Keyboards and Covert Channels

Gaurav Shah, Andres Molina and Matt Blaze
Department of Computer and Information Science
University of Pennsylvania
{gauravsh, andresmf, blaze}@cis.upenn.edu

Abstract

This paper introduces *JitterBugs*, a class of inline interception mechanisms that covertly transmit data by perturbing the timing of input events likely to affect externally observable network traffic. JitterBugs positioned at input devices deep within the trusted environment (e.g., hidden in cables or connectors) can leak sensitive data without compromising the host or its software. In particular, we show a practical *Keyboard JitterBug* that solves the data exfiltration problem for keystroke loggers by leaking captured passwords through small variations in the precise times at which keyboard events are delivered to the host. Whenever an interactive communication application (such as SSH, Telnet, instant messaging, etc) is running, a receiver monitoring the host's network traffic can recover the leaked data, even when the session or link is encrypted. Our experiments suggest that simple Keyboard JitterBugs can be a practical technique for capturing and exfiltrating typed secrets under conventional OSes and interactive network applications, even when the receiver is many hops away on the Internet.

1 Introduction

Covert channels are an important theoretical construction for the analysis of information security, but they are not often regarded as a significant threat in conventional (non-MLS) networked computing systems. A covert channel allows an attacker that has compromised a secure system component to leak sensitive information without establishing its own explicit connection to the outside world. Covert timing channels, for example, may exist if there is flexibility in the timing or sequencing of externally observable events (such as disk accesses or delivery of data packets). Covert channels are notoriously hard to detect or eliminate, but this is somewhat ameliorated by the fact that their bandwidth is often rather low, and, in any case, exploiting them requires that the attacker somehow compromise a sensitive system component in the first place. The sensitive system component typically gives the attacker total control over the system or an output channel, making the threat of covert channels relatively minor compared with that of whatever software vulnerability which made such a compromise possible in the first place. Outside of those intended explicitly to support multi-level security, conventional general purpose commercial operating systems, network components, application software, and system architectures largely ignore the threat of covert channels.

In this paper, however, we suggest that typical general purpose computing systems are indeed susceptible in practice to certain covert timing channels. These channels require only the compromise of an input channel or device and can leak sensitive information (such as typed passwords and encryption keys) through the network interface. Furthermore, this can remain a threat even under conditions that intuitively seem quite unfavorable to the attacker, where there is only an indirect, multi-stage link between the compromised system component and a receiver placed many hops away on the Internet.

Specifically, we investigate *loosely-coupled network timing channels,* in which a compromised input device is separated from a covert receiver by multiple system layers at different levels of abstraction. Each of these layers adds noise to the timing of received events through normal internal propagation delays, event scheduling, and buffering. The receiver is assumed only to passively measure the arrival times of some subset of network packets but otherwise has no access to sensitive data. We introduce *JitterBugs,* a class of mechanisms that exploit such channels. A JitterBug must have access to sensitive information along with the capability to modulate event timing. JitterBugs can thus capture and store this sensitive information and send it later through a loosely-coupled network timing channel. Loosely-coupled timing channels and JitterBugs provide a practical framework for exploiting timing channels that exist in general-

purpose computing systems.

In particular, we built a hardware keylogger, the *Keyboard JitterBug,* that can leak typed passwords over the Internet without compromising the host or its OS, without the use of a separate communication channel, and without the need for subsequent access to the device by the attacker. The Keyboard JitterBug is intended as an interesting artifact in its own right (demonstrating a practical attack tool that can operate under highly constrained conditions), but also as a platform for studying the propagation of timing information across hardware, operating systems, network stacks and the Internet. Assuming that the user is running an interactive network application (e.g. SSH, X-Windows), it can leak previously captured passphrases over such network connections. We show using experiments that one can get good performance independent of the OS, system and network conditions under which the Keyboard JitterBug is deployed. Such a device is therefore very robust against any changes in its environment. Keyboard JitterBugs also raise the threat of a *Supply Chain Attack*. In this attack, a powerful adversary subverts a large number of keyboards in the hope that a target of interest acquires one.

2 Related Work

A common simplifying assumption in the covert channel literature is that the attacker has direct control over the timing of the events being measured by the receiver. That is, the attacker is usually assumed to compromise important system components that allow partial or total access to the output subsystem. While this may be a useful conservative assumption for those concerned with minimizing covert channel bandwidth or for abstractly modeling information leakage, we note that those seeking to *exploit* a timing channel may be able to do so more indirectly. In particular, network packet timing is influenced by many system components outside a host's network subsystem, including input devices. Event timing information is propagated from one layer to another, eventually reaching the external network, where it can be measured by an adversary. We are not the first to observe that packet timing can leak sensitive information about non-network subsystems, which has been effectively exploited in remote timing "side channel" attacks against crypto systems [10] and for host fingerprinting [26, 8, 9]. Here, however, we are concerned not with incidental side channel leakage, but with leakage deliberately introduced (perhaps at somewhat higher bandwidth) by a malicious adversary.

The term "covert channel" was first used by Lampson [27] in describing program confinement to ensure processes were not able to leak private data to other processes. Covert channels have primarily been studied in the context of multi-level secure (MLS) systems. MLS systems have multiple security clearance levels. A "HIGH" level should be able to access any data at "LOW" level but not vice-versa. It is thus important that there be no covert channels that allow a rogue agent (e.g. software trojan horse, spy) to transfer information from "HIGH" to "LOW". As a result, some of the earliest research in covert channels was from the perspective of these systems. Due to resource sharing and some commonly used MLS primitives, getting rid of covert channels in such systems is often very hard and in some cases, effectively impractical [38, 33].

Identification of covert timing channels is concerned with enumerating all possible covert channels that might be exploited by a software or the user. The US Trusted Computer System Evaluation Criteria [2] requires explicit covert channel identification in any system certified at class B3 or higher. Many methods have been proposed to identify covert channels, e.g. dual-clock analysis [46], shared resource matrix [24], high-level scenarios [17]. Note that none of these methods guarantee that all covert channels will be found, and, more importantly, identified channels may represent an exploitable threat.

Once practical covert timing channels have been identified, it is often necessary to take steps to mitigate them. Mitigation of timing channels involves either neutralizing the channel completely or reducing its bandwidth to acceptable levels. The first step in covert channel analysis typically involves estimating the worst case bandwidth and the effect of various system parameters like noise and delay [35, 6, 16, 43, 34]. Once this is done, there are many ways in which channel bandwidth can be reduced, including the network pump [20, 21, 22], fuzzy time [18], timing jammers [16] and language transformations [5]. Reducing the bandwidth of covert channel does not imply that the covert channel threat is removed. Useful and important information like encryption keys can still be leaked out over low-bandwidth covert channel [31].

Because it is often not practical to neutralize covert timing channels completely, it might be preferable to detect their active exploitation rather than suffering the performance penalties associated with reducing their potential bandwidth [11]. The detection of network timing channel exploitation is known to be a difficult problem in general. Although specific methods [11, 7] have been proposed to handle various covert encodings, they do not work against every kind of timing channel. All these mechanisms rely on some notion of "regularity" to distinguish between regular network traffic and covert timing channel traffic. The exact regularity depends on the specific channel encoding to be detected and therefore, none of these methods work for every possible scheme.

Side-channel attacks against cryptosystems are some-

what similar to covert channels. Side-channel attacks exploit information leaked by an application's implementation of a crypto algorithm. By measuring the time it takes to perform different cryptographic operations and a knowledge of the implementation, it is sometimes possible to extract key bits [25]. It has been shown that side-channel timing attacks can be practical over a network [10]. Side-channel leakage can also occur in contexts outside of cryptographic algorithms themselves. Song et al. [41] describe a timing attack on the inter-keystroke timing of an interactive SSH connection. Their experiments indicate that one can gain 5.7 bits of information about an SSH password from the observed inter-keystroke timings over a network, assuming a password length of 8 characters. This corresponds to a 50x reduction in work factor for a bruce-force attack.

In fact, the most commonly studied examples of network timing channels in the recent literature are cryptosystem side-channel attacks. Here, the amount of information leaked per packet is very small but given sufficient data and large enough samples, it is possible to perform effective cryptanalysis [23].

Actual malicious attacks exploiting covert channels have not been commonly reported in the literature. Covert *storage* channels exploiting unused TCP/IP header fields have been used in the past by DDoS tools [13]. We are not aware of any public reports documenting the use of malicious covert network *timing* channels in the wild over the Internet, although it is at least plausible that they too have been exploited as part of real attacks.

Given the high variability in round trip times of network packets and their unreliable delivery mechanisms without any QoS guarantees, it is natural to ask whether covert timing channels are even practical on the Internet. Surprisingly, there has been relatively little research on the practical exploitation of covert network timing channels. Cabuk et al. [11] describe the design of a simple software-based network timing channel over IP. Because the timing channel is software based, the sender of the channel has complete control over the network subsystem. Their timing channel uses a binary symbol encoding where the presence or absence of a network packet in a timing interval signifies a bit of information.

The idea of perturbing the timing information of existing network traffic is not new. Addition of timing jitters to existing network packets has been studied previously for SSH stepping stone correlation [45] and for tracking VoIP calls [44]. VoIP tracking relies on encoding timing information in VoIP packets to encode a 24-bit watermark that can then be used to correlate two separate VoIP flows. This is made possible by exploiting the regularity of VoIP traffic and modifying the statistical properties of groups of packets to encode bits. Some timing attacks on anonymizing mix networks also rely on perturbing flows [36, 30].

3 Input Channels and Network Events

In the discussion that follows, we use the following terminology while talking about covert network timing channels. The *sender* of the channel is the subverted entity that is responsible for modulating the timing to encode information. It can be an application software, part of the operating system or a hardware device. The *receiver* in the channel can either be a network connection endpoint or a passive eavesdropper that extracts information from the channel by looking at network packet timings.

The sender in a covert network timing channel aims to modulate the timing of packets on the network to which the receiver has access. This may, for example, be the result of a software trojan that generates network packets at specific times corresponding to the information being sent [11]. Similarly, a router in the path of a network packet can change [44] the timing of the packets it receives before sending them to their destination. In both these examples, the sender of the timing channel has complete control over the network packets and can directly influence their timing on the network. Ideally, when the network delay is negligible, the receiver of the timing channel observes the same timings as those intended by the sender. Thus, the sender of the covert channel is a part of an already compromised output channel or device. Research in practical network timing channels typically considers such direct channel senders. This threat model, however, is overly conservative. It is possible to have usable and practical network timing channels that require only the compromise of system components that have traditionally been thought to lie comfortably within a host's security boundary: the input subsystems.

That the subversion of an input channel or device is a sufficient condition for a practical network timing channel to exist is a somewhat surprising claim. However, once we consider that many network events are directly caused by activity on input channels, it is easy to see how such covert channels might work. Also, because we are just interested in timing, we only need to modify the timing of existing input events. It is not necessary to generate any new traffic.

From the attacker's perspective, the goal of a covert channel is to leak secrets in violation of the host's security policy. Compromised input devices expose any secrets communicated over the input channel. For example, compromising a keyboard (used by the Keyboard JitterBug) allows the attacker to learn passphrases and other personal information that can then be leaked over the covert network timing channel.

In fact, compromising an output channel to leak secrets over a covert channel is not a very interesting scenario for the attacker. Once such a channel or device has been infiltrated by an attacker, leaking secrets from it is very easy. A compromised output subsystem has many options for communicating with an unauthorized receiver, often at much higher bandwidth than a covert channel could provide.

Input based channels do not fit well within the traditional model used in covert channel analysis. As we will see, their presence – as well as the fact that they can be exploited in practice – makes it necessary to include input devices in the Trusted Computing Base (TCB).

The coupling between input devices and the network is made possible by timing propagation often present in general purpose computing systems. Once these channels have been identified, they can be exploited with a JitterBug.

4 Networks and JitterBugs

Loosely coupled network timing channels and JitterBugs are a way of thinking about covert network channels in conventional computer architectures that emphasizes their potential for exploitation. As such, they also provide a model under which the threat of covert channels in conventional computer systems can be analyzed.

One of the characteristics of the software and router based network timing channel described in the previous section is that the sender and receiver of the channel are closely coupled together.

In *loosely coupled network timing channels*, the sender and receiver might be separated by multiple system layers, each belonging to a different level of abstraction. These channels are based on the observation that, just as data flow occurs in a general computing system, timing information also propagates from one system object to the other. By perturbing this timing information, it is possible to modulate a receiver many stages ahead in this flow. It is easier to see how this can be done by considering an example flow that is exploited by the Keyboard JitterBug.

Consider the case where the user is running an interactive network application. Each keypress triggers a sequence of events. The keyboard sends scan codes over the keyboard cable to the host's keyboard controller. This transmission is not instantaneous and depends on the state of the hardware, whether there's enough space in the keyboard controller buffer, etc. This in turn causes an interrupt to be generated to the operating system. Depending on the operations being performed, there might be a variable delay between when the value is received by the keyboard buffer and when it is read by the operating system. Once the interrupt handling routine has read the value from the keyboard controller, the operating system will typically perform some additional operations (e.g. scan-code → key-code translation) and put this value into a buffer to be read by the user-space network application, typically through a *read()* system call. Once the interactive network software gets the character, it might perform additional processing (e.g. encryption) before requesting the OS to send the character in a network packet. Similarly, additional delays will occur due to the network stack and hardware before the packet is sent out on the network. The timing of the network packet corresponds to the time when the key is pressed and the sum of all these additional delays.

In the above example, the flow of timing information (when the key is pressed) goes through several iterations of these added delays while the data moves through various system layers at different abstractions. Each of these layers adds noise to the timing information by imposing a non-deterministic delay due to their internal scheduling, buffering and processing mechanisms. Loosely coupled timing channels are based on the idea that the timing information can be influenced at any one of these several layers.

As long as the sender of the covert timing channel is positioned somewhere before or within any of these layers, it can modulate event timing to transmit data. The encoding applied by the sender is dependent on the properties of this channel that exists between itself and the receiver. The more the number of layers between the sender and receiver, the weaker is their coupling on the timing channel. A loosely coupled network timing channel is one where the source and the receiver of the timing channel are separated by many such delay inducing layers.

4.1 JitterBugs

JitterBugs are a class of mechanisms that can be used to covertly exploit a loosely coupled network timing channel. They have two defining properties. First, they have access to (and can recognize) sensitive information. Second, they have the ability to modulate event timing over a loosely-coupled network timing channel.

The covert transmission need not performed at the same time the sensitive information is captured. A JitterBug can collect and store sensitive information and replay it later over the loosely coupled network timing channel. A JitterBug is semi-passive in nature, i.e. it does not generate any new events. All modulation is done by piggybacking onto pre-existing events. This also makes a JitterBug much harder to detect in comparison to a more active covert timing channel source. Figure 1 shows the general architecture of a JitterBug.

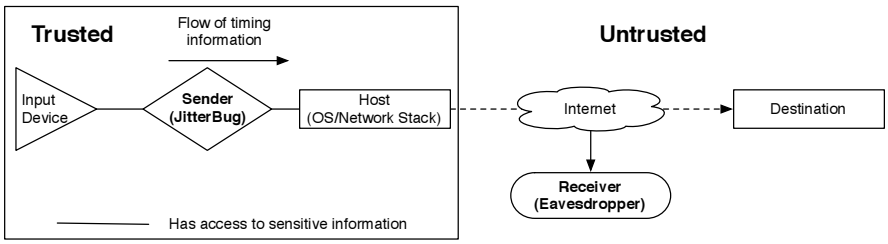

Figure 1: High-level overview of JitterBug

4.2 Example Channels

The keyboard is not the only channel susceptible to exploitation by a JitterBug. Other input peripherals can also provide a suitable environment for a covert network timing channel to exist. Various network computing applications allow users to remotely access hosts on the Internet as if they were being used locally. Some examples of such applications include NXClient, VNC (Virtual Network Computing) and Microsoft Remote Desktop. To minimize lag and keep the response time low, user input is typically transmitted over the network as soon as it is received on the sender's side. This timing channel can be exploited by placing a JitterBug between the communication path of the input device and the computer. Any digital input device – the mouse, digital microphone, web camera, etc. – is potentially exploitable in this way.

Many VoIP implementations support optimizations based on "silent intervals", periods of speech where nothing is being said. Network communication while using VoIP is typically regular. Packets with voice data are sent out at regular intervals over the network. If the silent interval feature is supported, then during periods of silence, packets are no longer sent to conserve bandwidth and system resources. By adding extraneous noise that influences the times at which these silent intervals are generated, a covert network timing channel can exist. In this case, a JitterBug can be placed in the audio interface or behind a digital microphone.

5 Keyboard JitterBug

In most interactive network applications (e.g. SSH, XServer, Telnet, etc.), each keypress corresponds to a packet (possibly encrypted) being sent out on the network. The timing of these packets is closely correlated with the times at which the keys were pressed. The Keyboard JitterBug adds small delays to keypresses that encode the data to be covertly exfiltrated. By observing the precise times packets arrive on the network, a remote receiver can recover this data. The Keyboard JitterBug does not generate any new network packets. It piggybacks its output atop existing network traffic by modulating timing.

The Keyboard JitterBug makes it possible to leak secrets over the network simply by compromising a keyboard input channel. It is, in effect, an advanced keylogger that solves the data exfiltration problem in a novel way.

5.1 Architecture

Our Keyboard JitterBug is implemented as a hardware interception device that sits between the keyboard and the computer. It is also possible to implement a JitterBug by modifying the keyboard firmware or the internal keyboard circuits, but the bump-in-the-wire implementation lends itself to easy installation on existing keyboards without the need for any major modification. Figure 2 shows the high-level architecture of the Keyboard JitterBug.

The Keyboard JitterBug adds timing information to keypresses in the form of small jitters that are unnoticeable to a human operator. If the user is typing in an interactive network application, then each keystroke will be sent in its own network packet. Ignoring the effects of buffering and network delays (the ideal case), the timing of the network packets will mirror closely the times at which the keystroke were received by the keyboard controller on the host. By observing these packet timings, an eavesdropper can reconstruct the original information that was encoded by the Keyboard JitterBug.

5.2 Symbol Encoding

The Keyboard JitterBug implements a covert timing channel by encoding information within inter-keystroke timings. By modifying the timing of keyboard events received by the keyboard controller, the inter-keystroke timings are manipulated such that they satisfy certain properties depending on the information it is trying to send.

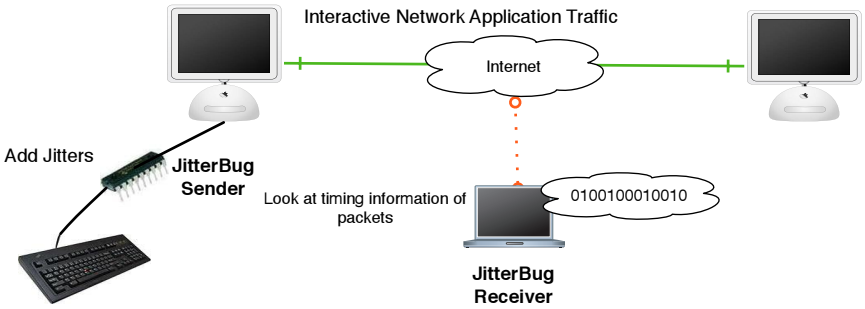

Figure 2: Keyboard JitterBug architecture

The sender and the receiver do not require synchronized clocks but they do need clocks with sufficient accuracy. Our prototype Keyboard JitterBug uses its own crystal controlled clock to govern timing.

Below we describe a simple binary encoding scheme where each timing interval corresponding to adjacent keystrokes encodes a single bit of information.

To encode a binary sequence $\{b_i\}$ using the Keyboard JitterBug, we manipulate the sequence $\{t_i\}$ of times when the keystrokes are pressed by adding a delay denoted by τ_i to each element of this original sequence. The new sequence of events $\{t'_i\}$, where each $t'_i = t_i + \tau_i$, are the times at which the keystrokes are released by the Keyboard JitterBug to the keyboard controller. The resulting sequence encodes information in the differences $\delta_i = t'_i - t'_{i-1}$, such that:

$$\delta_i \bmod w = \begin{cases} 0 & \text{if } b_i = 0; \\ \lfloor w/2 \rfloor & \text{if } b_i = 1; \end{cases}$$

where w is a real-time parameter called the *timing window*.

Therefore, to encode a '0' the delay added is such that $\delta_i \bmod w$ is 0 and to encode a '1', the delay added is such that $\delta_i \bmod w$ is $\lfloor w/2 \rfloor$. In this symbol encoding scheme, within the timing window of length w, $\lfloor w/2 \rfloor$ is the antipode of 0.

Observe that each $\tau_i < w$. Hence, w defines the maximum delay added to each keystroke by the Keyboard JitterBug.

It is easy to understand the encoding algorithm with the help of a simple example. Assuming a window size w of 20 ms, to transmit the bit sequence $\{0, 1, 0, 1, 1\}$, the JitterBug would add delay such that the modified inter-keystroke timings (modulo 20) would be $\{0, 10, 0, 10, 10\}$. So if the (original) observed inter-keystroke timings were (in ms) $\{123, 145, 333, 813, 140\}$, the delay added would be such that the modified inter-keystroke timings are $\{140, 150, 340, 830, 150\}$. Hence, the JitterBug would use the delay sequence $\{17, 5, 7, 17, 10\}$ where each of these individual delays is less than 20 ms.

5.3 Symbol Decoding

For the Keyboard JitterBug network timing channel, the receiver is a passive eavesdropper that needs only the ability to measure the times at which each network packet arrives on the network. There are two ways a receiver might extract this timing information: *TCP Timestamps* and *sniffer timestamps*.

The *TCP Timestamp* option, described in RFC 1323 [19], allows each TCP packet to contain a 32-bit timestamp. This 32-bit TCP timestamp is a monotonically increasing counter and acts as a virtual-clock. In most modern operating systems, the TCP timestamp is directly derived from the system clock. The granularity of this clock depends on the operating system in use. Some commonly used values are 10 ms (some Linux distributions and FreeBSD), 500 ms (OpenBSD), and 100 ms (Microsoft Windows). As TCP timestamps correspond to the time at which the network packet was sent according to the source clock, they are unaffected by network jitter. The chief disadvantage of using TCP timestamps is their much coarser granularity on many operating systems, requiring the use of large timing windows for symbol encoding and decoding. Also, TCP timestamps are only used for a flow if both ends support the option and in addition, the initial SYN packet for the connection contained this option.

Sniffer timestamps, in contrast, represent the times at which packets are seen by a remote network sniffer. Due to network delays, these timestamps are offset from the actual time the packet was sent at the source. In addition, these timing offsets are affected by any network jitter present.

Based on the above discussion, it is clear that the choice of the particular timestamp to use depends on the exact network conditions, timing window size and the placement of the receiver on the network relative to the

covert channel sender. However, we use sniffer timestamps exclusively for our experiments as they provide sufficient granularity for a much wider range of window sizes and operating systems. Also, since the Keyboard JitterBug has no control over the host or its OS, assuming only sniffer timestamps is a more conservative assumption for the attacker.

For decoding, the receiver on the timing channel records the sequence of times $\{\widehat{t_i}\}$ of network packets corresponding to each keystroke. Then the sequence of differences $\{\widehat{\delta_i} = \widehat{t_i} - \widehat{t_{i-1}}\}$, encodes the bits of information being transmitted. To allow the receiver to handle small variations in network transit times due to network jitter, the decoding algorithm allows some tolerance. The tolerance parameter ε is used by the decoder to handle these small fluctuations. The decoding algorithm is as follows:

$$\begin{array}{llll} \text{if} & -\varepsilon < \widehat{\delta_i} \leq \varepsilon & (\bmod w) & \text{then } b_i = 0; \\ \text{if} & w/2 - \varepsilon \leq \widehat{\delta_i} < w/2 + \varepsilon & (\bmod w) & \text{then } b_i = 1; \end{array}$$

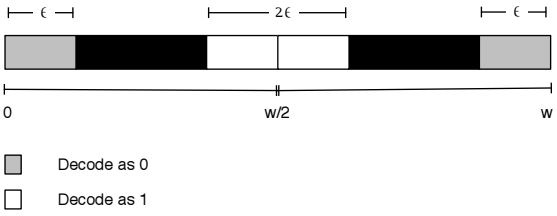

Figure 3: Timing Window for binary symbol decoding

The tolerance ε is an important parameter that decides the length of *guard bands* that compensate for the variability in the network and other delays. Figure 3 shows how the receiver decodes bits based on the inter-packet delays modulo the length of the timing window. The bands used for the decoding are calculated based on the value of ε. From the figure it is easy to see that maximum value of ε is $w/4$. Note that for a particular choice of w and ε, the proportion of timing window allocated for '1' and a '0' may not be equal.

For applications where the total added jitter is an important consideration, the tolerance ε can be used during symbol encoding to reduce the average jitter added at the cost of some channel performance.

The length of the timing window is an important parameter. We want it to be small so as to minimize the keyboard lag experienced by the user. At the same time, we want to make sure the guard bands are large enough to handle channel noise.

Because the receiver uses inter-packet delay and not absolute packet times, there is no need for synchronization between the source and receiver clocks. The clocks, however, need to run at the same rate.

The above scheme allows one bit of information to be transmitted per keypress. However, it is also possible to use a more efficient symbol alphabet with cardinality greater than two by subdividing the window further (instead of just two regions) corresponding to each possible symbol that can be transmitted. This choice however impacts the required granularity of the timing window. More specifically, for an encoding scheme with alphabet $\mathcal{A}$, cardinality k, and a tolerance of ε for each symbol, the timing window w needs to be atleast $2k\varepsilon$ units in length. We experimentally evaluate one such scheme in Section 6.3.6.

5.4 Framing and Error Correction

Our Keyboard JitterBug assumes that there will be bursts of contiguous keyboard activity in the interactive network application generating network packets, though these bursts themselves might be interrupted and infrequent. In our model, the only information sent over the covert timing channel is ASCII text corresponding to short user passphrases. Consequently, we do not perform any detailed analysis of the performance of the channel using different framing mechanisms. However, we tested the Keyboard JitterBug using two very simple framing schemes.

One approach is based on *bit-stuffing* [28], which uses a special sequence of bits known as the Frame Sync Sequence (FSS) that acts as frame delimiter. This sequence is prevented from occurring in the actual data being transmitted by "stuffing" additional bits when it is encountered in the data stream. Conversely, these extra bits are "destuffed" by the decoder at the receiver to recover the original bits of information. The advantage of using bit-stuffing is that it does not require any change to the underlying low-level symbol encoding scheme. For example, the symbol alphabet can still be binary.

An alternative framing mechanism adds a third symbol to the low-level encoding scheme. This special symbol in the underlying transmission alphabet acts as a frame delimiter. Note that if the length of the timing window is kept constant, this reduces the maximum possible length of the guard bands used for decoding the information symbols (0 and 1) compared to a purely binary scheme. So this might lead to lower channel performance if network noise is present. It is also useful to give more tolerance to the frame delimiter symbol encoding as framing errors cause the whole frame to be discarded at the receiver. Thus, delimiter corruption causes a much higher commensurate effect on the overall bit error rate than the corruption of a single bit. This issue is discussed further in Section 6.3.

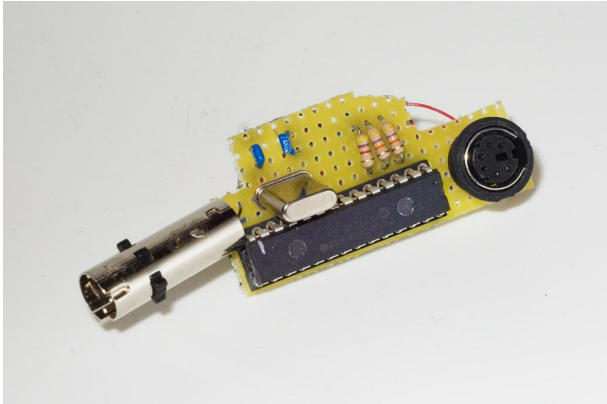

Figure 4: Prototype Keyboard JitterBug

Error correction might also be required if the timing channel suffers from a lot of noise. However, in the simple case in which a short encryption key or password is being leaked, forward error correction is provided inherently by repeating the transmission each time it completes.

5.5 Prototype PIC implementation

We implemented a prototype Keyboard JitterBug on the Microchip PIC18F series of Peripheral Interface Controllers (PICs). The PIC18F series is a family of 12-bit core flash programmable microcontrollers. Our source code is a combination of C and PIC assembly and the final machine code uses less than 5KB program memory. The implementation works for keyboards that use the IBM PS/2 protocol. It should be easy to port the code to other kinds of keyboards, e.g. USB. The bump-in-the-wire implementation acts as a relay device for PS/2 signals. It derives its power from the PS/2 voltage lines and hence no additional power source is required. When enabled, it adds jitters to delay the time at which the keyboard controller receives notification of the keypress. It also supports programmable triggers (as described in Section 5.6) that help identify typed sensitive information to leak over the covert channel. Figure 4 shows our PIC-based prototype implementation. A truly surreptitious Keyboard JitterBug would have to be small enough to conceal within a cable or connector. Since the computational requirements are quite modest here, miniaturization could be readily accomplished through the use of smaller components or with customized ASIC hardware.

5.6 Attack scenarios

We consider a real and somewhat famous example from recent news reports that motivated our design. In gathering evidence in the 2000 bookmaking case [3] against Nicodemo "Little Nicky" Scarfo, the FBI surreptitiously installed some sort of keylogger device in the suspect's computer to gain access to his PGP passphrases. Installing the device apparently required physical access to the suspect's office, a high-risk and expensive operation. Once installed, the device recorded keypresses under certain conditions. This introduced a new problem: retrieval of the captured information. A conventional keylogger must either compromise the host software (to allow remote access and offloading of captured data) or require physical access to recover the device itself. Neither option is entirely satisfactory from the FBI's perspective. Compromise of the host software creates an ongoing risk of discovery or data loss (if the host software is updated or replaced), and physical recovery requires additional (risky) physical access. The Keyboard JitterBug adds a third option: leaking the targeted information atop normal network traffic through the timing channel, obviating the need for subsequent retrieval or compromise of the host.[1]

As the Keyboard JitterBug lies in the communication path between the keyboard and the computer system, it has access to the keystrokes typed in by the user. The covert network timing channel is relatively low-bandwidth and thus the JitterBug needs the capability to recognize and store the specific information of interest with high confidence. JitterBug's programmable triggers do just that by acting as recognizers of sensitive information (like passphrases) and storing this information for sending out later over the covert network timing channel. Programmable triggers allow a Keyboard JitterBug to wait for particular strings to be typed. When such a condition is detected, it stores whatever string is typed next into its internal EEPROM for subsequent transmission.

For example, a Keyboard JitterBug might be programmed with the user name of the target as the trigger, on the assumption that the following keystrokes are likely to include a password. It might also be programmed to detect certain typing patterns that tend to indicate that the user is initiating an SSH connection (e.g. "ssh username@host"). By storing whatever is subsequently typed by the user, the Keyboard JitterBug effectively gets hold of the user's SSH password. The covert channel transmits the password back to the attacker without the need to retrieve the bug; the password can even be sent atop the victim's own encrypted SSH connection.

In this sense, Keyboard JitterBug can be seen as a next step in the evolution of keyloggers. The possibil-

[1] Because the Scarfo case never went to trial, the technology used by the FBI to capture the keystrokes was never publicly disclosed – it may have been a JitterBug, although it was more likely a conventional keylogger. The PGP passphrase of interest turned out to be based on Mr. Scarfo's father's US Prison ID number.

ity of such devices raises obvious privacy and security concerns.

The Keyboard JitterBug implementation can also serve as the basis for more advanced worms and viruses. Many newer keyboards are software programmable. Some of these keyboards even allow their internal firmware to be upgraded by software. A malicious virus program might rewrite the firmware with a JitterBug(ged) version and delete itself, effectively avoiding any form of detection by an antivirus program.

Finally, perhaps the most serious (and also the most sophisticated to mount) application of the JitterBug is as part of a *Supply Chain Attack*. Rather than targeting a specific system, the attacker subverts the keyboard supply and manufacturing process to install such a device in many keyboards from one or more suppliers, in the hope that a compromised device will eventually be acquired by a target of interest. Such an attack seems most plausible in the context of government espionage or information warfare, but could also be mounted by an industrial or individual attacker who manages to compromise a keyboard vendor's code base.

5.7 Non-interactive network applications

Although the Keyboard JitterBug's primary application is for leaking secrets or other information over interactive network applications, it can also be used in a restricted setting with very low bandwidth for less interactive network applications. Much network activity has a causal relationship with specific keyboard events. This is true for many commonly used network programs such as web browsers, instant messengers and email clients.

For IM programs, pressing return after a line of text causes the message to be sent over a network. In addition, many IM protocols also send a notification to the other end as soon as the user starts typing another line. By detecting and manipulating keystroke timings when such events happen, the Keyboard JitterBug can leak information. Similarly, typing a URL into a web-browser typically requires the user to press "return" before the browser fetches it. The Keyboard JitterBug can manipulate this timing to affect the time at which the URL is fetched over the network. The relevant "return" when the jitter should be added can be detected by using a programmable trigger (e.g. Ctrl-L → URL→<return> for Mozilla Firefox). E-mail clients also sometimes use keyboard shortcuts which cause specific network events (e.g. sending an e-mail) to occur. By adding jitter to the appropriate keypresses, the timing of these network events can be manipulated (and observed).

For the above applications, the coupling between keyboard events and network activity make them susceptible to attacks using the Keyboard JitterBug. The bandwidth of leakage, however, is significantly lower. One advantage they have over SSH from the perspective of the attacker is that many of these applications tend not to use encryption. This reduces the number of insertion errors (Section 6.2) by making it easier for the covert channel receiver to distinguish between normal network packets and those whose timing was manipulated by the Keyboard JitterBug.

6 Keyboard JitterBug: Evaluation

In this section, our focus is on evaluating the efficacy of the timing channel under a variety of practical conditions.

6.1 Factors affecting performance

Because the JitterBug is so far removed from its receiver, many factors affect its performance.

- **Buffering:** Keyboard buffering affects the delay between when the key is received by the keyboard controller and when it is available to the application that is trying to send the keystroke over the network. Similarly, network buffering affects the delay between when the request for sending the packet is received by the OS network stack and when it is actually transmitted over the network. If the variance of buffering delay (keyboard + network) is high, then the number of symbol errors increase, reducing the effective bitrate of the channel.

- **OS Scheduling:** For a loosely-coupled covert timing channel, the noise added by OS scheduling depends on a variety of factors including the time quantum used, the scheduling algorithm, system load, etc. Fortunately, keyboard and network handling in most modern operating systems is given high priority and hence, the noise added to the channel from scheduling effects is usually quite insignificant.

- **Nagle's algorithm:** Described in RFC 896 [37], Nagle's algorithm is used to handle the *small-packet problem* that is caused by the increase in packet header overhead when interactive network applications are used as each keystroke is sent in its own network packet. The algorithm is an adaptive way of deciding when to buffer data before sending it out in a single network packet based on the network conditions (latency and bandwidth). If Nagle's algorithm is enabled it can cause two problems with the timing channel. Firstly, it creates a varying network buffering delay that adds noise to the timing information. Secondly, it can lead to multiple

keystrokes being sent out in a single packet. Hence, the timing information for all but the first keystroke might be lost leading to missing symbols in the timing channel. Fortunately, Nagle's algorithm is usually disabled by default (using the TCP_NODELAY option) for better responsiveness in interactive network applications including most commonly used SSH client implementations (e.g. OpenSSH). This means that each keystroke generates it own network packet that is sent out as soon as possible (assuming no network congestion).

- **Network Jitter:** This is the most important factor for a network timing channel. Network Jitter, i.e. variability in round trip times (RTT), adds noise to the timing information and affects the accuracy of symbol decoding at the receiver. The placement of the receiver also affects the "observed" network jitter and thus changes the observed channel accuracy. Encoding a symbol in the timing of two adjacent packets has a mitigating effect on the channel accuracy as each change in network delay causes a maximum of one error to occur.

6.2 Sources of Error

The Keyboard JitterBug timing channel can suffer from three kinds of transmission errors: insertions, deletions and inversions.

Insertions occur when receiver cannot distinguish between network packets corresponding to the Keyboard JitterBug and those corresponding to other network traffic. This will happen when any form of encryption is being used. Depending on the protocol layer at which encryption is being applied, the frequency of insertion errors will be different. The worst case is when link encryption is being used. In this case, it would be very hard to separate covert channel packets with that of normal network traffic, causing insertion errors to happen all the time. Fortunately, the use of link layer encryption along the whole path of a network packet on the Internet is quite rare, so this restriction is not that much of an issue. Encryption at the network or transport layers (e.g. IPSec, TLS) would also cause significant insertion errors to occur, especially if one of the network applications of interest use them for communication. Application layer encryption can cause insertion errors but they are pretty rare as the visible packet format and size (e.g. SSH) makes it possible (in most cases) to distinguish packets of interest from normal network traffic. Finally, if no encryption is being used (e.g. telnet), then no insertion errors occur.

Deletion errors are of two kinds. As the Keyboard JitterBug only has access to keystrokes and no other system information, it is not possible to distinguish between when the user is typing inside a network application of interest or in other applications running on the system. The situation can be ameliorated somewhat by using heuristics to determine when the user is typing in a network application (e.g. by detecting shell commands being typed when previously the user opened up a new ssh connection) and add jitters only then. In cases where this is not possible, multiple chunks of bits might be lost. The second kind of deletion errors occur when network buffering causes multiple keystrokes to be sent in the same packet. These deletion errors occur less frequently and typically cause very few symbols to be lost. They can always be detected when no encryption is being used (e.g. telnet). For the more general case, an appropriate framing scheme would be required.

The main application of the Keyboard JitterBug channel is to leak passwords, typed cryptographic keys, and other such secrets. As these secrets are relatively short, they can be transmitted repeatedly to increase the chances that they will be received correctly. Both insertion and deletion errors are, by their nature, bursty. The redundancy through repetition provides inherent forward error correction (FEC) to handle them.

Finally, symbol corruption errors are caused by delays that might occur on the sender's side or in the network while the packet is in transit (due to network jitter). These errors cause a different symbol to be received than what was originally sent. For the binary symbol encoding scheme, the errors take the form of bit inversions. Symbol corruption errors can be handled by using suitable error correction coding schemes.

As insertion and deletion errors are very specific to the application and environment under which the Keyboard JitterBug is deployed, we do not focus on them in our experimental evaluation.

6.3 Experimental Results

We performed various experiments to test the Keyboard JitterBug under a variety of sender configurations, network and receiver conditions. The experiments were performed with our bump-in-the-wire implementation of the Keyboard JitterBug on a PIC microcontoller.

As our covert channel relies on manipulating the timing of keypresses to piggyback information, the keyboard needs to be in use for the channel to work and be tested. Instead of manually typing at the keyboard for each experiment, we built a keyboard replayer for our controlled experiments. A special mode in the Keyboard JitterBug allows it to store all keyboard traffic into the EEPROM for later replay. Then the covert timing channel can be turned on and the replay information is used to simulate a real user typing at the keyboard preserving

the original user's keystroke timing information. This way we can test different Keyboard JitterBug parameters under the same set of conditions. Note that the Keyboard JitterBug is still placed as a relay device between the keyboard and the computer. The available memory of the PIC device limits the maximum length of the replay. When the end of a replay is reached, the JitterBug starts the replay from the beginning. This does not materially affect our experiments, since we are concerned only with the inter-character timing, not the actual text.

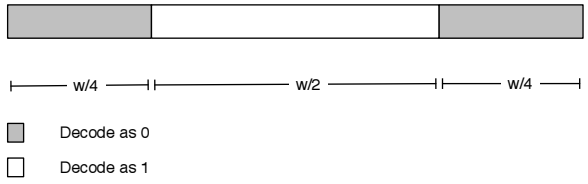

Figure 5: Timing Window ($\varepsilon = w/4$) used for binary symbol decoding in experiments

For all experiments where a pure binary symbol encoding is being used, the user-defined tolerance parameter $\varepsilon = w/4$. Figure 5 shows the decoding timing window used with the bands for '0' and '1'.

The source machines used for the experiments were connected to the LAN network at the Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia. The source machines ran Linux 2.4.20 (unless otherwise noted). All network connections were made via a 100Mbps switch. As we are interested in finding how well the Keyboard JitterBug performs under a range of different network conditions, we used the PlanetLab network [12] to test our covert network timing channel using various geographically displaced nodes. Interactive SSH terminal sessions were initiated between the source and destination nodes. All measurements of the timing information for the covert channel were performed at the destination host using *tcpdump*. Using the time of arrival of network packets at the destination host gives us a worst case estimate of the channel performance. In practice, the covert channel receiver can be placed anywhere in the path of the network packets. The channel is configured to send an ASCII encoded string.

The standard measure of the performance of channel under the presence of noise is the bit error rate (BER) [40]. For channels with bit slips[2], due to the possibility of bit loss, this metric cannot directly be used. For the Keyboard JitterBug, as network buffering can cause more than one keystroke to be sent in each packet, there is potential for missing bits leading to synchronization errors. Therefore, while measuring raw channel performance (without framing or error correction), the traditional definition of bit error rate based on the Hamming Distance metric cannot be used. Instead, we use *Levenshtein Distance*, also called the *edit distance* to get the raw bit error rate. Here, an error constitutes inversion or deletion of bits. The edit distance is a measure of similarity of two strings and is equal to the number of insertions, deletions, and substitutions needed to convert the source string (bits received) into the target string (bits sent).

While measuring channel performance with framing, the bit error rate is calculated using the Hamming Distance metric for correctly received frames. For frames discarded because of framing errors, all the data bits (of the frame) are assumed to have been in error. Because of framing, the receiver can detect and recover from bit deletions and synchronize itself with the covert channel data stream. For evaluating the performance of the channel with framing, three parameters are calculated: Net BER (E_C), Average Correct Frame BER (E_{CF}) and Frame Discard Rate (E_{DF}). Net BER measures the total fraction of bits that are lost or corrupted due to bit errors within a frame or framing errors caused due to corruption of the Frame Sync Sequence or delimiter. Framing errors cause whole frame(s) to be discarded leading to the loss of all bits they contain. These bit errors (equal to the frame size) are included in the calculation for Net BER. Average Correct Frame BER is the average BER only for the frames that were successfully decoded (without framing errors). Therefore, bits lost due to framing errors are not accounted for in calculating the Average Correct Frame BER. The suitable error correction coding scheme to use would depend on this measure. Frame Discard Rate is a measure of the frequency with which frames get dropped or lost due to framing errors. It is easy to see that:

$$E_C = E_{CF} + E_{DF} - E_{CF}E_{DF}$$

6.3.1 Window Size and RTT

Table 1 summarizes the measured raw BER of the covert network timing channel for six different nodes on the PlanetLab network using different window sizes. These nodes were chosen based on their wide ranging geographical distances from the source host and different network round-trip times.

The raw BER is the channel performance without the use of any error correction coding or framing. As the calculation of the raw BER metric uses the edit distance metric, the error rates also consider bit deletions and insertions in addition to inversions. The notion of acceptable raw channel performance would depend on a variety of factors including the framing mechanism used, the application, and the capability of error correction codes.

[2]In general, the lack of synchronization might occur for various other reasons, such as the lack of buffer space, variation in clock rate, etc.

Node	RTT	Hops	1s	500ms	100ms	20ms	15ms	10ms	5ms	2ms
ColumbiaU (NYC, NY)	6 ms	14	0	0	0	.007	.007	.010	.044	.089
UKansas (Lawrence, KS)	42 ms	14	0	0	0	.005	.007	.008	.067	.143
UUtah (Salt Lake City, UT)	73 ms	23	0	0	0	.005	.005	.005	.039	.092
UCSD (San Diego, CA)	84 ms	19	0	0	0	.010	.011	.011	.044	.102
ETHZ (ETH, Zurich)	112 ms	17	0	0	0	.005	.006	.009	.049	.092
NUS (Singapore)	236ms	18	0	0	0	.045	.047	.048	.228	.240

Table 1: **Measured Raw Bit Error Rate for different window sizes and network nodes (Levenshtein Distance Metric)**

Many error correction codes exist for channels where both substitutions and deletions are possible and that use the Levenshtein distance metric as the error rate metric [29]. Marker Codes [15, 39] and Watermark Codes [14] are some examples of such error correction schemes. As our primary application for the channel is very low-bandwidth, we consider a measured raw bit error rate of less than 10% to be acceptable. We discuss channel performance using the Hamming distance metric in Section 6.3.5 when we discuss experiments with the use of some simple framing schemes.

For a fixed window size, the round-trip times and the channel performance do not exhibit any clear trend. Intuitively, this lack of a trend is to be expected. The channel encoding relies on the packet inter-arrival times for encoding the information. Thus, it is the network jitter and not the end-to-end latency that affects performance of the channel.

Acceptable performance is achievable even if the receiver is at a large distance from the source of the timing channel. The node in Singapore, with a RTT of 236 ms, is a case in point. For a window size of 20 ms, the raw channel error rate is around 4.5%, which is quite usable for many low-bandwidth applications of the JitterBug.

The maximum lag introduced by the Keyboard JitterBug for each keypress is equal to the window size w. Consequently, the choice of this parameter is dependent upon how large the value can be made while still keeping the Keyboard JitterBug undetectable by the user. Although we can get a perfect channel for all the nodes tested with a window size of 1 second, this value is effectively unusable because the user will detect the presence of the Keyboard JitterBug. It is widely believed that 0.1 seconds is about the limit for the response time for a user to feel that the system is reacting instantaneously [32]. Therefore in practice, the window size will have to be smaller than that. Our own experience with the Keyboard JitterBug shows that 20 ms is a perfectly acceptable window size and this amount of added lag for each keystroke is effectively unnoticeable by the user.

The window size also affects the size of the guard bands that help absorb some network jitter. The jitter has two components: the frequency of change and the magnitude of change. For a window size of w the implementation can handle a maximum jitter of $w/4$ per packet pair.

Load	20ms	15ms	10ms	5ms
SSH	.010	.011	.011	.044
Telnet	0	.006	.01	.01

Table 2: **Measured Raw Bit Error Rate for SSH and Telnet (Levenshtein Distance Metric)**

From Table 1, it is clear that, as expected, smaller window sizes lead to higher error rates. The increase in the error rate, however, is not very drastic over the ranges we tested. The channel remain usable even if window sizes as low as 2 ms are used. For a window size of 20 ms or more, channel performance is consistently high on all the nodes tested. Our observations are supported by previous studies of round-trip delays on the Internet. It has been shown that on average, round-trip delays on the Internet tend to cluster around within a jitter window of 10 ms for significant periods of time [4]. Thus, this choice of window size is likely to work under a wide gamut of network conditions. When the exact conditions are known, it is possible to optimize the Keyboard JitterBug further by choosing smaller window sizes.

6.3.2 Network application

We measured the raw BER for four different windows sizes for a covert timing channel to a PlanetLab node in University of California, San Diego. The node is 19 hops away with an average Round-Trip Time (RTT) of 84.3 ms. Table 2 shows the measured raw BER for SSH and Telnet. The channel performance is not affected by the choice of the interactive network terminal application. The advantage of Telnet, of course, is its lack of encryption, which makes it easy to detect deletion errors caused by multiple characters being sent in the same network packet.

OS	20ms	15ms	10ms	5ms
Linux 2.4.20	.010	.011	.011	.044
Linux 2.6.10	.010	.010	.010	.013
Windows XP(SP2)	.001	.001	.001	.007
FreeBSD 5.4	.017	.033	.044	.058
OpenBSD 3.8	.022	.043	.05	.075

Table 3: **Measured Raw Bit Error Rate for different window sizes and operating systems (Levenshtein Distance Metric)**

Load	20ms	15ms	10ms	5ms
Idle	.010	.011	.011	.044
Heavy Load	.010	.016	.016	.05

Table 4: **Measured Raw Bit Error Rate for different windows sizes and system loads (Levenshtein Distance Metric)**

6.3.3 Operating System

To confirm that the performance of the channel is not significantly affected by the operating system through which the Keyboard JitterBug is working, we performed experiments to measure the performance of the implementation on several popular operating systems.[3] We again performed the experiments on the PlanetLab node at San Diego, California for four different window sizes. Table 3 summarizes the measured raw BER of the covert timing channel for different operating systems. The raw BER remains quite similar for all the operating systems tested without any major fluctuations. The small difference in the results arises from two factors: variations in network conditions and different OS implementations of keyboard processing. Both these factors affect the amount of noise present in the timing channel when it reaches the receiver.

6.3.4 System Load

Keyboard and network events in general-purpose operating systems are typically given high processing priority. Moreover, their implementation is usually interrupt-driven for better responsiveness and performance. So, we do not expect the normal variation in system loads to have any major influence on the performance of the timing channel. To confirm this, we used the *stress* [1] tool to generate high system loads[4] at the source machine and then measured the performance of the timing channel at the receiver. As before, the receiver of the timing channel is located at the PlanetLab Node in San Diego, CA.

[3]We did not perform experiments with Mac OS X because of the absence of a PS/2 keyboard port on the Mac hardware.

[4]The command-line used was: *stress –cpu 8 –io 4 –vm 2 –vm-bytes 256M*

Node	E_T	E_{CF}	E_{DF}
ColumbiaU (NYC, NY)	.142	0	.142
UKansas (Lawrence, KS)	.152	0	.152
UUtah (Salt Lake City, UT)	.093	0	.093
UCSD (San Diego, CA)	.184	0	.184
ETHZ (ETH, Zurich)	.112	0	.112
NUS (Singapore)	.384	.014	.375

Table 5: **Measured Bit Error Rate(s) with Framing (Bit-Stuffing)** (E_T = Net BER, E_{CF}: Average Correct Frame BER, E_{DF}: Frame Discard Rate)

Node	E_T	E_{CF}	E_{DF}
ColumbiaU (NYC, NY)	.121	.002	.12
UKansas (Lawrence, KS)	.104	0	.104
UUtah (Salt Lake City, UT)	.137	.001	.136
UCSD (San Diego, CA)	.202	.001	.2
ETHZ (ETH, Zurich)	.088	0	.088
NUS (Singapore)	.39	.005	.386

Table 6: **Measured Bit Error Rate(s) with Framing (Ternary Encoding)** (E_T = Net BER, E_{CF}: Average Correct Frame BER, E_{DF}: Frame Discard Rate)

The source of the timing channel is a Pentium 4 2.4 GHz Desktop System with 1GB of system memory running Linux 2.4.20.

Table 4 shows the measured raw BER for normal system load vs. heavy system load. The results show that the behavior of the channel remains quite similar without any drastic drops in the channel performance.

6.3.5 Framing

Many applications of the Keyboard JitterBug would require the use of framing for transmission of data on the timing channel. We tested the JitterBug with two very simple framing schemes: one based on bit stuffing and the other using a low-level special frame delimiter symbol. Our goal is to evaluate the performance of the channel using the Hamming distance metric rather than describe an optimal framing scheme for the timing channel.

The timing window used for the experiments is 20 ms and the frame size is 16 bits. The bit-stuffing frame sync sequence (FSS) used is 8 bits in length. The results are summarized in Table 5 and Table 6. As described in Section 6.3, three parameters are calculated for each run: the Net BER, Average Correct Frame BER and the Frame Discard Rate. The receiver discards any frame that is not the correct size or has a corrupted frame delimiter.

It is clear from the results that the bulk of the network errors are the result of discarded frames. Many of these are synchronization errors caused by deletion of bits from a frame due to network buffering. There are

Node	$\mathbf{E_T}$	$\mathbf{E_{CF}}$	$\mathbf{E_{DF}}$
ColumbiaU (NYC, NY)	.150	.011	.140
UKansas (Lawrence, KS)	.174	.030	.148
UUtah (Salt Lake City, UT)	.170	.012	.16
UCSD (San Diego, CA)	.173	.021	.156
ETHZ (ETH, Zurich)	.153	.007	.147
NUS (Singapore)	.34	.057	.299

Table 7: **Measured Bit Error Rate(s) with high bit-rate encoding (4bits/symbol + frame delimiter)** (E_T = Net BER, E_{CF}: Average Correct Frame BER, E_{DF}: Frame Discard Rate)

many possible ways the framing scheme could be optimized to reduce the frequency of framing errors. Using smaller frame sizes can reduce the affect of discarded frames on the overall BER. One could also use a much more optimistic decoder so that partial frames are not discarded completely but parts of their contents are recovered. This would most likely need to be combined with an error correction coding scheme for the data within the frame. Coding schemes based on either the Hamming distance metric (to handle substitutions) or Levenshtein distance metric [42] (to handle deletions as well) could be used. Another approach would be to modify the framing scheme to reduce the chance of frame corruption. For example, using two frame delimiters at the start of every frame instead of one. This way if only one of the delimiters gets deleted or corrupted, the frame can still be decoded correctly.

6.3.6 Encoding Scheme

Our results for smaller window sizes indicate that for many environments in which the Keyboard JitterBug might be deployed, one could use a more efficient symbol encoding scheme by packing more than one bit of information with each transmitted symbol. To confirm this hypothesis, we implemented a 16 symbol (four bits/symbol) encoding scheme with an additional symbol acting as the frame delimiter. The results of our experiments are summarized in Table 7. The frame size used is 16 bits (four symbols). The Average Correct Frame BER stays at above acceptable levels for all the nodes tested. The results show that it is possible to optimize the framing and encoding schemes to increase the bandwidth of the channel and at the same time maintain acceptable channel performance.

6.4 Summary of the results

Our experimental results indicate that a conservative choice of the window size as 20 ms is small enough to be undetectable by a normal user and at the same time gives good channel performance under a variety of system loads, operating systems and network conditions. One can also increase the bandwidth of the channel by choosing a more aggressive encoding scheme as our results for the high bit rate encoding show. However, our primary goal was to design an encoding scheme that is robust and general enough to work under any unknown environment without affecting user perception. The binary encoding scheme with a timing window of 20 ms serves that purpose quite well.

6.5 Detection

The detection of covert network timing channels is a separate research problem of its own and as such, quite difficult. Thus we do not focus on the detectability aspects of the channel in this paper. However, we briefly analyze some of the issues.

It has been suggested in previous studies that covert network timing channels can be detected by looking at the inter-arrival times of network packets [11, 7]. These detection algorithms rely on the notion of *regularity*, a channel-specific property that can be used to distinguish normal traffic from certain kinds of covert channel traffic. None of these techniques work for detecting the presence of *any* covert timing channel. The Keyboard JitterBug is a low-bandwidth timing channel and has a different form of regularity. Hence, these techniques are unlikely to be able to detect the exploitation of our timing channel.

However, it might be possible to detect Keyboard JitterBug activity by directly observing the inter-arrival times of network packets. The inter-arrival times tend to cluster around multiples of the window size or half the window size. This is because the symbol encoding scheme relies on using an inter-arrival time of 0 (modulo w) for sending a '0' and $w/2$ (modulo w) for sending a '1'. We collected an SSH trace without the use of a Keyboard JitterBug. We then modified the trace by adding simulated jitter so that packet timings corresponded to the case when a Keyboard JitterBug is being used. Because we do not model the effect of noise added by network jitter, this gives us a worst case analysis of the detectability of our channel.

Figure 6 shows the inter-arrival times for 550 packets in the original trace for a range between 0.2s and 0.3s. In Figure 7, we show the same trace except now with simulated jitter that would be added by a Keyboard JitterBug. Notice the banding around multiples of 10 ms, which corresponds to a window size of 20 ms. Thus, a simple plot of the inter-arrival times reveals that that a covert timing channel is being exploited.

To evade such a simple detection scheme, an approach based on rotating the timing window used for symbol encoding is described below. Note, however, that we do

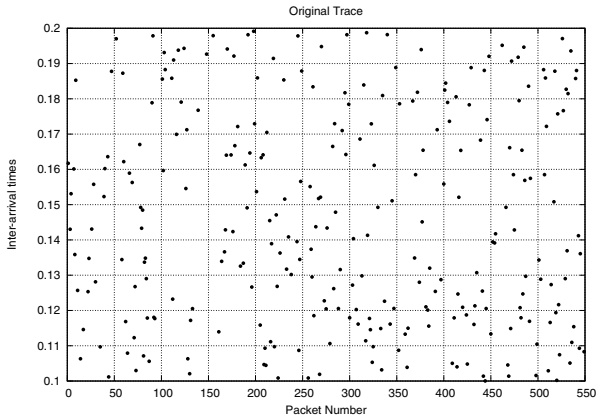

Figure 6: Original SSH Trace

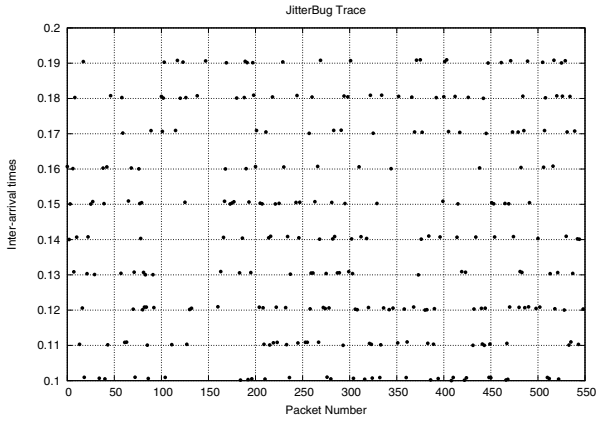

Figure 7: JitterBug applied to the original SSH Trace (stationary time windows)

not claim that the use of the following technique makes our channel undetectable using any other technique. It is simply a countermeasure against the most direct way of detecting our covert timing channel. The timing channel might still be susceptible to other forms of analysis that detect its presence in network traffic.

The method works as follows. As before, let us denote by $\{b_i\}$ the binary sequence to be transmitted using jitters, and by $\{t_i\}$ the sequence of the times when the keys are pressed. Assume there exists $\{s_i\}$, a pseudo-random sequence of integers that range from 0 to $w-1$, where w is, as before, the length of the timing window. The sequence $\{s_i\}$ is assumed to be known by the sender and the receiver but not by anyone else, and works as a shared secret. Rather than encoding bits by adding delays so that the inter-arrival distances cluster around 0 and its antipode, the source adds jitter such that they cluster around the sequence $\{s_i\}$ and its associated antipodal sequence.

More precisely, in order to transmit the bit b_i, the JitterBug adds a delay such that:

$$(\delta_i - s_i) \bmod w = \begin{cases} 0 & \text{if } b_i = 0; \\ \lfloor w/2 \rfloor & \text{if } b_i = 1; \end{cases}$$

where $\delta_i = t_i^{'} - t_{i-1}^{'}$, as before are the difference in times when adjacent keystrokes are sent to the keyboard controller by the Keyboard JitterBug.

Consider an example where Bob wants to send 3-bits of information $\{1, 0, 1\}$ to Eve using JitterBug. Assume that the window size is 20 ms, and that they agreed on the sequence $\{s_0, s_1, s_2\} = \{3, 9, 5\}$. Figure 9 illustrates how the timing window is rotated at each step before deciding on the amount of jitter to add.

Figure 8 shows the inter-arrival times for the same SSH trace with packet timing adjusted for JitterBug but this time using rotating windows during symbol encoding instead of the original static scheme. The sequence $\{s_i\}$ is chosen to be a pseudo-random sequence of integers between 0 and 19. The inter-arrival times are no longer clustered now and there are no new noticeable patterns compared to the original SSH trace.

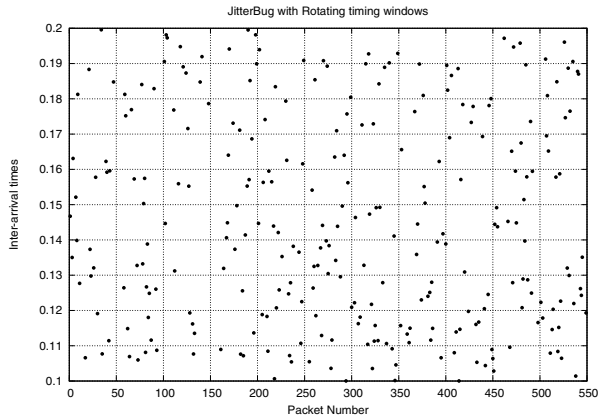

Figure 8: JitterBug applied to the original SSH Trace (rotating time windows)

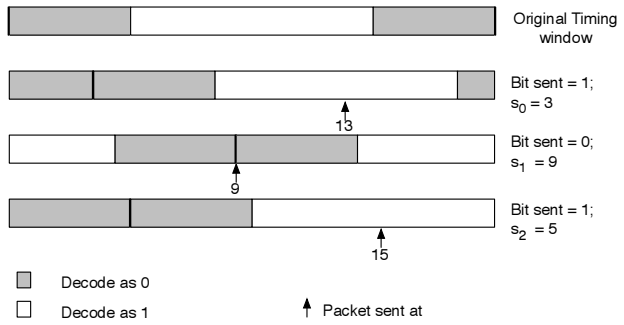

Figure 9: Rotating timing windows: The symbol encoding window is rotated for sending each bit

The intuition behind this approach is that the resulting sequence $\{\widehat{\delta_i}\}$ on the receiver's side looks as arbitrary as $\{s_i\}$. The choice of $\{s_i\}$ is obviously important and should be sufficiently random . Note that when $\{s_i = 0 \; ; \forall \; i\}$, this reduces to the original case with a stationary time window.

7 Conclusions and Future Work

Compromising an input channel is useful not only for learning secrets, but, as we have seen, is also often sufficient for leaking them over the network. We introduced *loosely-coupled network timing channels* and *JitterBugs*, through which covert network timing channels can be exploited to leak sensitive information in general-purpose computing systems. We described the *Keyboard JitterBug*, our implementation of such a network timing channel. The Keyboard JitterBug is a keylogger that does not require physical retrieval to exfiltrate its captured data. It can leak previously captured sensitive information such as user passphrases over interactive network applications by adding small and unnoticeable delays to user keypresses. It is even possible to use the Keyboard JitterBug, at low-bandwidth with other, non-interactive, network applications, such as web browsers and instant messaging systems.

Our experiments suggest that the distance over the network between the receiver and the JitterBug doesn't matter very much. The timing window size w is the basic parameter of the symbol encoding scheme. Its choice is dictated by the expected amount of jitter in the network and by the maximum delay that can be tolerated. A conservative choice of the window size as 20 ms is small enough to be unnoticeable to a human user and at the same time gives good channel performance over a wide range of network conditions and operating systems tested. This makes a Keyboard JitterBug very robust and less susceptible to major changes in the environment in which it is installed. We also described experimental results with some simple framing schemes and more aggressive encoding mechanisms. Our results show that the symbol encoding and framing could be further optimized for better performance in certain environments. Finally, we showed simple techniques for defeating the most direct ways of detecting our attacks.

The most obvious extension to this work is the development of better framing and encoding schemes with higher bandwidth, by making less conservative assumptions that take advantage of specific channel properties. In this paper, however, we deliberately avoided optimizing for any particular channel, operating system, or networked application, instead identifying parameters that give satisfactory performance and that remain highly robust under varied conditions.

All covert timing channels represent an arms race between those who exploit such channels and those who want to detect their use. This necessitates the use of countermeasures by a covert channel to elude detection by network wardens. We suggested only very simple countermeasures in this paper. Our initial results with rotating encoding timing windows indicate that the use of cryptographic techniques to hide the use of encoded jitter channels may be a promising approach. We plan to explore this direction in the future.

Acknowledgments

This research was supported in part by grants from NSF Cybertrust (CNS-05-24047) and NSF SGER (CNS-05-04159). Jutta Degener suggested the name "JitterBug". The idea of using a PIC chip to add jitters emerged from discussions with John Ionannidis. We thank Madhukar Anand, Sandy Clark, Eric Cronin, Chris Marget and Micah Sherr for the many helpful discussions during the course of this research. We are grateful for the facilities of PlanetLab to perform our experiments. Finally, we thank the anonymous reviewers and David Wagner for many helpful suggestions and comments.

References

[1] The *stress* project. http://weather.ou.edu/ apw/projects/stress/.

[2] Trusted computer system evaluation. Tech. Rep. DOD 5200.28-STD, U.S. Department of Defense, 1985.

[3] United States v. Scarfo, Criminal No. 00-404 (D.N.J.), 2001.

[4] ACHARYA, A., AND SALZ, J. A Study of Internet Round-Trip Delay. Tech. Rep. CS-TR-3736, University of Maryland, 1996.

[5] AGAT, J. Transforming out timing leaks. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2000), ACM Press, pp. 40–53.

[6] ANANTHARAM, V., AND VERDU, S. Bits Through Queues. In *IEEE Transactions On Information Theory* (1996), vol. 42.

[7] BERK, V., GIANI, A., AND CYBENKO, G. Detection of Covert Channel Encoding in Network Packet Delays. Tech. rep., Darthmouth College, 2005.

[8] BROIDO, A., HYUN, Y., AND KC CLAFFY. Spectroscopy of traceroute delays. In *Passive and active measurement workshop* (2005).

[9] BROIDO, A., KING, R., NEMETH, E., AND KC CLAFFY. Radon spectroscopy of inter-packet delay. In *IEEE high-speed networking workshop* (2003).

[10] BRUMLEY, D., AND BONEH, D. Remote Timing Attacks are Practical. In *Proceedings of the 12th USENIX Security Symposium* (August 2003).

[11] CABUK, S., BRODLEY, C. E., AND SHIELDS, C. IP covert timing channels: design and detection. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security* (New York, NY, USA, 2004), ACM Press, pp. 178–187.

[12] CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M., AND BOWMAN, M. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev. 33*, 3 (2003), 3–12.

[13] DAEMON9. Project Loki. *Phrack Magazine 7*, 49 (August 1996).

[14] DAVEY, M. C., AND MACKAY, D. J. Reliable communication over channels with insertions, deletions, and substitutions. *IEEE Transactions on Information Theory 47* (2001).

[15] F. F. SELLERS, J. Bit loss and gain correction code. In *IEEE Transactions on Information Theory* (1962), vol. 8, pp. 35–38.

[16] GILES, J., AND HAJEK, B. An Information-Theoretic and Game-Theoretic Study of Timing Channels. In *IEEE Transactions on Information Theory* (2002), vol. 48.

[17] HELOUET, L., JARD, C., AND ZEITOUN, M. Covert channels detection in protocols using scenarios. In *Proceedings of SPV '2003, Workshop on Security Protocols Verification* (2003). Satellite of CONCUR'03. Available at `http://www.loria.fr/~rusi/spv.pdf`.

[18] HU, W.-M. Reducing Timing Channels with Fuzzy Time. In *IEEE Symposium on Security and Privacy* (1991).

[19] JACOBSON, V., BRADEN, R., AND BORMAN, D. RFC 1323 - TCP Extensions for High Performance.

[20] KANG, M. H., AND MOSKOWITZ, I. S. A Data Pump for Communication. Tech. rep., Naval Research Laboratory, 1995.

[21] KANG, M. H., MOSKOWITZ, I. S., AND LEE, D. C. A Network Version of the Pump. In *IEEE Symposium on Security and Privacy* (1995).

[22] KANG, M. H., MOSKOWITZ, I. S., MONTROSE, B. E., AND PARSONESE, J. J. A Case Study Of Two NRL Pump Prototypes. In *ACSAC '96: Proceedings of the 12th Annual Computer Security Applications Conference* (Washington, DC, USA, 1996), IEEE Computer Society, p. 32.

[23] KELSEY, J., SCHNEIER, B., WAGNER, D., AND HALL, C. Side Channel Cryptanalysis of Product Ciphers. In *ESORICS '98* (1998).

[24] KEMMERER, R. A. A Practical Approach to Identifying Storage and Timing Channels: Twenty Years Later. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference* (Washington, DC, USA, 2002), IEEE Computer Society, p. 109.

[25] KOCHER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996), pp. 104–113.

[26] KOHNO, T., BROIDO, A., AND KC CLAFFY. Remote Physical Device Fingerprinting. In *IEEE Symposium on Security and Privacy* (2005).

[27] LAMPSON, B. W. A Note on the Confinement Problem. In *Communications of the ACM* (1973), vol. 16.

[28] LEE, P. *Combined error-correcting/modulation recording codes*. PhD thesis, Univesity of California, San Diego, 1988.

[29] LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet Physics Doklady* (1966), vol. 10, pp. 707–710.

[30] LEVINE, B., REITER, M., WANG, C., AND WRIGHT, M. Timing Attacks in Low-Latency Mix Systems. In *Proceedings of Financial Cryptography: 8th International Conference (FC 2004): LNCS-3110* (2004).

[31] MILLEN, J. 20 years of covert channel modeling and analysis. In *IEEE Symposium on Security and Privacy* (1999).

[32] MILLER, R. B. Response time in man-computer conversational transactions. In *AFIPS Fall Joint Computer Conference* (1968), vol. 33.

[33] MOSKOWITZ, I. S., AND KANG, M. H. Covert Channels – Here to Stay ? In *COMPASS* (1994).

[34] MOSKOWITZ, I. S., AND MILLER, A. R. The Influence of Delay Upon an Idealized Channel's Bandwidth. In *SP '92: Proceedings of the 1992 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1992), IEEE Computer Society, p. 62.

[35] MOSKOWITZ, I. S., AND MILLER, A. R. Simple timing channels. In *IEEE Symposium on Security and Privacy* (1994).

[36] MURDOCH, S., AND DANEZIS, G. Low-cost traffic analysis of tor. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy* (2005).

[37] NAGLE, J. RFC 896 - Congestion Control in IP/TCP Internetworks.

[38] PROCTOR, N. E., AND NEUMANN, P. G. Architectural Implications of Covert Channels. In *15th National Computer Security Conference* (1992).

[39] RATZER, E. A., AND MACKAY, D. J. C. Codes for channels with insertions, deletions and substitutions. In *Proceedings of 2nd International Symposium on Turbo Codes and Related Topics, Brest, France, 2000* (2000), pp. 149–156.

[40] SHANNON, C. E. A mathematical theory of communication. *Bell System Technical Journal* (1948), 379–423 and 623–656.

[41] SONG, D. X., WAGNER, D., AND TIAN, X. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium* (2001).

[42] TANAKA, E., AND KASAI, T. Synchronization and substitution error-correcting codes for the Levenshtein metric. In *IEEE Transactions on Information Theory* (March 1976), vol. 22, pp. 156–162.

[43] VENKATRAMAN, B. R., AND NEWMAN-WOLFE, R. Capacity Estimation and Auditability of Network Covert Channels. In *IEEE Symposium on Security and Privacy* (1995).

[44] WANG, X., CHEN, S., AND JAJODIA, S. Tracking anonymous peer-to-peer VoIP calls on the internet. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), ACM Press, pp. 81–91.

[45] WANG, X., AND REEVES, D. Robust Correlation of Encrypted Attack Traffic Through Stepping Stones by Manipulation of Interpacket Delays. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)* (2003).

[46] WRAY, J. C. An Analysis of Covert Timing Channels. In *Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, California* (1991).