays for interactive sessions even when no buffering is used. The packet
(a) No cross traffic

(b) 80 Mbps cross traffic

Figure 8: **DETER**: Comparison of frequency counts for watermarking bit differences with SNEAK and for random guessing.

28
rate was same as for the DETER testbed.

For our measurements, we report results from five sessions, each with 10 commands, for a total of 50 commands. We did this for redundancy numbers of $R = 1$ to $R = 6$. We used Algorithm 2 to make packet dropping decisions based on the better performance we observed using it on the DETER testbed.

5.2.1. Robustness

Figure 10 shows the results for robustness, as defined above in Section 5.1.1, on PlanetLab. With a redundancy number of three packets per query, we see a 98% success rate. The success rate rises to 100% when we have a redundancy of $R = 6$. Thus, with reasonable redundancy levels, we observe very high success rates and the attacker would need to wait for very few reactive retransmissions.

We also tabulated the upstream and downstream drop rates at each hop, as shown in Table 1. The drop rates are relatively consistent across our experiments. Let us assume that all drops are independent events. If we consider the

---

2Bolot actually shows that the probability of a drop increases if the prior packet was dropped [30]. This would suggest lower success rates than we see here; we assume that losses
Figure 10: **PlanetLab**: The effect of increase in the redundancy number on success rate of queries. The y-axis starts at 60% success rate.

<table>
<thead>
<tr>
<th>Redundancy number</th>
<th>Upstream Drop Agent 1</th>
<th>Upstream Drop Agent 2</th>
<th>Downstream Drop Agent 2</th>
<th>Downstream Drop Agent 1</th>
<th>Success Rate</th>
<th>Predicted Success Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.83%</td>
<td>10.11%</td>
<td>10.02%</td>
<td>7.71%</td>
<td>62.00%</td>
<td>62%</td>
</tr>
<tr>
<td>2</td>
<td>11.81%</td>
<td>10.95%</td>
<td>10.28%</td>
<td>9.49%</td>
<td>92.00%</td>
<td>85.56%</td>
</tr>
<tr>
<td>3</td>
<td>11.73%</td>
<td>9.86%</td>
<td>11.33%</td>
<td>9.40%</td>
<td>98.00%</td>
<td>94.51%</td>
</tr>
<tr>
<td>4</td>
<td>11.60%</td>
<td>11.98%</td>
<td>11.59%</td>
<td>10.68%</td>
<td>98.00%</td>
<td>97.91%</td>
</tr>
<tr>
<td>5</td>
<td>11.52%</td>
<td>11.57%</td>
<td>11.29%</td>
<td>9.44%</td>
<td>98.00%</td>
<td>99.20%</td>
</tr>
<tr>
<td>6</td>
<td>11.48%</td>
<td>11.40%</td>
<td>14.14%</td>
<td>10.89%</td>
<td>100.00%</td>
<td>99.69%</td>
</tr>
</tbody>
</table>

Drop rate for the experiments with redundancy number of one, the chance that a given packet makes it to the Server and a response gets back is only $p = 62\%$. However, with $r$ duplicates, the chance that the query gets through and gets one valid response is $1 - (1 - p)^r$. Allowing for some experimental variation, we see that we get approximately what we would expect from our redundancy approach.

We also tested the effects of running commands with long results. In partic-
ular we ran the command “ls /*”, which resulted in 200 packet responses. We found that with redundancy $R = 6$, we got no more than 1% loss. Thus, the delay due to reactive retransmissions will not be very high.

5.2.2. Usability

We tested response times on PlanetLab in the same way as on the DETER testbed, as described in Section 5.1.2. We observed during the experiments that the response time of the system is largely dependent on the prevailing network conditions and the responsiveness of the PlanetLab nodes. The average response times for a session of 10 commands ranged from 376 ms to 741 ms and were highly variable under the same testing conditions. This variability can also be seen from link loss rates, which also ranged quite a bit, from 0.80% to 4.55% for a given link averaged over five sessions. Note that link loss should be largely independent of our approach and would apply to other multimedia traffic like VoIP. Despite the variable network conditions, our system consistently provides response times of less than one second, which is sufficient for an attacker to get reasonable use from the system.

5.2.3. Watermark detection

We tested watermark-based detection against our system on PlanetLab in the same way as on the DETER testbed, as described in Section 5.1.3. Figure 11 shows the frequency counts for bit differences between the watermark embedded in the stream and the watermark detected after passing through the Agent as compared with probability-based random guessing. Our results on PlanetLab are very similar to random guessing, just as we found on DETER testbed.

6. Discussion

In this section, we discuss key issues in the practical implementation of our system and some possible avenues for finding countermeasures. We note that these issues and countermeasures do not contradict the key finding of this paper: building a system that evades existing timing-based stepping stone detection systems is both possible and practical.
One of the issues is the system’s security from observation. While it is simple in simulation to have the output stream be completely independent from the input stream, an implementation may retain some timing characteristics. For example, receiving a packet may cause the system to delay the sending of another packet while it processes the new arrival. Such correlations are likely to be small due to the I/O nature of the processes, e.g. much less than an operating system’s quantum, which is 10 ms in Linux [35]. More critically, it is very hard to know how such delays would differentiate stepping stones from non-compromised systems with multiple simultaneous multimedia streams. However, such possible information sources should not be dismissed.

The use of multiple stepping stones raises a number of usability considerations. If using this kind of buffering increases the network costs to the attacker, using multiple stepping stones could be difficult for interactive sessions. Based on our simulations, we believe that each buffered hop would add no more than 50ms of delay to the connection. According to classic research in human-computer interaction, response times of less than 100ms are seen as in-

Figure 11: **PlanetLab**: Comparison of frequency counts for watermarking bit differences with SNEAK and for random guessing.
stantaneous, while response times of one second allow the user’s flow of thought to remain uninterrupted [36]. While an attacker will not get instantaneous responses, he will be able to get an uninterrupted flow of thought in many cases using five or fewer stepping stones. We showed in our experiments that using two stepping stones provides reasonable performance despite the challenges faced with interactive connections over PlanetLab nodes. Given the importance of not being detected, we think that the attacker would be willing to accept even higher latency if it helped him escape otherwise effective stepping stone detection methods.

We should also consider the process of identifying appropriate targets for this kind of stepping stone. As mentioned earlier, this only works on nodes which normally might have constant-rate flows such as VoIP or multimedia. In some cases, such as for home PC’s with broadband connections, this kind of target is easy to find, as any user has the potential to start using VoIP, even if they hadn’t before. For other networks, however, it might be more difficult. Anomaly detection or alerts for the use of multimedia, e.g. for policy reasons, might be able to detect this kind of stepping stone when multimedia streams are uncommon in the network or for a given node. If the attacker, as we have assumed, has control of the node, he could monitor the node to see if constant-rate streams are used. He might also use attacks on link-layer address resolution, e.g. ARP spoofing, to try to monitor traffic from other nodes to see if constant-rate traffic is common in the network or if other nodes would be better targets. Of course, these tactics are somewhat risky for the attacker and could lead to more direct discovery.

The issue of node selection suggests some possible countermeasures to the proposed attacker method based on understanding the context of the traffic being observed. We note that smart stepping stone detectors would be able to learn much from correlating the start and stop times of connections at a node. The attacker could avoid this by introducing random delays before making the second connection. He could also have the next node in the stepping stone chain initiate a connection to the stepping stone, e.g. using a cron job and a
special program for that purpose. There are limits, however, to the amount of delay between each connection and all connections to the stepping stone must overlap in time for interactive use. In general, multiple connections into a node involving multimedia and/or interactive terminal sessions might automatically raise suspicions. Such suspicions could be combined, for example, with other inputs to enhance anomaly detection methods. Blocking multimedia traffic to a subset of nodes could make detection easier by forcing possible stepping stones into a smaller set of nodes.

Finally, we should consider the type of multimedia used as masking traffic. We have mentioned the use of VoIP, which has a number of desirable characteristics for the attacker: widespread use, the common use of peer-to-peer communication and encryption (mostly due to the use of Skype), two-way streams, and simultaneous multi-peer streaming (due to conference calls). The attacker would prefer to use multimedia that is common and would not raise suspicions just because of its use. Peer-to-peer mode of operation and encryption are important to prevent detection methods that raise alarms when multimedia traffic comes from unusual sources or has unusual data patterns. Such a detection method would need to balance its capabilities against the cost of false alarms.

Peer-to-peer file sharing, which have also seen greater use of encryption for *darknets* like Waste (see http://waste.sourceforge.net/), might prove to be another application that would work as masking traffic if system administrators allow their use. Again, stepping stones need not be on tightly secured systems, so such applications cannot be ruled out. Streaming media does not work well right now as a masking technology, but may become useful if streaming becomes more common between peers using systems like Freecast (see http://www.freecast.org). We note that two-way streams are not required, as they can be replaced by two independent one-way streams. However, this requires more compromised hosts and perhaps more complication at the network level. Also, the masking stream must have multiple simultaneous streams for the attacker to avoid detection purely on the basis of the stepping stone connecting to two different peers. The attacker can partially mitigate this by
using regular SSH for every other link in the connection chain.

7. Conclusion

In this paper, we introduced a method by which a hacker could evade stepping-stone detection schemes by disguising his traffic as multimedia and buffering it to remove timing characteristics. Building on prior work in anonymity and stepping-stone detection, the technique stops all timing-based detection schemes, as the timing of the outgoing connection is independent from that of the incoming connection. We developed a prototype stepping-stone application called SNEAK and performed experiments with it in both the DETER network testbed and on the PlanetLab testbed. We showed that the technique successfully removes watermarks from the stream, defeating an otherwise robust and effective watermarking technique. Further, the prototype is usable over multiple stepping stones in both networks.

Although our paper largely discusses evasion techniques, our algorithm could be applied by network administrators to secure their networks. As discussed by Gianvecchio and Wang [37], the watermarking technique proposed by Wang, Chen, and Jajodia [6] can be considered to be a type of covert channel. Thus, our technique also breaks timing-based covert channels present in the network. Additionally, our technique might be useful for strengthening anonymity systems against watermark-based attacks. In both cases, however, there is a substantial cost that the system would pay for the increased security in terms of bandwidth and latency. The attacker in our system does not pay for the bandwidth and has rather low bandwidth and usability needs, making it more practical in the scenarios we have outlined.

There is a growing body of work in watermarking for both stepping stone detection and attacking anonymous communications. We have shown here that advances in this direction could be undone by smart attackers. Nevertheless, existing stepping stone detection techniques are very valuable if they force attackers to use more complicated techniques like the one we have proposed in this
paper. We have described possible countermeasures that are based on examining the larger context of the traffic, as the attacker has been pigeonholed into certain types of multimedia traffic. We believe that future detection mechanisms will need to take this larger view to be successful against wary and resourceful adversaries.

8. Acknowledgements

Thanks to the people who develop and maintain DETER and PlanetLab, without whom this research would not have been possible. In particular, the staff of the University of Southern California’s Information Sciences Institute (ISI) helped greatly with our initial use of DETER.

Thanks to Hao Che, Donggang Liu, and Mike J. O’Dell who provided helpful comments to improve the two MS theses that led to this paper. Many UT Arlington students helped us in ways large and small to produce this work. Of special note are Aniket Pingley and Nayantara Mallesh. Thanks also to the anonymous peer reviewers who have helped us to improve the paper in myriad ways, particularly with clarifying the purpose and value of our research efforts.

This work was supported in part by the National Science Foundation under award number CNS-0549998 and NSF CAREER award number CNS-0954133. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

References


APPENDIX A: Packet Dropping Algorithms

**Input:** Incoming watermarked packet stream  
**Output:** Constant rate packet stream

/* Receiver thread */
begin
  Start receiving packets from previous hop with sequence number i;
  foreach packet_i where i ← 0 to n do
    if packet_i contains all NULs then
      Discard packet_i as dummy;
    else
      Enqueue packet_i in buffer;
    end
  end
end

/* Sender thread */
begin
  Start sending packets to next hop;
  Trigger send event after every ipd µs;
  foreach send event triggered do
    if buffer.length() > 0 then
      if buffer.length() ≥ 2 then
        Drop current packet;
      end
      Send current packet;
    else
      Remove current packet from the buffer;
    end
    Send dummy packet;
  end
end

**Algorithm 1:** SNEAK algorithm with sender side drops
Input: Incoming watermarked packet stream  
Output: Constant rate packet stream  
/* Receiver thread */

begin  
Profile connection to get $\sigma$ using $ipd$ as mean;  
Start receiving packets from previous hop with sequence number $i$;  

foreach packet$_i$ where $i \leftarrow 0$ to $n$ do  
  $t_c$ <- received packet timestamp;  
  $IPD_i$ <- $t_c$ - $t_p$;  
  if $IPD_i$ $\geq 3\sigma + ipd$ then  
    Drop packet$_i$;  
  else  
    if packet$_i$ contains all NULs then  
      Discard packet$_i$ as dummy;  
    else  
      if packet$_i$.id() $\in$ IDList then  
        Discard packet$_i$ as duplicate;  
      else  
        Add packet$_i$.id() to IDList;  
        packet$_i$.sent $\leftarrow$ 0;  
        Enqueue packet$_i$ in buffer;  
      end  
    end  
  end  
  $t_p$ $\leftarrow$ $t_c$;  
end  

end  

/* Sender thread */

begin  
Trigger send event after every $ipd$ $\mu$s;  

foreach send event triggered do  
  if buffer.length() $> 0$ then  
    Send current packet, packet$_i$;  
    packet$_i$.sent $\leftarrow$ packet$_i$.sent + 1;  
    if packet$_i$.sent $\geq r$ then /* redundancy number $r$ */  
      Remove packet$_i$ from buffer;  
  else  
    Send dummy packet;  
  end  
end  

Algorithm 2: SNEAK algorithm with receiver side drops