ABSTRACT

MITTAL, SWASTIK. T-Pack, Timed Network Security In Real Time System. (Under the direction of Dr. Frank Muller.)

Real-time systems are widely deployed as cyber-physical systems to control physical processes with timing requirements in a networked environment, i.e., as distributed systems. Purpose of realization as a distributed system ranges from providing fault tolerance, remote monitoring, co-ordination with other distributed control processes to data processing of networked sensors and actuators. Exposure to network communication between real-time control systems raises system vulnerability to malware attacks over the network, e.g., man-in-the-middle attacks, data corruption, denial of service and eavesdropping. Such attacks require code injection of malware into the system, which results in alteration not only of system behavior via hijacked execution control but also incurs timing dilation to plant the injected code or, in case of network attacks, to drop, add, reroute, or modify packets before they reach their target.

This work proposes to use the time overhead added by such cyber attacks to detect malware intrusion in cyber-physical real-time systems. A new method of timed packet protection, T-Pack, analyzes end-to-end transmission times of packets and detects a compromised system or network based on deviation of observed time from the expected time. Malware intrusion detection is demonstrated by observing timing constraints with and without a compromised network. First, a custom header is introduced to each packet to store timing information calculated to reflect end-to-end transmission times. Second, real-time application scenarios are analyzed in terms of their susceptibility to malware attacks. Results are evaluated on a distributed system of embedded platforms running a Preempt RT Linux kernel to demonstrate the real-time capability of our work.

T-Pack, Timed Network Security In Real Time System

by
Mittal, Swastik

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2020

APPROVED BY:

Dr. Muhammad Shahzad

Dr. Ruozhou Yu

Dr. Frank Muller
Chair of Advisory Committee

DEDICATION

To my parents.

BIOGRAPHY

The author was born in Dehradun, India. After completing his schooling in 2014 he started working towards his B.Tech. degree in Computer Science and Engineering at Vellore Institute of Technology, Chennai, India. He graduated with an undergraduate degree in 2018, after which he began pursuing his graduate degree in Computer Science at North Carolina State University, Raleigh. Since August 2019 he has been working as a Graduate Research Assistant under the guidance of Dr. Frank Mueller.

ACKNOWLEDGEMENTS

I would like to thank everyone who has helped me over these last 2 years. I would like to thank my advisor Dr. Frank Mueller for giving me the opportunity to work under his guidance. His deep insights, encouragement and advice have been invaluable towards the completion of this work. I would also like to thank Dr. Muhammad Shahzad and Dr. Ruozhou Yu for taking interest in my work and being part of my advisory committee. I would also like to thank the people in my research group with whom I've had truly wonderful discussions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1 INTRODUCTION

Computer security has become a critical requirement for any type of computer application in modern world. Some of the main components of computer security are: (1) System level security, securing end system's code at different levels and layers from attacks leveraging memory (e.g., buffer overflow attacks including stack smashing [One96], heap overflows [Con99]) or non-memory related attacks (e.g., value range overflows [Hor02], shell shock [Mar15], port smashing); and (2) network security, securing systems on the network from being attacked by some malicious users. For the latter, a wide range of attacks are common including relay, replay, phishing, spoofing, man-in-the-middle, denial of service, eavesdropping etc. [Fra11b; Fra11a; Cal09; SHA18]. One of the most common attack forms is the delay attack. In this attack, the objective is to stall execution/packets of a time-sensitive event (e.g., a code section within the system or some client/server request) causing excessive delays and resulting in performance degradation [Lou19]. Real-time systems are particularly susceptible to such attacks as deadlines are critical for correct system behavior, i.e., time dilation not only results in performance penalties or reduced network throughput but may cause a control system to malfunction, which can result in damage to the controlled environment or even loss of life. Past work shows how delay attacks have affected cyber-physical systems (CPS) [Lou] and network control systems [Sar15] subject to real-time constraints.

Significant work has been invested in analyzing and mitigating the impact of these attacks [Lou19; Lou; Sar15]. Real-time systems offer a unique opportunity for intrusion detection besides traditional, general-purpose cyber-security techniques: Their inherent knowledge of execution times, required for real-time scheduling, opens up the path for additional monitoring and protection. The same techniques for establishing timing bounds on the execution of the real-time task may be applied to bound execution of any code section within the application [Gor04; Wil08]. Past works have used this model for security in real time systems from memory attacks [Zim10], securing clock synchronizations from delay attacks caused by malware intrusion on the network [Nar18] and security in smart grid systems [Ped16]. Timed analysis is not just restricted to security, it can also be used to design attack models, e.g., in context of hardware security tokens such as Smartcards [Dhe98; Koc96; Sch00], exported secret keys used for RSA decryption [Koc96] and remote timing attacks [Bru05]. The novelty of our work is to utilize timed-based security to predict delay attacks on a network subject to communication with real-time constraints.

1.1 Motivation: Detecting Network Delay with Minimal Cost for Real-Time System Using Monitoring Techniques

Many real-time systems require communication between various subsystems over a network (mostly a dedicated intranet, unlike network control systems). Communication messages between these subsystems may be health checks, scheduler messages, raw sensor data, actuation commands, transformed signals, etc. For example, consider a simple drone. The flight dynamics of this drone are controlled using different subsystems, such as a braking system to regulate the speed of the drone, a propulsion system to handle signals between controller and sensors of the drone, management systems to schedule task between these subsystems and to also ensure that a task adheres to a given deadlines. Automotive real-time control systems, such as in Tesla architecture, also consist of a large number of independent subsystems communicating over various networks, multiple controller area networks and also Ethernet [Tes]. These subsystems rely on network connections to interact with other nodes to establish full-system operations. As discussed earlier, such a network is vulnerable to attacks. In safety critical real-time systems, slower response or failure could result in significant environmental damage or even in loss of life. System restarts often cannot be instant due to an unstable physical system state. This research focuses on intrusion detection of such attacks at packet level, i.e., before malware enters a given subsystem. The earlier intrusion is detected, the easier it is to resort to a safe operational mode with reduced or even without communication to another subsystem that has been compromised, e.g., using the Simplex design [Cre07; Bak09], thereby avoiding damage to life and property. Notice that our work focuses on intrusion detection and relies on established methods to transition to a safe state, i.e., Simplex and other methods to transition to are beyond the scope of this research.

1.2 Hypothesis

To tackle the above challenges we put forward the following hypothesis:

Monitoring the round trip time of periodic communication at the packet level in a real-time distributed system provides a means to detect network intrusions complementing conventional security methods.

The contributions of our work as as follows:

- A novel method, T-Pack, is designed and implemented that adds custom headers to the packets and enforces end-to-end deadlines on network traffic using timing information stored within the headers.
- End-to-end packet deadlines are exploited to detect malware intrusion within the network.
- Experiments are conducted on real-time applications with attack scenarios to assess the potential and limitations of T-Pack.
- Results indicate that T-Pack induces a modest amount of performance overhead to the real-time system (≈ 0.12 milliseconds). For a real-time based UAV model (UAV Paparazzi [Hat14], T-Pack was able to detect 100% of the delays caused by distributed denial of service attack in the form of "ping-of-death" with 30 attacker processes each on two nodes sending ICMP ping packets of size 1000 bytes at intervals of 0.001 seconds, and around 99% of the delays with 500 byte ICMP packets to the observed server as a ping flood (0 seconds interval) with 10 attackers per node. Attacks with lower intensity, i.e., one attacker sending 10 parallel ICMP ping packets of 500 bytes at 0.5 second intervals, go undetected by T-Pack.
- Similar results were observed for another real-time system in a drone-like multi-system. T-pack detected the first attack all the time (100%) and the second attack 95% of the time. Similar to UAV Paparazzi model, 99% of the attacks for the third attack scenario remained undetected (but one may argue that system functionality also remained intact so that the threat level was low).

The document is organized as follows. Chapter 3 discusses related work on cyber security in general and timing-based cyber analysis techniques in particular to detect malware and contrasts these with the T-Pack method of timed analysis for packet delivery. Chapter 4 provides an overview of the system design elaborating on the methodology of T-Pack for various communication models and attack scenarios. Chapter 5 indicates how the T-Pack design was implemented in the Linux kernel with the Preempt-RT real time patch deployed on Raspberry Pis. Chapter 6 details the experimental setup of the framework and Chapter 7 discusses experimental results and Chapter 8 summarizes the contributions.

CHAPTER

2 BACKGROUND

2.1 Computer Security

Computer security is the process of detecting and preventing intrusion and malware execution in computer systems and networks by maintaining policies to mitigate the effect of unauthorized access. Early signs of importance of computer security were reflected during World Was II. Arthur Scherbius's enigma machine prevented enemies from implementing counter strategies as they could not decode the intercepted messages sent between troops until it was deciphered by a team of British mathematicians. Since the birth of Apple and advent of personal computers, computer security took a giant leap. Many advancement were made in security policies. Today computer security is one of the essential aspects of computer systems and networking.

2.1.1 System Security

System-level security refers to the architecture, policy and processes that ensure data and system security on individual computer systems. It protects a standalone system from intrusions that can compromise standard system functionality. A system is secure when its resources can be used as intended under any circumstances. A violation of this could be due to system intrusion, e.g. by gaining system control via memory-related attacks or non memory-related attacks, as detailed in the next section.

2.1.2 Network Security

A compromised network could be fatal in today's world given that many devices are connected via the Internet. Hence, network security is essential to protect devices from malicious users who might disrupt communication between end users. Such disruptions or attacks could be as basic as increasing the traffic on the network with unwanted packets to reduce full capacity or interfering between the communication of end users listening to the messages and utilizing it to gain trust and access systems that are part of the network.

2.2 Attacks

2.2.1 System Attack

Processes or programs may be used by malicious users to gain unauthorized access to end systems effecting the intended functionality of the system. These attacks could exploit the memory or other processes in the system to attack. A common memory exploitation attack is the buffer overflow attack. Attackers exploit buffer overflow issues in user programs by overwriting the memory of an application. This changes the execution path of the program, triggering a response that damages files or exposes private information. Some of the common buffer overflow attacks include stack smashing, a vulnerability where the stack of a computer application or Operating System (OS) is forced to overflow. This occurs when a buffer overflow overwrites data in the memory allocated to drive execution , e.g. the return address of a function call. Another form of buffer overflow attacks includes heap overflows. Dynamically allocated heap can be exploited to overwrite internal structures such as linked list pointers and program data. Non-memory related attacks disrupt the system performance without exploiting the memory of the applications. Some of these attacks are shell shock and port smashing. Shell shock is a security bug in the Unix bash shell which gives attacker the ability to cause bash to execute arbitrary commands and gain unauthorized access. Port smashing is known to exploit Simultaneous Multi Threading (SMT), which enables running two program simultaneously on single CPU core (hyper threading). The attacker thread is used to monitor a worker thread indirectly to reveal details about the simultaneously running application. For example, port smashing uses an attack thread to continuously utilize a common port with instructions unless the CPU's scheduler stops running and hands the port over to the other thread. By measuring the time in between its own instructions running on that port, it can measure the time that the other thread takes to process its own instructions.

2.2.2 Network Attack

As mentioned in the context of network security, shared resource applications are increasing by requiring various homogeneous or heterogeneous systems to be connected over a network. These networks could easily be exploited to affect the intended functionality. There are a vast variety of network attacks including relay, replay, phishing, spoofing, man-in-the-middle, denial of service etc.

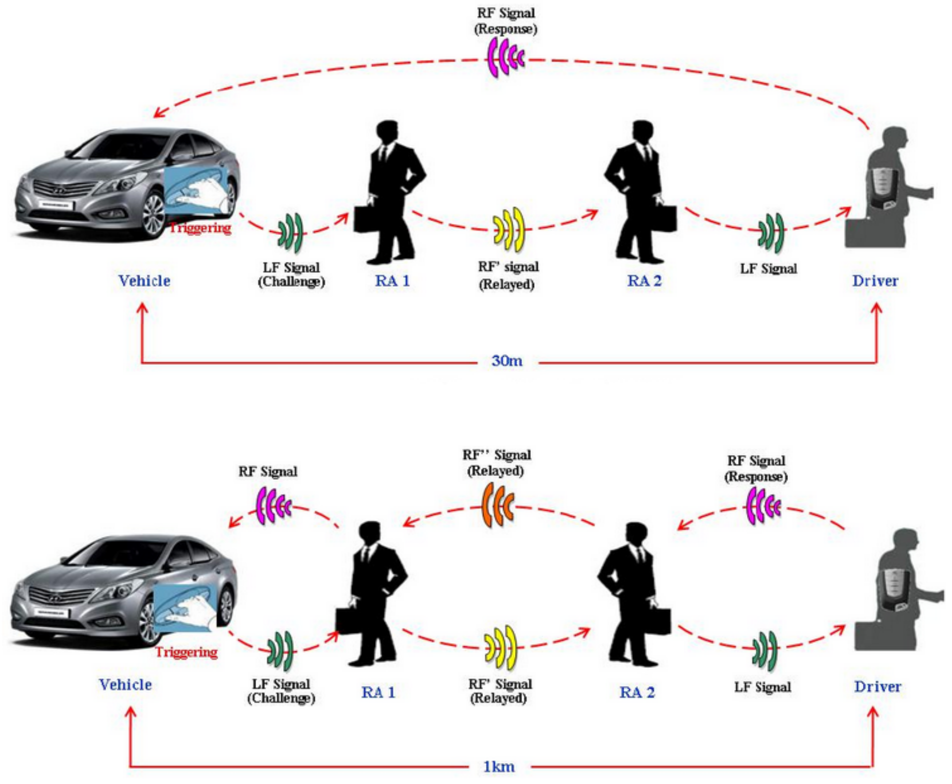


Figure 2.1 Relay Attack[Kim13]

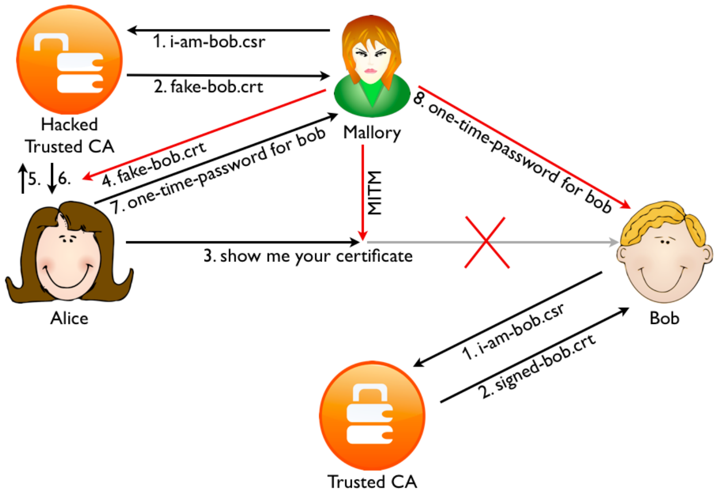


Figure 2.2 Replay Attack

image credit: Mount Knowledge, Strong Authentication For 2012, Richard (January 14, 2012)

Relay and replay attacks are common in real-time system. Both can be classified as subcategories of man-in-the-middle attacks. In a relay attack, as seen in Figure 2.1, RA1 approaches the car with a key that simulates the same behaviour as the car key. It pings the car locking system impersonating the key, and the system responds by sending a signal, which the key is meant to pick up to respond back to. RA1 forwards the signal to another attacker, RA2, near the actual key, which in turn relays it to the actual key receiving the correct reply and unlocking the car. Replay attacks also are a form of man-in-the-middle attacks, where the hijacker (or man in the middle) on the network intercepts the signal from a sender or client with the client certificate and impersonates itself as the same sender to send this signal to the receiver, which in turn believes that the signal arrived from a credible source. Man-in-the-middle attacks in general could do much more than just relay or replay, intercepted signals could be decrypted and modified affecting the intended functionality of the probable distributed system. Denial of service is a very common form of attack, where network capability and capacity could be compromised to deny services to other systems in the network. In this research, ping-of-death is used, a form of denial of service attack that just floods the servers in client-server models with high frequency ping traffic causing server network buffer overflow, which increases the receiving and transmission time and thus affects the network performance. Denial of service attacks are common as they are not as hard to execute requiring the attacker to just be a part of the network. DOS attacks effected scheduled flights ar the Warsaw airport in 2015 leading to cancellations and delays as air traffic control could not retrieve flight plans from the servers.

2.3 Real-Time Systems

Systems with a predictable execution times adhering to deadlines of tasks are called real-time systems. In such a real-time system deadline miss could lead to fatal loss of life or environmental damage. For example, autonomous vehicles, power grid control systems etc. A failure to apply breaks within a given time could lead to accidents in autonomous driving. These systems are build considering predictability and performance. Predictability, unlike in commodity computing cannot be compromised for performance. Fig 2.3 and Fig 2.4 show cyclic execution results for four CPU cores of a Linux System (Ubuntu 16.04) vs a Preempt-RT Linux running on a Raspberry Pi. Cyclic execution measures the amount of time that passes between, when a timer expires and when the thread, which set the timer, actually runs. It is accomplished by taking a time snapshot just prior to waiting for a specific time interval (t1), then taking another time snapshot after the timer finishes (t2), then comparing the theoretical wake up time with the actual wake up time (t2 -(t1 + sleep_time)). This value is the latency for that timer wake up. Results show a smaller average latency for Linux compared to Preempt-RT. However, there are also a few outliers with non-deterministic high latency values for some of the thread task (From Fig. 2.3), which are not observed for Preempt-RT (Fig. 2.4). Maximum recorded latency for Linux was 1828us compared to 102us of Preempt-RT Linux. This experimental result demonstrates the requirement of predictability, even if it leads to decrease in average performance.

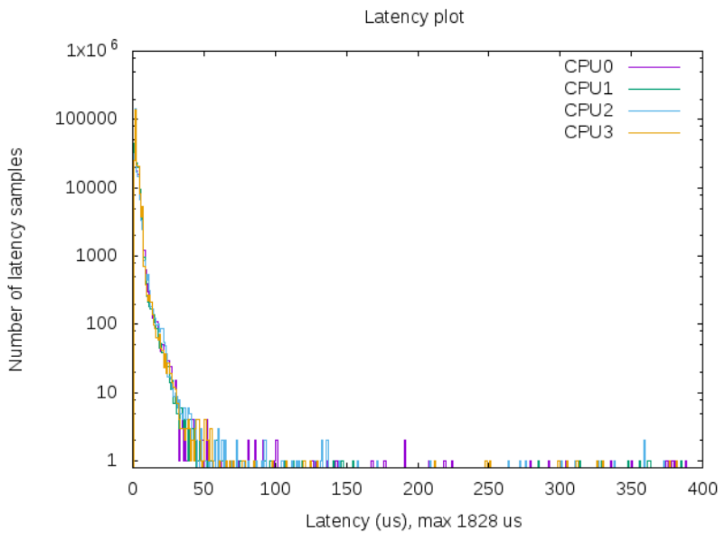


Figure 2.3 Cyclic test for Linux OS

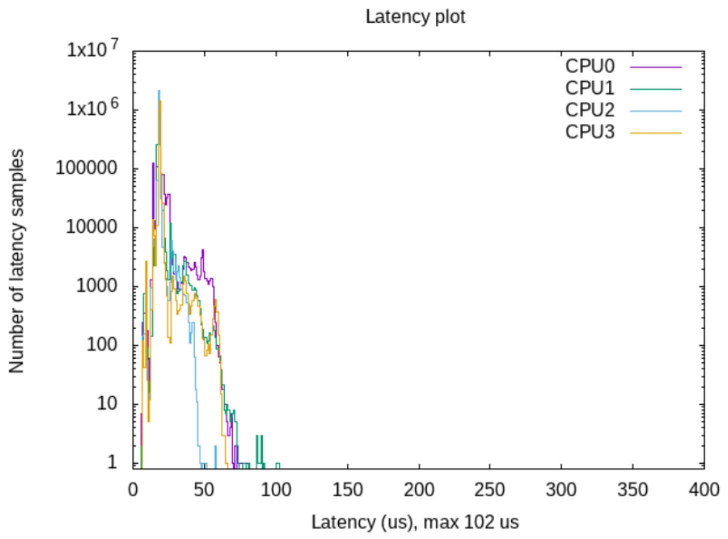


Figure 2.4 Cyclic test for Preempt RT Linux

2.3.1 Preempt RT Linux

A popular open source UNIX-like operating system, Linux, a widely used operating system in personal computers. Popularity of Linux and its free license led to its use in many embedded systems, i.e, devices like routers, automation controls, smartwatches etc.

Linux systems are not fully preemptible - a task may issue a kernel request via syscalls during which they cannot be preempted by other tasks. The Linux Preempt-RT patch provides full preemptible functionality to Linux to reduce the length of delays that can be encountered during syscalls, thereby providing real-time capabilities. The patch achieves this by: 1) Making in-kernel locking-primitives (using spinlocks) preemptible though re implementation with rtmutexes; 2) Critical sections protected by the spinlock_t and rwlock_t now become preemptible. The creation of non-preemptible sections (in kernel) is still possible with raw_spinlock_t (same APIs as spinlock_t); 3) Converting interrupt handlers into preemptible kernel threads: The RT-Preempt patch treats soft interrupt handlers in a kernel thread context, which is represented by a task_struct like a common user space process. However, it is also possible to register an IRQ in kernel context; 4) Converting the old Linux timer API into separate infrastructures for high resolution kernel timers plus one for timeouts, leading to user-space POSIX timers with high resolution.

Well documented, flexible and easy to use network API's of Linux with real-time capabilities of Linux Preempt RT make it the most suitable operating system for implementation and testing of T-Pack.

2.4 Timed Security in Real-Time Systems

Our day to day modern life is surrounded by devices with real-time system, e.g. Cyber Physical System (CPS) devices (E.g. smart grid, autonomous automobile systems, medical monitoring, industrial control systems, robotics systems, and automatic pilot avionics) - and the Internet of Thing (IoT) devices (E.g. thermostat, home-security systems and smartphones etc). Attacks discussed above pose a severe risk to such devices. Execution of malware within these systems may result in time lag of control actions. These devices are also inherently networked posing a high security risk if the network is compromised- DoS attacks discussed above could hinder the flow of incoming packets leading to missed deadlines because of network delays.

Such devices utilized to control systems with soft or even hard real-time constraints. The execution path of such control code on embedded devices thus follows a stringent and predictable behavior, which can be characterized by timing analysis. Once upper bounds on timings along execution paths are established, this information not only aids in the verification of timing constraints, but it can also be exploited to detect deviations from the certified timing behavior. Timing-based malware detection or Timed Security thus provides a means for non-stop system integrity. What is more, it can be used to trigger transitions into a safe operating mode at an early intrusion detection point to prevent anomalous behavior from escalating.

CHAPTER

3 RELATED WORK

Deterministic execution time is a necessity in real-time systems, be it a real-time task executing within the system or for packet transmission over a network. Analysis of timing bounds for execution of a task or section of code as well as packet delivery time are required for validating deadlines, but they can also disclose vital information about the system. Once timing bounds on real-time tasks have been established, such knowledge can be exploited for cyber attacks. Timing constraints reveal the extent to which some external medium may influence system behavior and, what's more, indicate to what extent time divergence from expected behavior can be tolerated without violating deadlines. Hence, cyber attacks can be constructed to intentionally violate deadlines or to intrude the system without deadline violation and without being detected. Prior work exploited timing bounds derived from timing analysis of code to detect malware intrusion. In particular, Zimmer et al. [Zim10] developed techniques to provide micro-timings for multiple granularity levels of application code. They implemented a timed analysis method, T-Rex, which incorporates application level checks to detect buffer overflow attacks by monitoring the elapsed time of return from a function call. They further developed T-Prot, which utilizes synchronous system calls at security checkpoints and validates WCET bounds of longer code sections, and T-Axt, which checks timing bounds asynchronously within the real-time scheduler. Their work demonstrated an advantage of these timed analysis of code execution in constraining the window of vulnerability of code injections, which usually take tens of millions of cycles, down to tens, hundreds, or thousands of cycles, depending on the respective protection technique. In contrast, our work focuses on network protection.

Cyber-physical control systems are generally subject to real-time constraints. These systems are vulnerable to malware intrusion over the network. Prior work [Che11; Kos10; Che18] demonstrated the viability of attacks on the network of a real-time systems and uncovered potential damages. Our work proposes to mitigate damages by detecting intrusion prior to such attacks using timed analysis of packets on the network. Timing packets is not a new concept, one application of it is embedded within the widely used TCP protocol, namely to predict the loss of packets on the network if a timeout (threshold time) has been exceeded for re-transmission in an effort to adhere to reliable data transfer. Earlier TCP methods, e.g., TCP-Reno and TCP-Tahoe, maintained a weighted moving average of estimated round trip times (RTT) as a threshold, which is calculated by measuring the RTT of a packet (data packet send and acknowledgment received).

In our work, this technique is utilized for establishing end-to-end packet delivery times for intrusion detection in real-time system. Our technique can be deployed on a commodity network, however, with limitations due to highly dynamic and unpredictable data rates on the network would present a challenge in establishing timing bounds on the worst-case packet latency and worst-case execution time of kernel code within the network stack. The threshold bound in question is the worst-case RTT instead of common estimated RTT that serve as indicators for re-transmission within the TCP proposal. In a real-time system with a given set of messages between subsystems, the size of each message and the time of when data is transmitted are known a priori. This makes it feasible to exploit more rigid timing bounds in dedicated control networks, i.e., without commodity background traffic, to establish bounds on the worst-case time of packets and, hence, further utilize this information for malware intrusion detection.

The closest work to our T-Pack method measures RTT by timing signal features at the sender and receiver to secure clock synchronization and to subsequently detect intrusion attacks on the network [Nar18]. Their detection of intrusion resides on observing a delay attack using timed sender/receiver analysis under certain conditions: If the RTT of the signal between two nodes exceeds a threshold, then the signal is predicted to be delayed by some interception in the middle. We use a similar attack model and analysis method to predict performance and accuracy of T-Pack, but instead of server/receiver analysis, we embed timing information within packets, which is subject to assumptions detailed in our attack model. T-Pack also helps in detecting network intrusion more accurately compared to [Nar18] as it eliminates delay due to internal errors within the system (upper layers of the network stack) by measuring time information at lower layers of the network stack. This has also been demonstrated through experiment 1 in Chapter. 7 of the this thesis.

CHAPTER

4 DESIGN

Table 4.1 ABBREVIATIONS

S	Sender Node
R	Receiver Node
t_S	Time at which packet is sent from S
t_R	Time at which packet is received at R
ACK	Acknowledgment packet from R to S in 2-way message
t_{AR}	Time at which ACK is sent from R
t_{AS}	Time at which ACK is received at S
RTT	Round trip time (TCP)
ETT	End-to-end time (UDP)
Δt_{rs}	Constant clock difference between S and R
T_{obs}	Observed end-to-end time
T_{exp}	Expected end-to-end time
T_d	Mean random delay on the uncompromised network
ΔT_d	Deviation of delay from mean
T_c	Added delay due to compromised network
ΔT_c	Deviation of added delay from the mean added delay
T_{WCET}	Expected worst-case end-to-end time of packet

T-Pack is a novel methodology to verify end-to-end timing of each packet on the network of a real-time system during message transfer between subsystems. Figure 4.1 depicts a high-level timing model for message transfer between two subsystems for unidirectional UDP (left) and bidirectional TCP transfers (right) using the notation established by Table 4.1. A message from sender S to receiver R is analyzed at packet level, considering packet P being sent at time t_S from S to R, where it is received at time t_R. The observed end-to-end time, T_{obs}, is compared with the expected time, T_{exp}, to detect malware intrusion in the network. The work assumes loosely synchronized clocks with a constant time difference, Δt_{rs}, between any two subsystems, which may be dynamically updated due to clock drift.

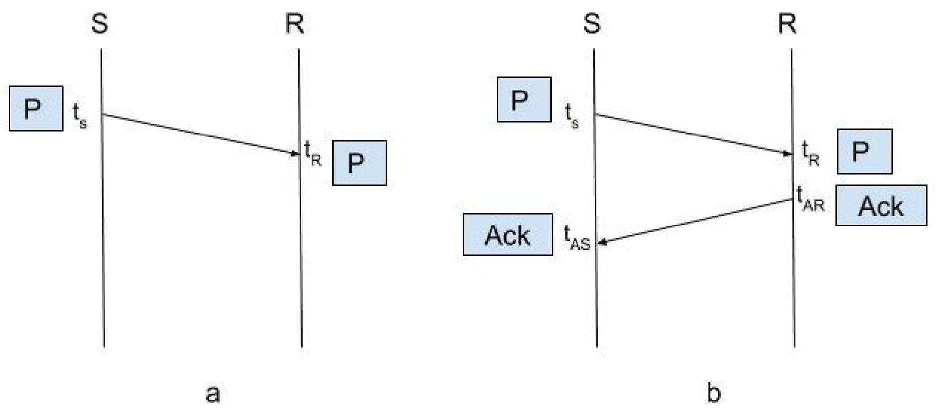


Figure 4.1 a. One-Way (UDP) b. Two-Way (TCP)

4.1 One Way Message Transfer

Real-time systems with a reliable data transfer protocol exposed to the application level rely on known network bandwidth and latency bounds. The UDP transport protocol is a suitable protocol assuming point-to-point full-duplex switch connectivity between network endpoints. Under UDP, a message is transferred from S to R (Fig. 4.1a) without any acknowledgment from R. In this scenario the observed end-to-end time is the time it takes for the packet to reach R having been sent by S.

$$ETT = t_r - t_s + \Delta t_{rs} = T_{exp} \tag{4.1}$$

ETT is the ideal end-to-end time. Considering internal delays, we get

$$T_{exp} = ETT + T_d + \Delta T_d. \tag{4.2}$$

Note: We consider end-to-end time, which in measurements includes queuing and transmission delay within the subsystem as well, this delay is denoted by $T_d + \Delta T_d$, which will be included in $t_r - t_s$ as shown in Table 4.1.

4.2 Two-Way Message Transfer

TCP is a two-way message transfer transport protocol based on a handshaking protocol that acknowledges packets sent from the sender, S, to achieve reliable communication over the network (Figure 4.1b). In this scenario, a transfer is said to be complete once the acknowledgment (ACK packet) is received at S, again under point-to-point full-duplex switch connectivity. Here, we assume a constant clock difference between sender and receiver for the duration of packet communication. Clock drift requires this difference to be updated from time to time, which is typical for distributed systems and beyond the scope of the paper [Mil91].

$$t_r = t_s - \Delta t_{rs} + ETT \tag{4.3}$$

$$t_{AS} = t_{AR} + \Delta t_{rs} + ETT \tag{4.4}$$

Adding Eq 4.3, Eq 4.4 and simplifying yields

$$RTT = (t_{AS} - t_{AR}) + (t_r - t_s) \tag{4.5}$$

RTT is the ideal end-to-end time. Considering with internal delays, we get

$$T_{exp} = RTT + T_d + \Delta T_d. \tag{4.6}$$

When packets are small, which is often the case in real-time systems as singular or small sets of sensor values are transmitted, TCP combines multiple small sized packets into one to reduce the headers overhead on the network [Nag]. TCP also implements delayed acknowledgments on the receiver side to send a cumulative acknowledgment for multiple packets, thereby reducing network congestion. These implementation details of TCP create non-deterministic execution behavior, which can be fatal for real-time systems. On the sender side, combining multiple packets delays sending earlier sensory data until a cumulative packet reaches the critical threshold of flushing it to the network. On the receiver side, a delay in sending ACK messages can be incurred as the receiver accumulates packets for some time up to a threshold before cumulatively acknowledging packet receipts. This would adversely affect real-time systems because of the non-deterministic behavior of packet delivery combined with delayed reception of time-critical sensor data, which may cause sensor updates to arrive past an actuation deadline. Work in [Buf] shows how packet buffering and delayed ACKs could affect the performance of client-server applications, which matches the communication pattern of distributed real-time systems. Given the small size of the packets, additional data due to headers hardly affect average network performance as opposed to buffering and the resulting delayed ACKs. Cumulative packets from sender and delayed acknowledgement from receiver would further violate our model of packet-level timing analysis from Fig. 4.1. To counter this problem, we require socket options of “TCP NODELAY” to be used, which ensures that packets are immediately forwarded to next software layer without buffering, irrespective of their message payload size, and ultimately put on the wire. We also require "TCP QUICKACK" socket option at the receiver side to avoid delayed, cumulative acknowledgments under TCP for multiple packets in favor of singular packet acknowledgments.

To further demonstrate our claim of "TCP NODELAY" and "TCP QUICKACK" having only minor effects on network performance, we measured the network utilization on one of subsystem by executing an experimental model (Paparazzi UAV) as described in Chapter 6. We monitor the average bandwidth utilization along with the number of packets flowing in and out of the interface for a period of time with and without TCP modifications discussed above. For a period of 60 seconds in the middle of execution of the Paparazzi UAV model, the average rate of data observed at the receiving (rx) and the transmitting (tx) end are 353.50 Kbps and 779.67 Kbps, respectively, without TCP modifications (i.e. with "TCP NODELAY" and "TCP QUICKACK"). This decreases to 347.98 Kbps, 753.80 Kbps, respectively, with TCP modifications. Considering the theoretical bandwidth of 100 Mbps of the Raspberry Pi, this represents an overhead of 0.354% (rx), resp. 0.780% (tx), in utilization without TCP modifications and 0.348% (rx), resp. 0.754% (tx), with them. Hence, using "TCP NODELAY" and "TCP QUICKACK" results in a minute increase in the network utilization, which is insignificant relative to total available bandwidth. As the experiment features a time-triggered real-time system, it features a consistent flow of packets on the network. This implies that duration and time of measurements in the above experiment do not significantly affect the result. During the above time duration, in and out flows were 473 packet/sec (rx), resp. 713 packets/sec (tx), with TCP modifications and 486 packets/sec (rx), resp. 742 packets/se (tx), without it. The average number of packets increase with "TCP NODELAY" and "TCP QUICKACK", but by a very small margin.

4.3 Attack Model

Real-time systems require deterministic timing. Each request made by the subsystem over the network should be completed within a given deadline; otherwise, the system is considered incorrect with potentially severe consequences as discussed. Attacks on the network of a real-time systems may not necessarily be tampering with the data of a packets, they could also be causing a delay to inflict deadline misses, thereby disrupting not only packet flow but also the primary functionality of the system. Our T-Pack model vision is to detect such attacks on the network and to convey to any real-time subsystem that adverse actions may be required to prevent any damage.

We implement checksum techniques within T-Pack to detect corruption of timestamp information that would affect correctness of timings under T-Pack.

As discussed, a prior clock synchronization protocol [Nar18] studied delay attacks in experiments. We aim to detect such attacks using our T-Pack model. We consider a compromised real-time system, where one of the subsystems or the switch behaves as the malicious node delaying any packet that is forwarded through it to other subsystems (Fig. 4.2). For our two-way message transfer, we have:

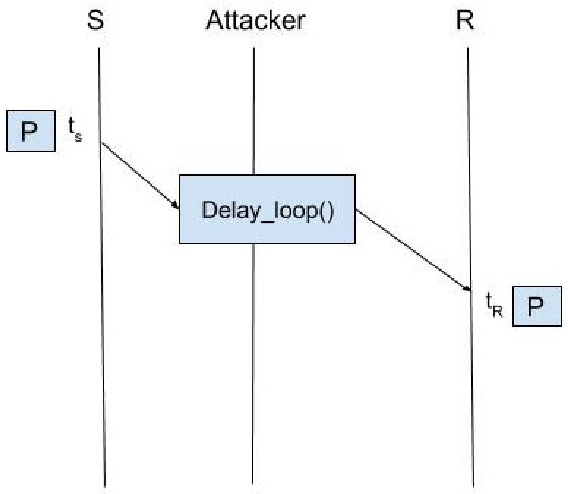


Figure 4.2 Message Transfer In Compromised Real-Time System

Using T_{exp} From Eq 4.6

$$T_{obs} = T_{exp} + T_c + \Delta T_c. \tag{4.7}$$

Let us stress again that T-Pack's objective is not to prevent intrusion but rather to detect intrusion when some subsystem inflicts incorrect timing on network behavior. T-Pack's objective in this case is to prevent other subsystems from becoming compromised as well — by detecting intrusion and then transitioning into a safe mode on the uncompromised systems, e.g., via Simplex [Cre07; Bak09] or other mode transitions depending on the application scenario, which is beyond the scope of the paper.

4.4 Vulnerability Of T-Pack

We can assume that an attacker may not cause packets to be modified without detection since all packets are encrypted, i.e., the attacker would not be able to update packet integrity (e.g., checksum) within the encrypted message without the knowledge of the private key, which is known to the uncompromised subsystems, i.e., the receiver. Similarly, any timestamp values added to a packet by our T-Pack protocol cannot be corrupted without receiver detection.

We determine the WCET bound for a packet to be complete if its end-to-end transfer based on our model is as follows. In case of a two-way transfer, this includes T_{exp} with an inherent random delay of $T_d + \Delta T_d$ (Eq. 4.6). Hence, the WCET is bounded by

$$T_{WCET} = RTT + T_d + |\Delta T_d|, \tag{4.8}$$

where $|\Delta T_d|$ signifies the maximum positive deviation for the WCET. So, for a compromised network, we obtain T_{obs} from Eq 4.7.

For some values of $T_c + \Delta T_c$, where the attacker delays the packet transfer by a small value, we may find that $T_{obs} \leq T_{WCET}$, i.e., short delays may remain undetected, i.e., our model is probabilistic and may result in missed intrusion detection (false negatives), where our model does not identify an attacker in the network. This illustrates two points: (1) Our model complements existing cyber security measures and (2) the objective of T-Pack is to make the attack window that remains undetected as small (short) as possible.

4.5 Time Information inside a Packet

The objective of T-Pack is to verify that end-to-end times of a packet are within given WCET bounds. To this end, timing information is embedded within each packet. A custom header of our T-Pack network layer is added to a packet to indicate the time at which the packet was sent so that the receiver node is aware of the time of packet transmission on to the network by the sender. A custom header is utilized to provide the flexibility of resizing the packet such that the new header can be accommodated. It is assumed that this custom header is universally added for all traffic, just as given by the network or transport header, such that all nodes are aware of this header.

The objective of T-Pack is to extract information of the time the packet spends on the network excluding time due to processing within the kernel of any subsystem. By setting the timestamp value in the lower level of the network stack, the time for processing the packet within the kernel from the lower layer to the application layer and vice versa is excluded, which provides a better bound on the elapsed network delay.

Consequently, it was decided to place the custom header at the end of the packet after the data payload. Each hop on the network processes encapsulation and decapsulation of the frame header to reference the network header and extract the information to determine the next hop for routing. Hence, a custom header on top of network header would require header-aware switches and routers on the network with one more encapsulation and decapsulation per each packet, thereby slowing down network performance, which violates the end-to-end principle [Sal84]. Also, by appending the header at the end of the packet and not before the transport header, we eliminated the overhead of encapsulating and decapsulating the network header, which includes an additional memory write operation with the associated performance overhead.

4.6 Challenges

Handling time from different Subsystems: In two-way message transfers, it is possible that a receiver receives packets from two subsystems at the same time. To maintain the end-to-end time for both packets received before an acknowledgment is transmitted, a lookup table is maintained for each subsystem. In a real-time system, the number of subsystems involved in communication and the communication pattern with its timing is known a priori. Inclusion of TCP_NODELAY and TCP_QUICKACK ensures that only one outstanding packet from the same subsystem exists before an ACK is sent for the respective packet at any time. Hence, the lookup table stores the one way end-to-end time of the packet from each subsystem, which is added to timestamp in custom header of the ACK packet. Due to the static size of the lookup table, the execution time of looking up the value is constant, which makes bounds under T-Pack highly predictable. The burden for additional predictability lies with the application programmer: By designing subsystem communication such that no two subsystems issue a message to the same receiver at the same time, queuing delays in switches can be eliminated under the assumption of full-duplex point-to-point connectivity.

CHAPTER

5 IMPLEMENTATION

5.1 Linux

T-Pack is implemented in a PREEMPT-RT Linux kernel that provides real-time capabilities to the operating system as we saw in Chapter 2. Linux provides the flexibility of utilizing Linux network APIs such as socket buffers and Netfilter to implement T-Pack.

5.2 Netfilter

Netfilter is a framework provided by the Linux kernel to implement customized handlers on events in the network layer (for pre-routing, post-routing, etc.). T-Pack utilizes this framework to implement callback functions to insert and remove the custom header from a packet (Fig. 5.1).

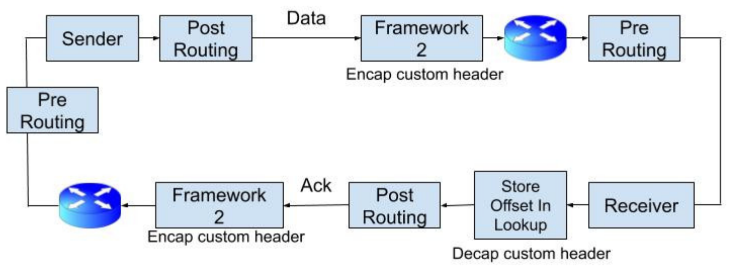


Figure 5.1 Netfilter Events Triggered by Custom Headers and Processed by Framework 2

5.3 Socket Buffers

Socket buffers are data structures provided by Linux as a common reference to packets in all layers of the network stack within the kernel. Socket buffer helper functions provide the ability to perform memory remapping of the outgoing or incoming packet within layers of the network stack. Some of the functions are shown in Fig. 5.2. Fig. 5.3 shows that the socket buffer maintains four memory pointers: head, data, tail and end. Space between the head and the end is allocated for the packet by the socket buffer initially when a socket file descriptor relays the socket information to the kernel which in turn initializes the socket buffer data structure representing the created packet. This works the same way for an incoming packet at the physical layer. Data and tail pointers in the socket buffer are used to expand above and below toward head and the end, respectively. Initial memory is allocated alongside the headroom space between data and head pointers by shifting data and tail toward the end after reserving headroom space using *skb_reserve*. Any header added to the packet is realize by increasing the data memory pointer toward the head using the *skb_pull* function and then copying the header starting from data pointer toward the tail using copy functions. *skb_put* is used to append the data by shifting the tail toward the end pointer returning the original position of the tail and then copying the data from the returned pointer to the tail. *skb_push* is used to push the data pointer toward the tail to pop headers from the packet as it passes through the network layers after receiving at the physical layer. Even though initially allocated memory for the packet is enough to accommodate standard intended encapsulation/decapsulation of the headers, memory can be expanded by functions like *pskb_expand* to reallocate existing linear memory and the additional added memory to the packet at a different memory location in RAM (as the packet has to be linearly allocated to avoid complications and performance degradation).

The T-Pack prototype utilizes these helper functions in the socket buffer API to manipulate packet memory in order to create additional space for the custom header (Fig. 5.4).

5.4 Implementation Framework

The overall framework (Figures 5.1 and 5.4) consists of the following components:

1. At the sender, the Netfilter post routing hook is utilized to call a handler with the socket buffer referencing the detected packet passed as an argument.

2. The $skb\_put(sizeof(custom\_header))$ function is used to create additional space at the end of the packet to attach the custom header. The header is appended to reduce the overhead of encapsulating and decapsulating the network header.

3. $memcpy(ptr, custom\_header)$ initializes the allocated space from previous step, encapsulating the custom header to the packet by copying it to that space. The custom header is a C structure with variables *ktime_t timestamp*, *long sendtime* and *long checksum*. The timestamp

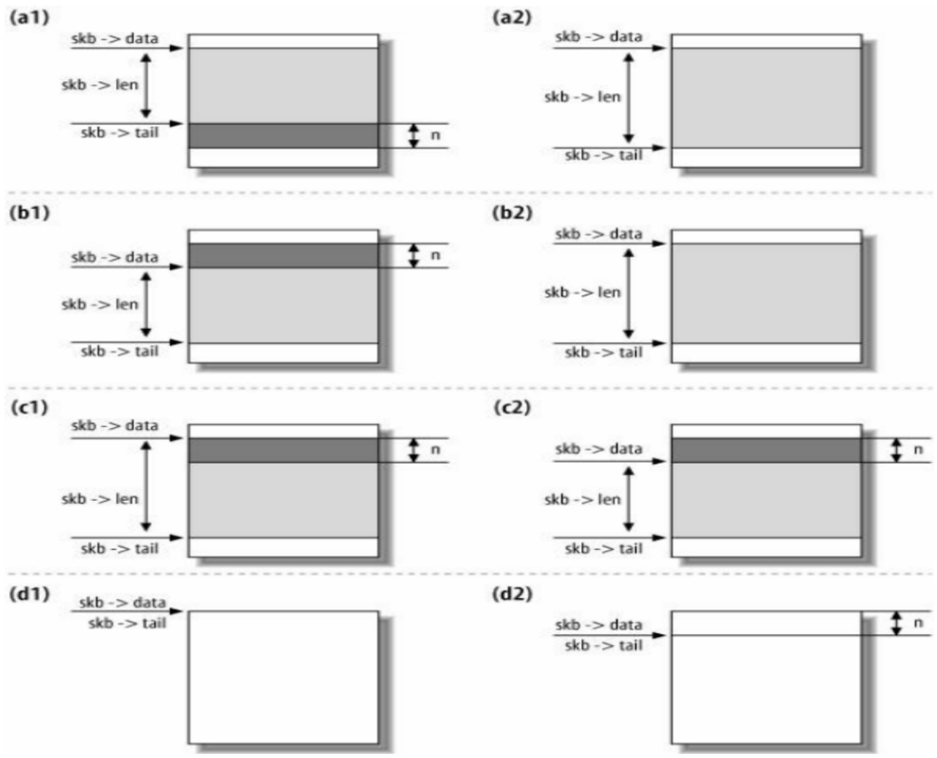
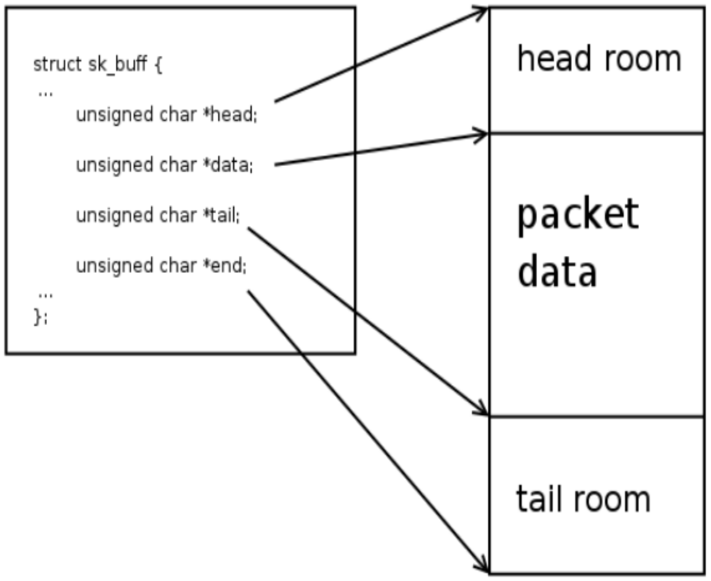


Figure 5.2 (a)skb_put, (b)skb_push, (c)skb_pull, (d)skb_reserve

image credits: SlideShare, Sourav Punoriyar (March 31, 2016)



```
struct sk_buff {
 ...
      unsigned char *head;
      unsigned char *data;
      unsigned char *tail;
      unsigned char *end;
 ...
};
```

Figure 5.3 Memory Mapping of Socket Buffer for Packet Representation

image credits: kernel.org

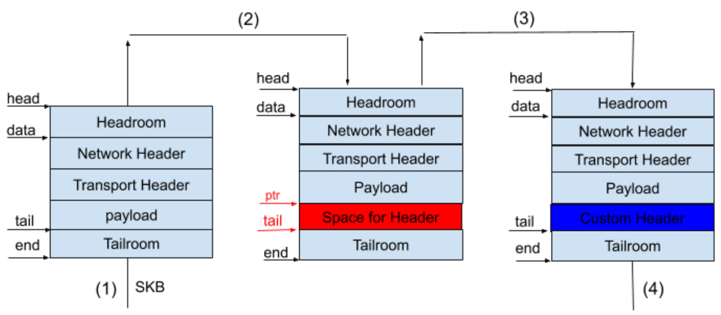


Figure 5.4 Framework 2: Custom Header Insertion/Removal using Socket Buffers

variable is used to store the current time, i.e., the time at which this header was created in the network layer.

- Upon UDP reception, the difference of the current time and the time denoted by the attached timestamp represents the end-to-end processing time of lower layer UDP activities, subject to validation against an expected upper bound for the exchange. If validation fails, an intrusion is signaled.
- Upon TCP reception, the difference of the current time and the time denoted by the attached timestamp is stored in a lookup table. The objective here is to capture the send cost of only the lower layers of the network stack, which provides tighter bound on worst-case packet handling. When a TCP receiver acknowledges, the timestamp is again populated with the current time and the sendtime from the corresponding entry in the lookup table. A TCP sender, when receiving the acknowledgment, will subsequently calculate the difference of current time and the timestamp (of the acknowledgment), which represents the end-to-end processing time of lower layer TCP activities, subject to validation against an expected upper bound for the exchange. If validation fails, an intrusion is signaled.
- On adding and removing of the custom header, the checksum value is verified to detect header corruption if any. This detects any corruption in the timestamp information, which helps avoiding T-Pack failures due to incorrect timestamps.

4. Checksum values for IP and Transport header are recalculated using $csum\_tcpudp\_magic()$, $csum\_partial()$ and $ip\_send\_check()$, respectively, before the modified packet is transmitted to the receiver.

At the receiver, the Netfilter pre-routing hook is utilized to call a handler with the socket buffer referencing the packet. The same functions as above are used to decapsulate the custom header and to reset the total length and the tail reference pointer of the packet to create the default packet without the custom header. We use TCP flag values to detect an acknowledgment and, once received, offset values are handled accordingly as discussed before.

The same network stack is traversed for each corresponding acknowledgment transmitted from the receiver with offset values set to data packet end-to-end time as indicated before.

Below are the pseudo codes of adding and removing the custom header from sender to receiver only.

Algorithm 1 Add Custom Header

```
1: procedure APPEND HEADER
2:     $header \leftarrow kmalloc(sizeof(custom\ header))$
3:     $header \longrightarrow time = current\ time$
4:     $ptr \leftarrow skb\_put(sizeof(header))$
5:     $memcpy(ptr, header)$
6:     Reset network and transport header length, Recalculate IP checksum
```

Algorithm 2 Remove Custom Header

```
1: procedure REMOVE HEADER
2:     $ptr \leftarrow skb \longrightarrow tail - \text{sizeof(custom header)}$
3:     $header \leftarrow kmalloc(sizeof(custom\ header))$
4:     $header \leftarrow ptr$
5:     $offset = \text{current time} - header \longrightarrow time$
6:     $memcpy(ptr, header)$
7:     Reset network and transport header length, Recalculate IP checksum
```

CHAPTER

6 EXPERIMENTAL FRAMEWORK

6.1 Experiment 1: Client Server Model

A client server model is implemented, where the client sends periodic messages to the server to resemble the network activity of a time triggered real-time system. UDP messages are used with an explicit reply packet from the server instead of an implicit acknowledgment (which is only part of TCP). Server and client paradigms are common in distributed systems, where the master provides a service to one or more clients.

In this setup, the client sends periodic messages to the server every 10ms. The messages are small enough to fit within a single packet. The server receives the message and immediately replies back with a similar type of message. This ping-pong message transfer is utilized to measure round trip time (RTT) at the client.

We measure the end-to-end RTT (from sender back to the sender) of the request at the application layer and the network layer to assess the benefits of locating our T-PACK functionality at network layer. We also measure the RTT at the application layer with and without the T-Pack implementation to analyze performance overhead of T-PACK itself.

6.2 Experiment 2 : Paparazzi UAV Model

The Paparazzi UAV [Zim12; Bri06] models a real-time system based on a traditional shared memory real-time control systems. A peer-to-peer network of 3 subsystems, including an auto pilot (AP), a fly by wire (FBW) control and a ground station (GS), connects the subsystems. As shown in Fig 6.1, we prototyped a model constrained to only Paparazzi's periodic messages scheduled between the above three subsystems. Each subsystem is connected via the TCP protocol with a persistent connection. As seen in Fig 6.1, the subsystems communicate with each other periodically transferring necessary information for flying by wire autonomously. T-Pack is integrated into the Paparazzi prototype as outlined in Chapter 5. The RTT is measured between autopilot and ground station to monitor the T-Pack functionality. A delay attack in form of Distributed Denial of Service (ICMP packet flooding / ping-of-death) [Cro03] is induced at the ground station using other nodes in the network as attackers to analyze the accuracy of T-Pack during the attack. We implemented this Paparazzi model on a network of Raspberry Pi systems with a Preempt RT patched Linux kernel to provide real-time capabilities.

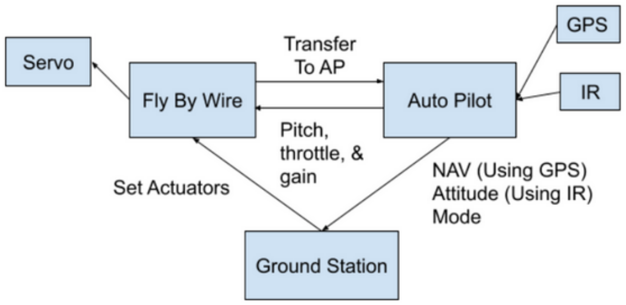


Figure 6.1 Paparazzi Model: Message Scheduling Between Subsystems

6.3 Experiment 3 : Waters Workshop Challenge 2018, a Drone-like Multi-System

This experiment also models a real-time system described in the Waters workshop challenge from 2018, a drone-like multi-system. A peer-to-peer network of seven subsystems is implemented consisting of a Mission management system (MMS), Electrical Propulsion System (EPS), Hydraulic Braking System(HBS), Sensors (communicating also with other sensors in the Waters model), Ground Station and Maintenance System, all connected via a hub and spoke topology within the same subnet. Similar to the Paparazzi experiment, we model functions and communication patterns in each subsystem as described in the challenge description, including periodic calls to the functions. Each subsystem is connected via the TCP protocol with a persistent connection. Each subsystem suggests changes to the dynamics of the drone and sends such a message to another relevant subsystem. These events occur in parallel and create a real-time scenario with random congestion on the network. T-Pack is implemented as discussed in Chapter 5. The RTT is measured between EPS and MMS to monitor T-Pack functionality. A delay attack between the two subsystems is induced to analyze the accuracy of T-Pack during the attack. As the paparazzi model this experiment is carried out by deploying the drone model on a network of Raspberry Pis, each with a Preempt RT patched

Linux kernel to provide real time capabilities.

CHAPTER

7 EXPERIMENTAL EVALUATION

7.1 Experiment 1

The results from the first experiment, the server client model prototyping a time-triggered real-time system, are depicted in Fig. 7.1. RTT is plotted (y-axis) for the message request and its reply from client to the server (x-axis). RTTs are shown as box plots indicating maximum, top quartile, median, bottom quartile and minimum times as well as outliers (dots). Min-max values or variability outside the upper and lower quartiles are denoted by whiskers. The outliers in this graph are only 0.5% of the total data values. We report all values from the experiments, even the first iteration of execution, which may be subject to additional cache misses resulting in an outlier.

We observe that time measured by T-Pack (variant 1) for the reduced network stack is much lower than the one measured by the application, both with and without T-Pack for the latter variants 2 and 3. This is because T-Pack measures the time the client request protocol spends between network layers of the two systems instead of including the upper application layers. This removes the time the request spends in the upper layers of the kernel, both at server and client. This experiment reveals a significantly tighter time bound for activities at and below the network layer between the server and the client: The results are dominated by the time spend for the request on the network, which allows detection of malware intrusion at the network layer to provide more accurate results than at upper layers. This shows the importance of measuring the time between the lower layers of the network stack, which can be achieved only within the kernel and by extending packet headers to include time stamps that are added within the kernel.

The results also reveal the overhead of implementing T-Pack to analyze its effect on the performance of a real-time system. To this end, the RTT of the client request is measured within the

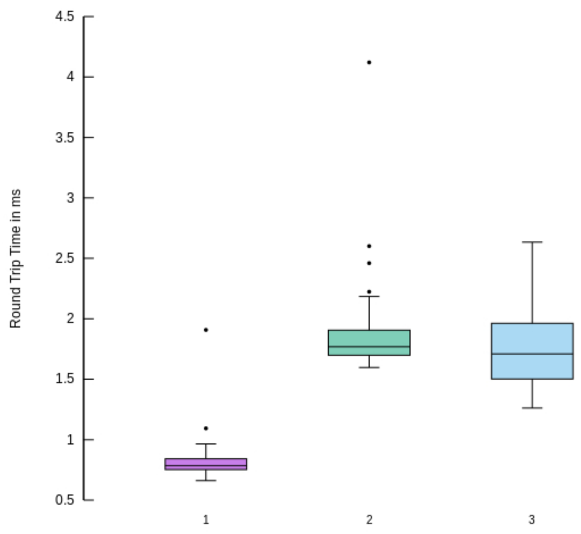


Figure 7.1 Box plot for RTT in ms measured by 1. T-Pack, 2. Application with T-Pack running, 3. Application without T-Pack running. The range of whisker is 3.5 times the inter-quartile range.

application of the client and server running (a) with the T-Pack module and (b) without it. Fig. 7.1 indicates that T-Pack incurs a modest cost in terms of performance on the application as the overall mean RTT of the client request-reply increases by a marginal amount of approximately 0.12 msecs.

7.2 Experiment 2

The experiment with the Paparazzi model assess how the T-Pack module aids in detecting intrusions due to attacks on the UAV system. We pick two subsystems as indicated in the experimental setup, the auto pilot (AP) and the ground station (GS). The auto pilot acts as the sender relaying necessary flight details to the ground station. We execute the Paparazzi prototype on top of our T-Pack implementation in the Linux kernel. We capture the RTT of TCP messages sent from AP to GS by measuring the time a message spent from AP to GS plus the time it took to receive an ACK from GS for the sent message. We introduce two additional attackers on the network executing a distributed denial-of-service attack via ping flooding. Each attacker utilizes a multi-threaded program to send large ICMP ping packets in quick intervals to the GS server. This causes a buffer overflow at the receiving interface, which is handled but causes performance degradation as a side effect. The attack intensity is investigated in a sensitivity study, indicated in Fig. 7.2 as a $P(n, t, b, i)$ tuple (parameter P), varying the number of attackers, n, the number of threads running within an attacker, t, the size of the ping packets, b, that transmitted to the server in bytes and the time interval, i, between each packet in seconds.

Fig. 7.2 depicts RTT as box plots again (y-axis) over different intensities of DDOS attack. The outliers in this graph are only 0.5% of the total data values (all data reported, including 1st iterations with additional cache misses). We observe that, as we keep increasing the attack intensity by modifying the attack parameters P, the RTT increases slightly. The attack scenario features a single attacker in the compromised network sending 500 byte ICMP ping packets from 10 parallel processes within intervals of 0.5 seconds. This affects the RTT by ≈ 0.01 msecs on average (Data set 1 vs. 2 on the x-axis of Fig. 7.2). Two attackers continuously sending packets (0 seconds interval) with otherwise the same parameters result in increased the RTT by ≈ 0.95 msecs on average (Data sets 1 vs. 6). Finally, two attackers with 30 processes each and 1KB packets within 1ms intervals increases the RTT by ≈ 3.5 msecs on average (Data sets 1 vs. 7). In this last scenario, all the measured RTT of data set 7 exceed those of data set 1 without any attack, i.e., without network intrusion. In other words, T-Pack can safely detect an intrusion whose "attack vector" (i.e., code footprint of the injection) affects the real-time system with similar time delay, since WCET bound of RTT without intrusion is lower than the minimum RTT on a compromised network. Recall that to take over an entire kernel, millions of instructions are typically required. We can limit an attack to 15k instructions here assuming, e.g., a CPU clock of 1GHz.

We also observe a sudden increase in the RTT in Fig. 7.2 when the intensity of an attack increases. Distributed denial of service prevents effective resource utilization by consuming most of the resources (network and receiver buffer in this case) by the attacker [Mir04]. A low intensity of such an attack does not affect the performance of a low traffic network, which is typical for a number of distributed real-time systems. However, intensifying the attack can cause sudden spikes that instantly degrade the network latency leading to packets arriving after a deadline if at all. T-Pack detects these cases, which allows a system to switch into offline mode while continuing to operate. Transitioning back online requires the attack source to be removed in a DDOS, whether it be in a real-time environment or commodity computing environment, i.e., counter measurements addressing the root cause remain unchanged and are beyond the scope of this paper.

Let us further analyze the results of frequency distributions shown in Figures 7.5, 7.4 and 7.3, which depict the number times (y-axis) a certain RTT (x-axis) was measured in our experiments. Figure 7.3 indicates the distribution for times without intrusion, P(0,0,0,0) — referred as Attack 1, and a compromised network affected by DDOS attack of intensity, P(2,30,1000,0.001) — Attack 7 (red). Results indicate that the intersection between the distributions is empty, i.e., both cases can be discretely distinguished for each of the experiments so that an attack with similar time delay effects as Attack 7 will always be detected by T-Pack. This is consistent with Fig. 7.2 that indicated the ability to flag instructions via T-Pack for similar delay attacks.

Fig. 7.4 depicts the distribution for times without intrusion, and a compromised network affected by DDOS attack of intensity, P(2,10,500,0) — Attack 6 (red). We observe a slight overlap between the blue (no attack) and red (with attack) curves ranging from 1-1.25 msecs. 1% of the samples fall into the 1-1.25 msecs range, i.e., 99% of the attacks would be detected by T-Pack for a compromised network with time delay similar to that caused in Attack 6.

Fig. 7.5 depicts distribution for times without intrusion, and a compromised network affected by DOS attack of intensity P(1,10,500,0.5) — Attack 2 (red). Fig. 7.5 indicates a significant overlap between the blue (no attack) and red (attack) curves ranging from ≈ 0.45-1 msecs. More than 99%

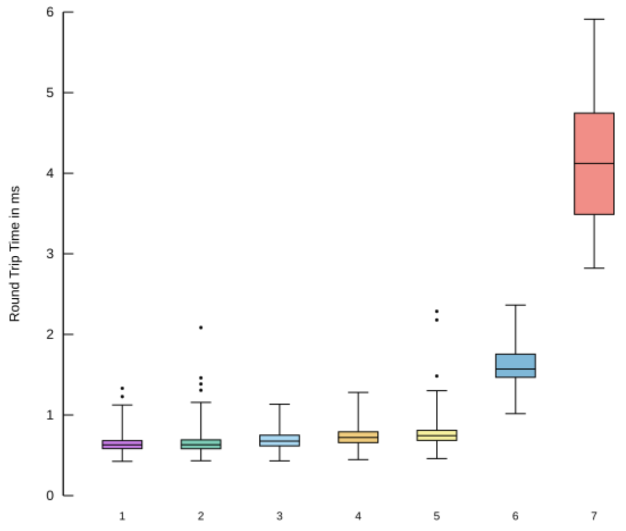


Figure 7.2 Box plot for RTT in ms measured by T-Pack under different attack conditions for P(n,t,b,i) parameters: 1 (0,0,0,0) (No Attack / Attack 1), 2 (1,10,500,0.5) (Attack 2), 3 (1,10,500,0.5) (Attack 3), 4 (2,10,500,0.1) (Attack 4), 5 (2,30,500,0.05) (Attack 5), 6 (2,10,500,0) (Attack 6), 7 (2,30,1000,0.001) (Attack 7). The range of whiskers is 5 times the inter-quartile range.

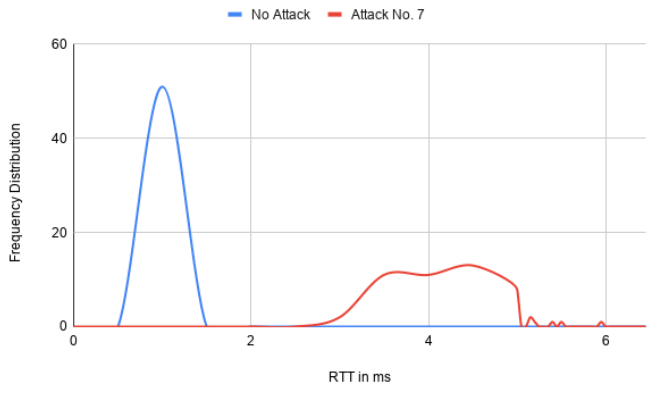


Figure 7.3 Frequency Distribution Curve and Overlapping Region

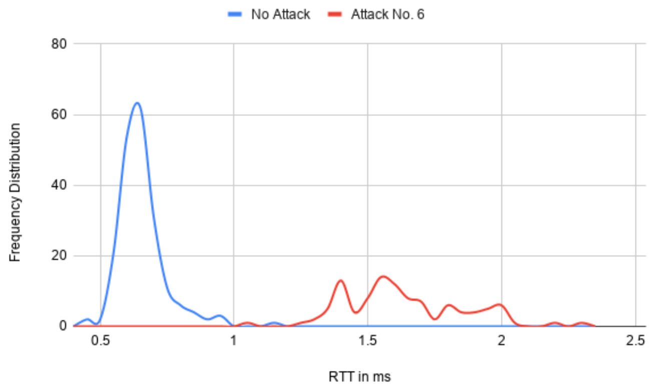


Figure 7.4 Frequency Distribution Curve and Overlapping Region

of the samples fall into this range. This illustrates the limitations of T-Pack. Any attack with similar delays will affect the system without being detected by T-Pack. This experiment also illustrates why T-Pack can only complement other system security methods, it cannot replace them as it is not a panacea for intrusions.

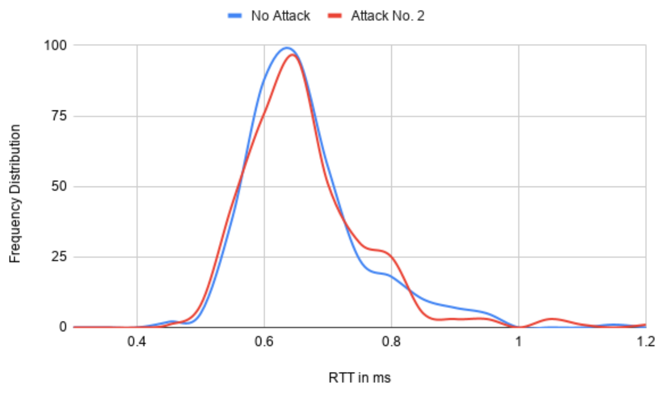


Figure 7.5 Frequency Distribution Curve and Overlapping Region

In summary, we observe that T-Pack is able to detect 100% of the attacks caused by a delay similar to Attack 7. Slightly milder attack intensities have 99% of chance to be detected (Attack 6) while low-intensity attacks (Attack 2) remain undetected 99% of the time, but will not affect real-time packets either as sufficient bandwidth remains on the network.

7.3 Experiment 3

The experiment with a drone-like multi-system assesses how the T-Pack module aids in detecting intrusions due to delay attacks on the drone network. We pick two subsystems as indicated in Chapter 6 Experimental Framework, the Electrical Propulsion System (EPS) and the Mission Management System (MMS), where EPS acts as sender and MMS as receiver. We run the drone communication model as described by the Waters challenge over our T-Pack model. We observe the RTT of TCP by measuring the time a message spent from EPS to MMS plus the time it took to receive ACK from MMS for the sent message. We introduce similar attack models as in experiment 2 for the Paparazzi system to assess performance and vulnerability of T-Pack on a drone-like multi-system model.

Fig 7.6 depicts RTT as box plots again (y-axis) over different intensities of DDOS attacks. The outliers in this graph are 0.5% of the total data (no data omitted, 1st iterations subject to additional cache misses). Similar to experiment 2, as the intensity of the DDOS attack parameters is increased, a slight increase in RTT is seen until a sudden and significant increase under more intense attacks. We again observe that the DDOS attack causes a large increase in average RTT of packets. Eventually, all measured values of RTT are greater than those without attack. This solidifies our previous findings that 100% of attacks are detected with via T-Pack for delays similar to that of Attack 7.

This experiment required two switches due to the number of port available per switch as more subsystems are connected in this drone model. The attackers were on a different switch than the MMS server, which caused the attack intensity to be slightly lower than that in the Paparazzi model. This explains the small decrease in RTT values in Fig. 7.6 during attacks relative to a system without attack or the Attack 1 scenario of Fig 7.2. RTT values without attack and for Attack 1 are slightly higher for the drone model compared to Paparazzi because of higher base traffic on the network given by the larger number of drone subsystems.

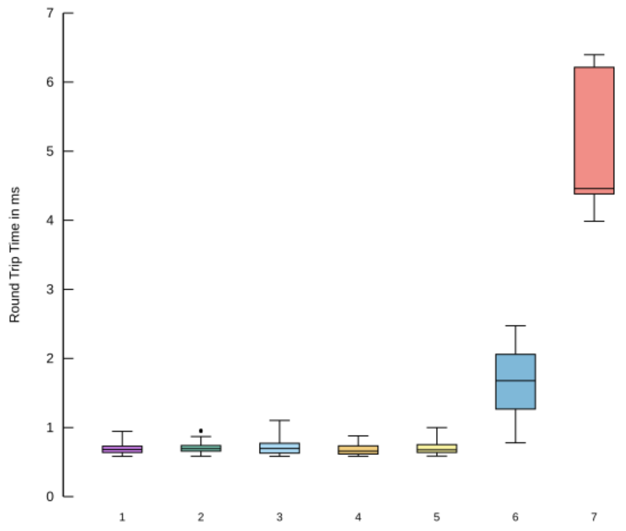


Figure 7.6 Box plot for RTT in ms measured by T-Pack under different attack conditions for P(n,t,b,i) parameters: 1 (0,0,0,0) (No Attack / Attack 1), 2 (1,10,500,0.5) (Attack 2), 3 (1,10,500,0.5) (Attack 3), 4 (2,10,500,0.1) (Attack 4), 5 (2,30,500,0.05) (Attack 5), 6 (2,10,500,0) (Attack 6), 7 (2,30,1000,0.001) (Attack 7). The Range of whiskers is 2.5 times the inter-quartile range.

We further analyze the results of frequency distributions depicted in Figures 7.8 and 7.7, which show the number of times (y-axis) a certain RTT (x-axis) was measured in our experiment. We can observe a similarity in the patterns to those in Figures 7.4 and 7.5, respectively.

Fig 7.7 reveals an overlapping region for the frequency distribution of RTT without attack (blue) and with attack (intensity P(2,10,500,0) — Attack 6), just as in Fig 7.4. 95% of the attacks lies outside this region detected by T-Pack leaving only 5% of the malicious packets undetected.

Fig 7.8 features an attack intensity of P(1,10,500,0.5) — Attack 2 — and a 99% overlap of the frequency distribution. This illustrates the vulnerability of T-Pack (also observed with Paparazzi model in experiment 2, see Fig 7.5). Here, an attack chaining of 66 instances with 15k instructions each would be required to take over the system without being detected. While this illustrates the limitations of T-Pack, it also shows that an attacker would need to carefully inject packets to remain unnoticed. A more complex, chained attack would need to be orchestrated by attackers skipping

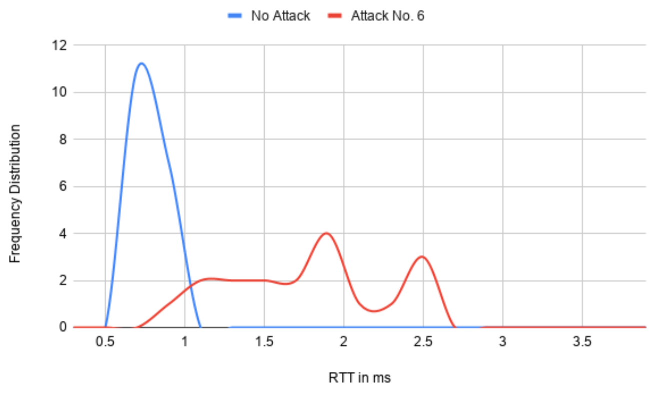


Figure 7.7 Frequency Distribution Curve and Overlapping Region

over intervals when the application actually uses the network. Again T-Pack can only complement other system security methods, it cannot replace them as it is not a panacea for intrusions.

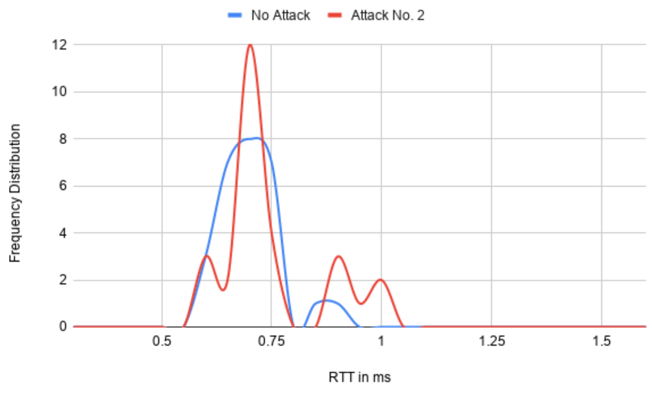


Figure 7.8 Frequency Distribution Curve and Overlapping Region

In summary, we observe consistency of the T-Pack model on different real-time systems and under different attack intensities for DDOS ping-of-death scenarios that affect network delays.

CHAPTER

8 CONCLUSION

This work contributes the design and implementation of a novel network timed security method, T-Pack, to detect malware intrusion in real-time system by timing end-to-end response times of packet delivery. Experimental results indicated that T-Pack successfully detected malware intrusion with almost 100% accuracy for attacks caused by distributed denial of service with 30 attackers each in two nodes sending 1KB ping packets intervals of 0.001 seconds. Results also confirmed that T-Pack can be implemented with only a small overhead to the overall network performance of the system relative to the range of delays for the attacks it protects against. Hence, the implementation and results demonstrated in this work support the hypothesis, monitoring the round trip time of periodic communication at the packet level in a real-time distributed system provides a means to detect network intrusions complementing conventional security methods.

BIBLIOGRAPHY

[Bak09] Bak, S. et al. “The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety”. *IEEE Real-Time Embedded Technology and Applications Symposium*. 2009, pp. 99–107.

[Bri06] Brisset, P. et al. “The paparazzi solution”. 2006.

[Bru05] Brumley, D. & Boneh, D. “Remote timing attacks are practical”. *Computer Networks* **48**.5 (2005), pp. 701–716.

[Buf] Buff, R. & Goldberg, A. *Web Servers Should Turn Off Nagle to Avoid Unnecessary 200 ms Delays*. Tech. rep.

[Cal09] Callegati, F. et al. “Man-in-the-Middle Attack to the HTTPS Protocol”. *IEEE Security & Privacy* **7**.1 (2009), pp. 78–81.

[Che11] Checkoway, S. et al. “Comprehensive experimental analyses of automotive attack surfaces.” *USENIX Security Symposium*. Vol. 4. San Francisco. 2011, pp. 447–462.

[Che18] Chen, C.-Y. et al. “Securing real-time internet-of-things”. *Sensors* **18**.12 (2018), p. 4356.

[Con99] Conover, M. *w00w00 on heap overflows*. 1999.

[Cre07] Crenshaw, T. et al. “The Simplex Reference Model: Limiting Fault-Propagation Due to Unreliable Components in Cyber-Physical System Architectures”. *IEEE Real-Time Systems Symposium*. 2007, pp. 400–412.

[Cro03] Crosby, S. A. & Wallach, D. S. “Denial of Service via Algorithmic Complexity Attacks.” *USENIX Security Symposium*. 2003, pp. 29–44.

[Dhe98] Dhem, J.-F. et al. “A practical implementation of the timing attack”. *International Conference on Smart Card Research and Advanced Applications*. Springer. 1998, pp. 167–182.

[Fra11a] Francillon, A. et al. “Relay attacks on passive keyless entry and start systems in modern cars”. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Eidgenössische Technische Hochschule Zürich, Department of Computer Science. 2011.

[Fra11b] Francis, L. et al. “Practical Relay Attack on Contactless Transactions by Using NFC Mobile Phones.” *IACR Cryptology ePrint Archive* **2011** (2011), p. 618.

[Gor04] Gorrieri, R. et al. “Automated analysis of timed security: a case study on web privacy”. *International Journal of Information Security* **2**.3-4 (2004), pp. 168–186.

[Hat14] Hattenberger, G. et al. “Using the paparazzi UAV system for scientific research”. 2014.

[Hor02] Horovitz, O. “Big loop integer protection”. *Phrack Inc., Dec* (2002).

[Tes] *https://en.wikipedia.org/wiki/Tesla_Autopilot*.

[Kim13] Kim, G. ho et al. “Vehicle Relay Attack Avoidance Methods Using RF Signal Strength”. *Communications and Network* **5** (2013), pp. 573–577.

[Koc96] Kocher, P. C. “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems”. *Annual International Cryptology Conference*. Springer. 1996, pp. 104–113.

[Kos10] Koscher, K. et al. “Experimental security analysis of a modern automobile”. *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 447–462.

[Lou] Lou, X. et al. “Assessing and Mitigating Impact of Time Delay Attack against Cyber-Physical Systems” ().

[Lou19] Lou, X. et al. “Assessing and mitigating impact of time delay attack: a case study for power grid frequency control”. *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*. ACM. 2019, pp. 207–216.

[Mar15] Mary, C. “Shellshock attack on linux systems–bash”. *International Research Journal of Engineering and Technology* **2**.8 (2015), pp. 1322–1325.

[Mil91] Mills, D. L. “Internet time synchronization: the network time protocol”. *Communications, IEEE Transactions on* **39**.10 (1991), pp. 1482–1493.

[Mir04] Mirkovic, J. & Reiher, P. “A taxonomy of DDoS attack and DDoS defense mechanisms”. *ACM SIGCOMM Computer Communication Review* **34**.2 (2004), pp. 39–53.

[Nag] Nagle, J. *Congestion Control in IP/TCP Internetworks*. `https://tools.ietf.org/html/rfc896`.

[Nar18] Narula, L. & Humphreys, T. E. “Requirements for secure clock synchronization”. *IEEE Journal of Selected Topics in Signal Processing* **12**.4 (2018), pp. 749–762.

[One96] One, A. “Smashing the stack for fun and profit”. *Phrack magazine* **7**.49 (1996), pp. 14–16.

[Ped16] Pedroza, G. et al. “Timed-model-based Method for Security Analysis and Testing of Smart Grid Systems”. *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE. 2016, pp. 35–42.

[Sal84] Saltzer, J. H. et al. “End-to-end arguments in system design”. *Technology* **100** (1984), p. 0661.

[Sar15] Sargolzaei, A. et al. “A novel technique for detection of time delay switch attack on load frequency control”. *Intelligent Control and Automation* **6**.04 (2015), p. 205.

[Sch00] Schindler, W. “A timing attack against RSA with the chinese remainder theorem”. *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2000, pp. 109–124.

[SHA18] SHAAR, F. & EFE, A. “DDoS attacks and impacts on various cloud computing components”. *Int. Journal of Information Security Science* **7** (2018), pp. 26–48.

[Wil08] Wilhelm, R. et al. “The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools”. *ACM Transactions on Embedded Computing Systems* **7**.3 (2008), pp. 1–53.

[Zim10] Zimmer, C. et al. “Time-Based Intrusion Dectection in Cyber-Physical Systems”. *International Conference on Cyber-Physical Systems*. 2010, pp. 109–118.

[Zim12] Zimmer, C. & Mueller, F. “Fault resilient real-time design for noc architectures”. *2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*. IEEE. 2012, pp. 75–84.