Designing and Modeling of Covert Channels in Operating Systems

Yuqi Lin, Saif U. R. Malik, Member, IEEE, Kashif Bilal, Qiusong Yang, Yongji Wang, and Samee U. Khan, Senior Member, IEEE

Abstract—Covert channels are widely considered as a major risk of information leakage in various operating systems, such as desktop, cloud, and mobile systems. The existing works of modeling covert channels have mainly focused on using Finite State Machines (FSMs) and their transforms to describe the process of covert channel transmission. However, a FSM is rather an abstract model, where information about the shared resource, synchronization, and encoding/decoding cannot be presented in the model, making it difficult for researchers to realize and analyze the covert channels. In this paper, we use the High-Level Petri Nets (HLPN) to model the structural and behavioral properties of covert channels. We use the HLPN to model the classic covert channel protocol. Moreover, the results from the analysis of the HLPN model are used to highlight the major shortcomings and interferences in the protocol. Furthermore, we propose two new covert channel models, namely: (a) Two Channel Transmission Protocol (TCTP) model and (b) Self-Adaptive Protocol (SAP) model. The TCTP model circumvents the mutual inferences in encoding and synchronization operations; whereas the SAP model uses sleeping time and redundancy check to ensure correct transmission in an environment with strong noise. To demonstrate the correctness and usability of our proposed models in heterogeneous environments, we implement the TCTP and SAP in three different systems: (a) Linux, (b) Xen, and (c) Fiasco.OC. Our implementation also indicates the practicality of the models in heterogeneous, scalable and flexible environments.

Index Terms—covert channels, high-level Petri nets (HLPN), operating systems, modeling and security

1 INTRODUCTION AND MOTIVATION

The emergence of technological advances, such as virtualization and mobile computing, has brought us into an environment where various computing systems coexist. Windows and Unix-like operating systems still play important roles in desktop and laptop computers. Similarly, mobile operating systems, such as Android [1] and Apple iOS are advancing at an astonishing pace. Virtualization-based cloud computing [2] is widely considered to be next generation of computing. Microkernels [23] are attracting much more attentions due to special security features. These operating systems (OSes) have their respective application fields and compose a whole computing environment together. Regardless of the underlying OSes, security is one of the most concerned topics and covert channel is considered to be a major threat of information leakage in the OSes.

Covert channel [4] is a means of communication between two entities that are unauthorized or prohibited to communicate with each other. Covert channel exploits imperfections in the isolation of shared resources between two unrelated entities and enables communications between them via unintended channels, bypassing mandatory auditing and access controls placed on standard communication channels [8]. Covert channel can be constructed using a variety of shared media by attackers. Numerous studies [1], [5 - 8] have shown that covert channels are a threat to the information security of databases, networks, and desktop operating systems. Some recent researches [1], [6] also highlight the threat of covert channel to cloud computing and mobile computing. Even in Sel4 [23], a formally verified microkernel, covert channel is still specified to be a potential threat of information leakage [3].

The Finite State Machine (FSM) [9] and its transforms [10 - 12] are typically used by the researchers to model the covert channels. As shown in Fig. 1, a FSM describes the state change of shared media and the messages conveyed. The model works well on evaluating the upper and lower limits of channel capacity. However, a FSM is rather an abstract model, where information about the operation...
sequences, synchronization, and operations on shared resources cannot be characterized, leading it hard for researchers to understand the process of a covert transmission. Except the capacity evaluation, the FSM exhibits very limited ability to perform covert channel analysis, such as the process of covert channel construction, detection of covert channel attacks, and covert channel restriction mechanism.

In this paper, we use High-Level Petri Nets (HLPN) to model covert channels in a more concrete form. The HLPN [13] is a classical transform of Petri net. A detailed discussion of HLPN is provided in Section 2. Modeling the covert channels using HLPN can benefit researchers in following aspects: (a) Covert channel construction: It’s much easier for researchers to construct covert channel attacks based on the HLPN model as it describes the process and operations of the sender and receiver concretely. (b) Threat estimation: More details on timing sequence and system calls contained in the model make it more accurate to estimate the threat of a channel, such as the transmission rate and max bandwidth. (c) Covert channel detection: Lots of works use probability statistics based methods to detect covert channels. Information in the HLPN model can help to refine the probability model and improve the detection accuracy. (d) Channel restriction: Security analysts can discover the key operations in a covert channel attack and design related restriction mechanism through model analysis.

Our aim is to perform the modeling of the covert channels to present the communication process in detail, regardless of the underlying platform. We use the HLPN to model the classic covert channel protocol by providing mathematical representation and analyzing the behavior of the sender, receiver, and shared media. The results from the analysis are then used to identify the factors that reduce the transmission accuracy. Moreover, the issues identified during the analysis of classic covert protocol motivated us to propose an improved protocol called Two Channel Transmission Protocol (TCTP). Finally, we take the noise and interference into account and improve the TCTP into Self-Adaptive protocol (SAP). To demonstrate the correctness and practicability of our models in covert channel construction, we implement and evaluate TCTP and SAP in three representative heterogeneous OSes: (a) Linux Fedora [24], (b) Xen [19], and (c) microkernel Fiasco.OC [25]. Using our models researchers can automate the construction of various covert channels on different platforms. The proposed TCTP model proves to deliver higher accuracy in an environment with little noise. In case of strong noises, the SAP model ensures the correctness of transmission. An experiment on transmission rate estimation also demonstrates the effectiveness of HLPN on modeling covert channels.

Contributions of this paper consist of the following:

(a). We simulate the classic covert channel protocol [6] and provide mathematical representation to analyze the behavior and structural properties of the protocol by HLPN.

(b). We discover the limitations in the classic protocol by model analysis and improve it to the TCTP model, which achieves much higher accuracy with minor decrease in transmission rate in normal environments.

(c). We take noises into account and improve the TCTP model into SAP model, which achieves more than 90% accuracy in presence of strong noise where the classic protocol only gets less than 30%.

(d). We introduce the process of automatically constructing covert channels in different OSes and estimating the capacity of covert channels based on the HLPN models. The results demonstrate HLPN performs much better than traditional FSMs on covert channel analysis.

The rest of the paper is organized as follows: Section 2 presents some preliminaries tools and technologies used in this paper. Description, modeling, and analysis of the classic covert channel protocol are discussed in Section 3. Design, modeling, and analysis of TCTP are explained in Section 4. Section 5 presents the SAP in detail. The experiments performed to demonstrate the effectiveness of our proposed strategies are discussed in Section 6. In Section 7, we provide some related work done in the field of covert channel modeling and construction. Section 8 concludes the paper.

2 PRELIMINARIES

This section will discuss some of the technologies used in this work to elaborate the concepts and improve readability of the paper.

2.1 High-Level Petri Nets

Petri nets [13] are graphical and mathematical modeling tool that is applicable to many systems characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, or stochastic. In this paper, we have used a variant of the classical Petri Net model, namely, HLPN.

**Definition 1(HLPN)** [15] A HLPN is a 7-tuple $N = (P, T, F, ψ, R, L, M₀)$ here:

1. $P$ is a set of finite places.
2. $T$ is a set of finite transitions such that $P ∩ T = ∅$.
3. $F$ is a flow relation such that $F \subseteq (P \times T) \cup (T \times P)$.
4. $ψ$ is a mapping function that maps $P$ to data types such that $ψ: P \rightarrow \text{Type}$.
5. $R$ define rules that maps $T$ to predicate logic formulas such that $R: T \rightarrow \text{Formula}$.
6. \( L \) is a label that maps \( F \) to labels such that \( L: F \rightarrow Label \).
7. \( M_0 \) is the initial marking where \( M_0: P \rightarrow Tokens \).

Places (\( P \)) store tokens to indicate the state of a HLPN; Transitions (\( T \)) cause flow of tokens to indicate the state change; Flow relation (\( F \)) associates \( P \) with \( T \). Further describe the attributes of \( (P, T, F) \) respectively as introduced in Definition 1.

In HLPN, places can have tokens of different types and can also be a cross product of two or more types, such as in Fig. 2 the places are mapped to the types: \( \psi(P1) = \text{Bool}, \psi(P2) = \text{ID}, \psi(P3) = \text{Int}, \psi(P4) = \text{Char} \).

The preconditions must hold for any transition to be enabled. Moreover, the variables from the incoming flows are used to enable a certain transition. For example, preconditions for \( t_1 \) will use \( x \) and \( y \) from \( P1 \) and \( P2 \), respectively. Similarly, post condition uses variables from outgoing flows for transition firing, and can be written as: \( R(t1) = (x = \text{true}) \land (y \geq 50) \land (4 \leq z \leq 20) \).

There are four steps to construct a HLPN model in general: a) determine the places (\( P \)), transitions (\( T \)) and flows (\( F \)) based on the modeled object; b) map \( P \) to concrete data types ; c) map \( F \) to corresponding labels; d) mathematically describe the preconditions and post conditions for each \( T \). Further details and examples are presented in Section 3, 4, 5 where we model three different covert protocols by HLPN respectively.

2.2 Covert Channel Communication

During a typical transmission cycle of a covert channel, the confidential information is transmitted from the sender to receiver. The sender encodes the information into binary bits and changes the properties of the shared resources according to the bits. The receiver observes the changes, and decodes the confidential information from the changes. The sender and receiver predetermine the parameters and repeat the cycle until all the confidential information has been transmitted.

The sender, receiver, and the shared resources in covert communications vary according to the underlying OSes. The communication scenarios in the three different OSes are introduced respectively as follows:

**Unix-like OS:** The sender and receiver are two processes, which are limited to communicating with each other by access control or other security policies. The shared resources could be the number of processes, number of free file system nodes, temporary files and etc. Take the last\_pid channel [16] as an example: the variable last\_pid shared by all the processes is the maximum PID (process ID) allocated to the processes. When a new process is forked, last\_pid will be checked and increased. Whatever security policy is deployed, the processes in different levels share the value of last\_pid all the time. Covert messages can be transferred from the high level host to the low level object by altering and viewing this value [16].

**Xen-based virtualized environment:** The Xen hypervisor is an open source virtualization solution for various platforms including x86, x86 64, IA64, and ARM [2]. Xen supports both full virtualization and Para-virtualization and manages the resource of virtual machines using a reference hypervisor.

Ristenpart et al. [6] first introduced the concept of cross-vm covert channels. Their basic idea is to construct certain patterns of contention on the hardware resources shared by two co-located VMs and use the contention patterns to encode information. For example, to send a single bit via a shared hard disk, attackers may let both the sender and the receiver VMs operate on large files concurrently for a specific period of time. During that time, the sender can choose to read files or do nothing to represent bit one or zero. At the same time, the receiver can distinguish the two by timing its own disk operations to decode the information.

**Microkernel based OS:** Microkernel minimizes the functionality that is provided by the kernel. The kernel provides a set of general mechanisms, while user-mode servers implement the actual OS services. User code obtains a system service by communicating with servers via an Inter-Process Communication (IPC) mechanism, typically message passing. Therefore, IPC is on the critical path of any service invocation and this is the main difference between microkernel and traditional OS.

Besides global IDs and hardware-based channels like in Unix-like OS and Xen platform [3], a unique channel called IPC-based channel threatens the isolation in microkernel. The basic idea is to use normal IPCs to encode information. For example, there are two legal IPCs (ipc1 and ipc2) between two tasks; the attacker use ipc1 to represent sending bit one and ipc2 sending bit zero. Confidential information could be transmitted in a series of ipc1 and ipc2 operations.

3 MODELING AND ANALYSIS OF CLASSIC COVERT CHANNEL PROTOCOL

In this section we perform the formal modeling and analysis of the classic covert channel protocol. We use the HLPN to model the behavior and information flow of the protocol. The models serve as the basis for the performing formal verification of the underlying protocols. We performed the verification of the models using SMT-lib and Z3. However, because of the page limitations and restricted scope of the paper, we do not include the verification of the models in the paper.

3.1 The Classic Covert Channel Protocol

The classic covert channel protocol includes basic steps of...
a covert transmission. The most famous example is the L2 cache-based covert channel proposed by [6], further studied in [7], [16–17] that divided all the cache lines into two subsets (a and b). To send a bit, the sender evicts the receiver cache content from the cache lines that corresponds to one subset and leaves the other untouched by accessing the memory addresses mapped to the chosen cache lines. Then, the receiver can decode the information by comparing the time in accessing the two subsets separately, and if subset a takes a significantly longer time to read as compared to subset b, it is bit one; otherwise it is bit zero. This process is illustrated by Fig. 3.

### 3.2 Modeling and Analysis

The model of the classic protocol is illustrated in Fig. 4. The model ignores the characteristic of the platform and abstracts the state change of shared resource and the operation caused the change. As stated in Definition 1, the HLPN is a 7-tuple $N = (P, T, \psi, R, L, M_0)$. To begin modeling the protocol, we first need to specify $P$ and the associated types. As depicted in Fig. 4, there are 10 places in the model. The names and mapping of $P$ are shown in Table 1. The types used in the model are illustrated in Table 2.

The next step is to define the set of rules, preconditions, and post conditions to map to $T$. The set of transitions $T = \{\text{Start}, \text{Get1bit}, \text{Send0}, \text{Send1}, \text{Read0}, \text{Read1}, \text{Store1bit}\}$.

$P(\text{Rdy}_{-}\text{St})$ stores the token $\text{Ready}$ representing the sender and receiver are both ready for the transmission. The $\text{Start}$ transition is mapped to the following formula:

$$
\begin{align*}
R(\text{Start}) & = \forall v_r \in R | v_r = \text{TRUE} \land \\
& \forall v_r \in R | \text{ref } 1 \Rightarrow v_r \land \\
& \text{ref }' = \text{ref } \cup \{(\text{ref } 1, \text{ref } 2)\} \\
\end{align*}
$$

(1)

The formula in (1) depicts the start condition of the whole transmission, where the $\text{Ready}$ token in $P(\text{Rdy}_{-}\text{st})$ is true. After (1) is executed successfully, the $\text{Ready}$ token in $P(\text{Info}_{-}\text{Tosend})$ is set to true for the associated sending queue. The next transition $\text{Get1bit}$ extracts one bit from the sending-queue and copy it to the ready to send buffer, as represented in (2).

$$
\begin{align*}
R(\text{Get1bit}) & = \forall v_b \in \text{Inf}_{-}\text{Tos} | v_b[1] = \text{True} \land \\
& v_b[2] \neq \text{NULL} \land v_b[2] = \text{Pop}(v_b[2]) \land \\
& \forall v_b \in \text{ITS} | v_b := \text{Top}(v_b[2]) \land \\
& \text{Inf}_{-}\text{Tos}' = \text{Inf}_{-}\text{Tos} \cup \{v_b[1], v_b[2]\} \land \\
& \text{ITS}' = \text{ITS} \cup \{v_b}\} \\
\end{align*}
$$

(2)

While analyzing the classic model, we identify a problem in (2). The formula does not take into account the state of $P(\text{Oneb}_{-}\text{Tosend})$ that causes $P(\text{Oneb}_{-}\text{Tosend})$ to store an incoming token $\text{bit}_\text{sending}$. If the previous bit has not been sent yet, then the previous bit will either be lost or the transmission of the bits will be out of order. Although, we can add a judgment in (2) to stop the disorder, we found that this reflects a defect in the classic protocol design. A detailed discussion of the aforesaid is provided in Section 3.3.

The next $\text{Send0}$, $\text{Send1}$, and $\text{Read0}$, $\text{Read1}$ are four abstract transitions which describe the process of sending and reading one bit. As our model focuses on portability and versatility, we describe the operations in an abstract mode instead of concrete ways because the operations and shared resources may be different due to platforms or covert channels. In Section 6, some concrete examples are introduced. $\text{Send0}$ and $\text{Send1}$ illustrate the process of sender selecting the operations and changing the state of shared resource, based on the bit to send. The rules for the aforesaid transitions are mapped to (3) and (4).

### Algorithm 1 Classic Covert Channel Protocol

```plaintext
Cache[2]: A shared processor cache, divided into two regions. Each cache region can be put in one of the two states, cached or flushed.

$D_{\text{Send}0}, D_{\text{Read}1}$: $N$ bits data to transmit and receive, respectively.

**Sender Operations**

Wait for receiver to initialize the cache;

**Receiver Operations**

access memory maps to Cache[0] and Cache[1]; put Cache[0] and Cache[1] into flushed state;

**Step send:**

1. if $D_{\text{Send}0}[i] = 1$ then
   access memory maps to Cache[0];
   put Cache[0] into flushed state;
2. else $D_{\text{Send}0}[i] = 0$ then
   access memory maps to Cache[1];
   put Cache[1] into flushed state;

**Step receive:**

1. timed access memory maps to Cache[0];
   detect the state of Cache[0] by latency;
2. timed access memory maps to Cache[1];
   detect the state of Cache[1] by latency;
3. if $\text{Time}_{-}\text{Cache}[0] > \text{Time}_{-}\text{Cache}[1]$
   $D_{\text{Read}1}[i] := 1$; Cache[0] is flushed;
4. else $D_{\text{Read}1}[i] := 0$; Cache[1] is flushed;

end if;

repeat Step send and receive until all bits sent.
```
Fig. 4. Model of Classic Protocol.

TABLE 1
PLACES AND MAPPINGS OF CLASSIC PROTOCOL

<table>
<thead>
<tr>
<th>Place</th>
<th>Mapping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rdy_St</td>
<td>P(Ready)</td>
<td>Holds the flag indicating sender and receiver are both ready.</td>
</tr>
<tr>
<td>Info_Tosend</td>
<td>P(Ready × SQueue)</td>
<td>Holds the sending queue and ready flag.</td>
</tr>
<tr>
<td>Oneb_Tosend</td>
<td>P(BitSending)</td>
<td>Holds the bit to be sent.</td>
</tr>
<tr>
<td>Shar_Res</td>
<td>P(S0 × S1 × R0 × R1)</td>
<td>Holds the state of the shared resource.</td>
</tr>
<tr>
<td>Oneb_Sent</td>
<td>P(BitSending)</td>
<td>Holds the bit that has been sent.</td>
</tr>
<tr>
<td>Oneb_Rec</td>
<td>P(BitSending)</td>
<td>Holds the bit that has been received.</td>
</tr>
<tr>
<td>Info_Store</td>
<td>P(RQueue)</td>
<td>Holds the receiving queue.</td>
</tr>
</tbody>
</table>

\[
R(\text{Send0}) = \forall b \in \text{OTS} \mid b = \text{'0'}' \land \\
\forall sr \in \text{Sh}_R \mid sr[1] = \text{TRUE} \land \\
\text{OB} = \text{OB} \cup \{b\} \land \\
b := \text{NULL} \land \text{OTS} = \text{OTS} \cup \{b\}
\]

(3)

\[
R(\text{Send1}) = \forall b \in \text{OTS} \mid b = \text{'1'}' \land \\
\forall sr \in \text{Sh}_R \mid sr[2] = \text{TRUE} \land \\
\text{OB} = \text{OB} \cup \{b\} \land \\
b := \text{NULL} \land \text{OTS} = \text{OTS} \cup \{b\}
\]

(4)

After (3) or (4) is fired, the bit\textunderscore sending token is extracted from P(Oneb\_Tosend) and stored in P(Oneb\_Sent), indicating one bit is sent but has not been read. The next two transitions, Read0 and Read1, depict the process as receiver senses the state change of the shared resource and read the bit. Once the token in P(Oneb\_Sent) is not null and the shared resource for bit reading is available, Read0 or Read1 will be fired based on the value of the bit. The said transitions will move the bit on sending from P(Oneb\_Sent) to P(Oneb\_Rec). The two transitions are mapped to (5) and (6), respectively. The last transition is Store1bit, which extracts the Bit\_Sending to the receiver’s R\_Queue. It will be fired only if: (a) the receiving queue is not full; (b) the Bit\_Sending is not null. The transition is mapped to (7).

3.3 Protocol Limitations

As discussed in Section 3.2, transition Get1bit ignores the state of P(Oneb\_Tosend) as described in formula (2), leading that the sender will continuously send information to the receiver until all of the bits are transmitted. The problem with the aforesaid mechanism is that transmission between the sender and receiver is independent, which can cause out of order bit transmission and unacknowledged bit failure. In case of covert channel, the synchronization between the sender and receiver is very important, because information is encoded and decoded by the state change of the shared resource. In covert channels, the shared resource cannot express previous information at all. Therefore, if the sender and receiver are not synchronized, then the sender’s successive operations to transmit

TABLE 2
DATA TYPES USED IN THE MODEL OF CLASSIC PROTOCOL

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready</td>
<td>A bool type for indicating whether the sender and receiver are ready for the transmission.</td>
</tr>
<tr>
<td>S_Queue</td>
<td>A bit array type for information to send.</td>
</tr>
<tr>
<td>Bit_Sending</td>
<td>A bit type for the bit in transmitting.</td>
</tr>
<tr>
<td>S0,S1,R0,R1</td>
<td>Bool types for indicating whether the state of shared resource representing sending 0, sending 1, reading 0 and reading 1 respectively.</td>
</tr>
<tr>
<td>R_Queue</td>
<td>A bit array type for storing the received bits.</td>
</tr>
</tbody>
</table>

\[
R(\text{Read0}) = \forall bs \in \text{Onb}\_S \mid bs = \text{'0'}' \land \\
\forall sr \in \text{shared}_R \mid sr[3] = \text{TRUE} \land \\
\text{Oneb}_R' := \text{Onb}_R \cup \{bs\} \land \\
bs := \text{NULL} \land \text{Onb}_S' := \text{Onb}_S \cup \{bs\}
\]

(5)

\[
R(\text{Read1}) = \forall bs \in \text{Onb}\_S \mid bs = \text{'1'}' \land \\
\forall sr \in \text{shared}_R \mid sr[4] = \text{TRUE} \land \\
\text{Oneb}_R' := \text{Onb}_R \cup \{bs\} \land \\
bs := \text{NULL} \land \text{Onb}_S' := \text{Onb}_S \cup \{bs\}
\]

(6)

\[
R(\text{Store1bit}) = \forall br \in \text{Onb}_\text{Rd} \mid br \neq \text{NULL} \land \\
\forall vb \in \text{Inf}_S \mid \text{Length}(vb) < \text{Size}(vb) \land \\
\text{vb} := \text{Push}(br) \land \text{Inf}_S' := \text{Inf}_S \cup \{vb\} \land \\
br := \text{NULL} \land \text{Onb}_\text{Rd}' := \text{Onb}_\text{Rd} \cup \{br\}
\]

(7)
Several bits will result only in retrieving the last bit sent by the sender at the receiver’s end. As depicted in Algorithm 1, the sender and receiver wait after performing the send and receive steps, respectively. However, the problem resides in how the both sides (sender and receiver) will come to know that operation from the other side is complete and when to start the next step.

To solve the aforesaid problems, we propose TCTP (Two Channel Transmission Protocol). The modeling and analysis of TCTP is presented in next section.

4 DESIGN, MODELING AND ANALYSIS OF TCTP

In this section, we present our technique to tackle the existing difficulties in classic covert channel protocol and develop a reliable protocol called TCTP. In later subsections we will thoroughly discuss the design, modeling, and analysis of the proposed protocol.

4.1 Design

Considering a normal communication protocol, the receiver will send back an acknowledgement (ack) message when receiving a message from the sender, which is ignored in the classic covert protocol. We use the same idea and propose a new protocol. The transmission mechanism of TCTP is shown in Fig. 5.

The transmission of one bit requires 4 steps: (a) sending the bit, (b) sending S_Ack, (c) reading the bit, and (d) sending R_Ack. The first two are operated by the sender and the last two by the receiver. Not only the receiver, but also the sender should send out an ack to the receiver to indicate that the encoding operation on shared resource has been finished. The receiver starts examining the state of shared resource after getting the ack from the sender.

The next step after ascertaining the ack messages from both the sender and receiver is to determine the way to implement them. In a covert channel the direct communication is limited, and the sender and receiver should be separated strictly by security policies. We propose to use the covert channels for the transmission of ack messages. There are two possible ways to achieve the aforesaid: (a) using the same shared resource employed in information encoding, and (b) using another shared resource to implement the acks. In a real scenario, the state of a shared resource can be changed by various noises, such as operations performed by other parties and system scheduling. Therefore, using a single shared resource to encode and synchronize will lead to a possible interference resulting in an out of order transmission. Considering the aforesaid situation, we choose to adopt different shared resource for the ack messages in our Two Channel Transmission Protocol (TCTP), discussed in algorithm 2.

The key idea in TCTP is to achieve the synchronization between the sender and receiver by implementing ack messages, and the separation of operations on encoding and synchronizing. The experiment results in Section 6 reveal that TCTP achieves better performance than the classic protocol in transmission accuracy with the same rate.

![Fig. 5. Transmission of TCTP.](image-url)
Fig. 6. Model of TCTP.

4.2 Modeling and Analysis

The model for TCTP is demonstrated in Fig. 6. The first step towards modeling the protocol is to identify the required types, P, and mapping. The types and the descriptions are shown in Table 3, and the mapping of P to types is depicted in Table 4. The tokens Ready, R_Ack and S_Ack control the flow of bits in the model of the system. The set of transitions is $T$ = $\{\text{Start, Get1bit, send0, send1, send_s_ack, read0, read1, send_r_ack, store1bit}\}$.

<table>
<thead>
<tr>
<th>Place</th>
<th>Mapping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rdy_St</td>
<td>P(Ready)</td>
<td>Holds the flag indicating sender and receiver are both ready.</td>
</tr>
<tr>
<td>Info_Tosend</td>
<td>P(Ready $\times$ S_Queue)</td>
<td>Holds the sending queue and ready flag.</td>
</tr>
<tr>
<td>Oneb_Tosend</td>
<td>P(R_Ack $\times$ Bit_Sending)</td>
<td>Holds the receiving ack and bit to be sent.</td>
</tr>
<tr>
<td>Shar_Res1</td>
<td>P(S_Queue $\times$ S0 $\times$ R0 $\times$ R1)</td>
<td>Holds the state of the shared resource 1.</td>
</tr>
<tr>
<td>Shar_Res2</td>
<td>P(Sr_Sack $\times$ Sr_Rack)</td>
<td>Holds the state of the shared resource 2.</td>
</tr>
<tr>
<td>Oneb_Sent</td>
<td>P(S_Ack $\times$ Bit_Sending)</td>
<td>Holds the sending ack and the bit that has been sent.</td>
</tr>
<tr>
<td>Oneb_Toread</td>
<td>P(S_Ack $\times$ Bit_Sending)</td>
<td>Holds the sending ack and bit to be read.</td>
</tr>
<tr>
<td>Oneb_Rec</td>
<td>P(R_Ack $\times$ Bit_Sending)</td>
<td>Holds the receiving ack and the bit that has been received.</td>
</tr>
<tr>
<td>Info_Store</td>
<td>P(R_Queue)</td>
<td>Holds the receiving queue.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready</td>
<td>A bool type for indicating whether the sender and receiver are ready for the transmission.</td>
</tr>
<tr>
<td>S_Queue</td>
<td>A bit array type for information to send.</td>
</tr>
<tr>
<td>Bit_Sending</td>
<td>A bit type for the bit in transmitting.</td>
</tr>
<tr>
<td>S0,S1,R0,R1</td>
<td>Bool types for indicating whether the state of shared resource1 representing sending 0, sending 1, reading 0 and reading 1 respectively.</td>
</tr>
<tr>
<td>R_Queue</td>
<td>A bit array type for storing the received bits.</td>
</tr>
<tr>
<td>Sr_Sack,</td>
<td>Bool types for indicating whether the state of shared resource2 representing sending S_Ack and R_Ack respectively.</td>
</tr>
<tr>
<td>Sr_Rack</td>
<td></td>
</tr>
</tbody>
</table>

The first transition is Start that has the same rules and constraints as was in (1). The next transition is Get1bit, which extract one bit from the sending-queue to the buffer of ready to send. The extracted bit is the one going to be sent by the channel. The transition is successfully fired only if: (a) the sending-queue is not empty, (b) the transmission has been started and (c) the sender receives the R_Ack from the receiver. Once the conditions are satisfied, the sending-queue in P(Info_Tosend) is updated to the one of popped out the first bit in the queue. Moreover, the value of Bit_Sending in P(Oneb_Tosend) is updated based on the value of the first bit in the sending-queue and the R_Ack is updated to false to indicate the buffer received the bit which is ready to send. The said transition is mapped to (8). After Get1bit is fired, one bit is extracted from the sending-queue to the ready to send buffer. The next two transitions are Send0 and Send1, which means the sender selects operations and changes the state of shared resource based on the bit to send. The two transitions are mapped to the rules (9) and (10), respectively.
After the above transitions are fired, the bit_sending token is extracted from $P$(Oneb_Tosend) and stored in $P$(Oneb_Sent), indicating one bit is sent but has not been read; what’s more, the R_Ack token in $P$(Oneb_Tosend) is set to 0 marking another ack needed from the receiver to continue the transmission and the S_Ack token in $P$(Oneb_Sent) is set to 1. The next transition send_s_ack which means sending an ack to notice the receiver one bit has been sent is mapped to (11):

$$\begin{align*}
R(\text{send_s_ack}) &= \forall b \in \text{Onb}_S \mid b[1] = \text{TRUE} \land \
& b[2] \neq \text{NULL} \land \
& \text{ackSh} \in \text{Sh}_R \mid \text{ackSh}[1] = \text{TRUE} \land \
& \text{Onb}_{SS'} = \text{Onb}_{SS} \cup \{(\text{ackSh}[1], b[2])\} \land \
& b[1] := \text{FALSE} \land b[2] := \text{NULL} \land \
& \text{Onb}_S' := \text{Onb}_S \cup \{(b[1], b[2])\} 
\end{align*}$$

(11)

The following transitions, Read0 and Read1, represent the receiver observing the state change of the shared resource and decoding the bit sent from the sender, after receiving the ack. Corresponding formulas are depicted in (12) and (13).

The bit_sending token in $P$(Oneb_Toread) is transferred to $P$(Oneb_Read), representing that the bit is read successfully. Moreover, the S_Ack token in $P$(Oneb_Toread) is set to false that enables the transmission of next bit. Furthermore, the R_Ack token in $P$(Oneb_Read) is also set to true. The receiver needs to send an ack to the sender after receiving the bit to inform the acceptance and the transmission start of next bit. The transition Send_r_ack performs the aforesaid and is mapped to (14).

$$\begin{align*}
R(\text{Read0}) &= \forall br \in \text{Onb}_R \mid br[1] = \text{TRUE} \land \
& br[2] = \text{NULL} \land \
& \text{ackSh} \in \text{Sh}_R \mid \text{ackSh}[1] = \text{TRUE} \land \
& \text{Onb}_{R'} := \text{Onb}_R \cup \{(br[1], br[2])\} \land \
& br[1] := \text{FALSE} \land br[2] := \text{NULL} \land \
& \text{Onb}_{R'} := \text{Onb}_R \cup \{(br[1], br[2])\} 
\end{align*}$$

(13)

$$\begin{align*}
R(\text{Read1}) &= \forall br \in \text{Onb}_R \mid br[1] = \text{TRUE} \land \
& br[2] = \text{NULL} \land \
& \text{ackSh} \in \text{Sh}_R \mid \text{ackSh}[1] = \text{TRUE} \land \
& \text{Onb}_{R'} := \text{Onb}_R \cup \{(br[1], br[2])\} \land \
& br[1] := \text{FALSE} \land br[2] := \text{NULL} \land \
& \text{Onb}_{R'} := \text{Onb}_R \cup \{(br[1], br[2])\} 
\end{align*}$$

(14)

The $R$ _Ack_ token in $P$(Oneb_Tosend) is set to true after this transition is fired. The last transition is store1bit which has the same meaning to Section 3.2 and is mapped to (7).

### 4.3 TCTP Summary

The TCTP uses two shared resources in one covert channel. Using the aforesaid design may increase the risk of being detected by IDS (Intrusion Detection System) or other covert channel protecting mechanism. Therefore, TCTP may have more noises or other interferences produced by the protecting mechanism. The issues of noises have always been affecting the transmission of covert channels. From the design perspective of TCTP, no methods are available to deal with strong noises that have also been proved by the experiments performed in Section 6. To overcome the issues of noises, we propose Self-Adaptive Protocol (SAP). The design and modeling of the SAP is discussed below.

### 5 Design, Modeling and Analysis of SAP

In this section, we present Self-Adaptive Protocol (SAP), which evolved from TCTP. Works on the design, modeling, and protocol analysis are presented in ensuing sections.

#### 5.1 Design

The term “Self-Adaptive” means that the protocol can identify interference and adjust itself to deliver the information correctly. Our principal thinking in the design of SAP lies in splitting all of the bits into several blocks, where each block contains few bits. The redundancy check is performed after the bits in the block are sent. If the check is failed, the complete block is resent. A sleeping time also maintained for each block that changes dynamically based on the results of the check performed above. The sleeping time increases if the check is failed, decreases otherwise based on the feedback received from the receiver.
Algorithm 3  Self-Adaptive Protocol

\textit{Shared\_Resource}[2]: Two separated shared resources. One used to transmit information, another used to send ack. Each has two states.

\begin{align*}
D_{\text{Send}}[N], D_{\text{Rec}}[N]: & \text{ N bits data to transmit and receive, respectively.} \\
L: & \text{ the length of each block; } \\
L_0: & \text{ current length of the sending block.} \\
T_0: & \text{ the initial sleeping time; } \\
T_1: & \text{ the increment of sleeping time.}
\end{align*}

**Sender Operations**

- **Step sending block:**
  - sleeping \(T_0\)
  - \(L_0 := 0; \) // initializing the value of \(L_0\)
- **Step sending one bit:**
  - if \(L_0 < L\) then
    - if \(D_{\text{Send}}[i] = 1\) then
      - set \(\text{Shared\_Resource}[0]\) to state 1;
    - else if \(D_{\text{Send}}[i] = 0\) then
      - set \(\text{Shared\_Resource}[0]\) to state 0;
    - end if;
    - set \(\text{Shared\_Resource}[1]\) to state 1;
    - busy loop to detect the state of \(\text{Shared\_Resource}[1]\);
      - if \(\text{state} = 0\) then
        - \(L_0 := L_0 + 1;\)
        - goto Step sending one bit;
      - else
        - continue looping;
    - end if;
    - // similar to TCTP;
  - else if \(L_0 = L\) then
    - calculating the redundant check bit rcB;
    - send rcB;
    - busy loop to detect the state of \(\text{Shared\_Resource}[1]\);
      - if \(\text{state} = 2\) then
        - \(T_0 := T_0 + T_1;\)
      - else
        - \(T_0 := T_0 + T_1;\)
      - goto Step sending one bit;
    - end if;
  - end if;

**Receiver Operations**

- **Step receiving block:**
  - \(L_0 := 0; \) // initializing the value of \(L_0\)
- **Step receiving one bit:**
  - if \(L_0 < L\) then
    - busy looping to detect the state of \(\text{Shared\_Resource}[1]\);
      - if \(\text{state} = 1\) then // sent_ack received:
        - detect the state of \(\text{Shared\_Resource}[0]\);
      - else
        - \(D_{\text{Rec}}[i] := 0; \) // get one 0-bit;
      - end if;
      - \(L_0 := L_0 + 1;\)
      - goto Step receiving one bit;
    - else
      - continue looping;
    - end if;
  - else \(L_0 = L\) then
    - receive rcB;
    - executing redundant check;
      - if check passed then
        - changing the state of \(\text{Shared\_Resource}[1]\) to 2;
      - else
        - \(D_{\text{Rec}}[i] := 1; \) // get one 1-bit;
      - end if;
      - \(L_0 := L_0 + 1;\)
      - goto Step receiving one bit;
    - else
      - continue looping;
    - end if;
  - end if;

and decreases if the check is passed. The sender and the receiver do nothing during the sleeping time. The sleeping time occupies most of the transmission time when intense noises are identified by the SAP. The process in SAP implies that the capacity of the channels varies based on the environment.

Fig. 7 illustrates an example of SAP using the simple parity check, where the information to be sent is split into five blocks and each block has 4 bits. The initial sleeping time is T, changes t at a time. As depicted in the Fig. 7, the first block fails in the check three times. Therefore, the block is resent three times and the sleeping time changes from T to T+t and T+2t. The next four blocks pass the check at one round, resulting in the sleeping time change of T-2t. The complete details and steps performed by SAP are described in Algorithm 3.

5.2 Modeling and Analysis

The model of SAP is depicted in Fig. 8. There are 11 places in the model. The names and mapping of P are shown in Table 5. The types used in the model are illustrated in Table 6.

The first transition \textit{Start} is the same as previous protocols and is mapped to the same rules depicted in formula (1). The next two transitions, Getblock_S and Getblock_F, depict the process of getting the bits blocks. The transition Getblock_S is successfully fired only if: (a) the previous block sent to the receiver passed the redundancy check and the check result is award by the sender according to the state change of shared resource 2 and (b) the sending-queue is not empty. Once the conditions are satisfied, the tokens in P(Block_S) is updated to the next bits block that has to be sent next and the sleeping time T decreases as T = T- t. Moreover, the value of Check_Bit is updated based on the token S_Block and the referenced clock Timer is set to 0 to restart the timing. The Check_S token in P(Shar_Res2) is also set to false to control the flow of bits. However, the operations performed by the transition Getblock_F are the opposite. The said two transitions are mapped to (15) and (16), respectively.
Table 5

<table>
<thead>
<tr>
<th>Place</th>
<th>Mapping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rdy_St</td>
<td>P(Ready)</td>
<td>Holds the flag indicating whether the sender and receiver are both ready.</td>
</tr>
<tr>
<td>Info_Tosend</td>
<td>P(Ready×S_Queue)</td>
<td>Holds the sending queue and ready flag.</td>
</tr>
<tr>
<td>Block_S</td>
<td>P(S_Block×L×T×1×Timer×Check_Bit)</td>
<td>Holds the bits block to send and its related attributes.</td>
</tr>
<tr>
<td>Oneb_Tosend</td>
<td>P(R_Ack×Bit_Sending)</td>
<td>Holds the receiving ack and bit to be sent.</td>
</tr>
<tr>
<td>Shar_Res1</td>
<td>P(S0×S1×R0×R1)</td>
<td>Holds the state of the shared resource 1.</td>
</tr>
<tr>
<td>Shar_Res2</td>
<td>P(Sr_Sack×Sr_Rack×Check_S×Check_F)</td>
<td>Holds the state of the shared resource 2.</td>
</tr>
<tr>
<td>Oneb_Sent</td>
<td>P(S_Ack×Bit_Sending)</td>
<td>Holds the sending ack and the bit that has been sent.</td>
</tr>
<tr>
<td>Oneb_Toread</td>
<td>P(S_Ack×Bit_Sending)</td>
<td>Holds the sending ack and bit to be read.</td>
</tr>
<tr>
<td>Oneb_Rec</td>
<td>P(R_Ack×Bit_Sending)</td>
<td>Holds the receiving ack and the bit that has been received.</td>
</tr>
<tr>
<td>Block_R</td>
<td>P(R_Block×L×Check_Bit)</td>
<td>Holds the bits block received and the check bit.</td>
</tr>
<tr>
<td>Info_Store</td>
<td>P(R_Queue)</td>
<td>Holds the receiving queue.</td>
</tr>
</tbody>
</table>

Table 6

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready</td>
<td>A bool type for indicating whether the sender and receiver are ready for the transmission.</td>
</tr>
<tr>
<td>S_Queue</td>
<td>A bit array type for information to send.</td>
</tr>
<tr>
<td>Bit_Sending</td>
<td>A bit type for the bit in transmitting.</td>
</tr>
<tr>
<td>S0,S1,R0,R1</td>
<td>Bool types for indicating whether the state of shared resource 1 representing sending 0, sending 1, reading 0 and reading 1 respectively.</td>
</tr>
<tr>
<td>R_Queue</td>
<td>A bit array type for storing the received bits.</td>
</tr>
<tr>
<td>Sr_Sack, Sr_Rack</td>
<td>Bool types for indicating whether the state of shared resource 2 representing sending S_Ack and R_Ack respectively.</td>
</tr>
<tr>
<td>Check_S, Check_F</td>
<td>Bool types for indicating whether the state of shared resource 2 representing check passed or failed.</td>
</tr>
<tr>
<td>S_Block</td>
<td>A bit array type for the bits to send block.</td>
</tr>
<tr>
<td>R_Block</td>
<td>A bit array type for the bits received block.</td>
</tr>
<tr>
<td>L</td>
<td>An integer type for the max length of the block.</td>
</tr>
<tr>
<td>T</td>
<td>An integer type for the sleeping time.</td>
</tr>
<tr>
<td>Timer</td>
<td>An integer type for a referenced clock.</td>
</tr>
<tr>
<td>Check_Bit</td>
<td>A bit type for the check bit.</td>
</tr>
</tbody>
</table>
The next transition Get1bit is very similar to the one in TCTP, only difference is the bit to send comes from the S_block instead of the sending queue in TCTP. The transitions, send0, send1, send_s_ack, read0, read1, send_r_ack, and store1bit have the same rules and functionalities as discussed in TCTP. The remaining two transitions, Storeblock_S and Storeblock_F, illustrate the process of storing the block. The redundancy check is also enforced by the aforesaid two transitions. Storeblock_S depicts the successful storage of the block that requires certain constraints to be satisfied, such as, the length of R_Block must be equal to the max length L and the redundancy check is passed by the block by verifying the value of R_Block and Check_Bit. After the transition Storeblock_S is fired, Check_S in P(Shar_Res2) is set to true and the R_Block is stored in the R_Queue. The R_Block and Check_Bit are cleared for next block. The transition Storeblock_F depicts a scenario when any of the constraints discussed in Storeblock_S are violated. The two transitions are mapped to (17) and (18), respectively.

\[
R(\text{Getblock}_S) = \forall vb \in \text{Inf}_ToS \mid vb[1] = \text{TRUE} \land vb[2] \neq \text{NULL} \land \forall b \in \text{Sh}_R | b[3] = \text{TRUE} \land \forall BS \in \text{Inf}_ToSS | \text{GetCheck}(BS[b - 1]) = \text{TRUE} \land BS[3] := BS[3] - BS[4] \land \text{Inf}_\text{ToSS} = \text{Inf}_\text{ToSS} \cup (BS[1], BS[2], BS[3], BS[4]), BS[5], BS[6]) \land b[3] := \text{FALSE} \land Sh_\text{R2} = Sh_\text{R2} \cup ((b[1], b[2], b[3], b[4])) \]

\[
R(\text{Getblock}_F) = \forall vb \in \text{Inf}_ToS \mid vb[1] = \text{FALSE} \land vb[2] = \text{NULL} \land \forall b \in \text{Sh}_R | b[4] = \text{TRUE} \land 
\]

6 Experiments and Evaluations

In this section, we implement and evaluate the proposed protocols in Linux, Xen, and Fiasco OC, to demonstrate the effectiveness and generality of the proposed protocols. The scenarios are constructed based on the models. Moreover, a comparison of experimental transmission rate and estimated rate is presented to prove the model’s ability on describing covert channels.

6.1 Evaluation Criteria

Capacity [9] [10] [11] is the amount of information transferred by the channel per unit time (bits per second). Formal and non-formal methods are presented to calculate the capacity by Millen and Tsai. The accurate capacity can be obtained by experiment. The capacity (C) in our work is calculated as follows:

\[
C = \frac{N(t)}{t},
\]

where \(N(t)\) is the amount of the information transmitted in total time \(t\).

Accuracy is also a metric to evaluate the channel performance. If the covert message cannot be decoded correctly, the obtained information is valueless. The channel accuracy is measured as the percentage of correctly received bits. As Cabuk et al. [5], the edit distance [22] is adopted to measure the accuracy, which is the minimum distance between two strings.

6.2 Experiment Assumptions

Two scenarios are considered in our experiments. The first scenario simulates a normal environment without restricting mechanism, where noises come from the third party who has ten activities and one can cause the change of shared resource. Each activity is launched randomly. The later simulates an environment with mitigating mechanism where noises are produced intentionally,
which means the shared resource is always changed when the mechanism is running. However, as the mechanism degrades the system performance, it cannot keep running all the time. We assume it runs according to a random timing sequence.

6.3 Evaluations

The TCTP aims at reliable, efficient, and deterministic covert information transmission in OSes. Constructing a TCTP needs to specify the shared resources and related operations. In Linux, we use the last_pid to encode and temporary file to synchronize [16]. In Xen, the shared memory [12] acts as two resources: the time intervals between two normal data are used to encode and special words embedded in the data indicate synchronization; in Fiasco.OC, two normal IPCs are used to represent S_ACK and R_ACK respectively and the time intervals between them are used to encode.

The experiment results in Fig. 9 reveal that the capacity of TCTP and the classic covert channel protocol are almost the same in Linux and Fiasco.OC; TCTP decreases fewer than 5% in Xen platform. The decrease is caused by the synchronization operation in TCTP. Fig. 10 depicts the decode error rate of TCTP and the classic covert channel protocol in normal and mitigating environment. It clearly shows that TCTP performs much better in all of the scenarios. This suggests that the synchronization and shared resource separation designed in TCTP increase transmission accuracy effectively. The Fig. 10 also reveals that some characters in Xen can increase the decode error rate of covert channels; and although microkernels are considered more secure in some properties, they suffer more in covert channel attacks when compared to the other OSes.

The decode error rate of TCTP in scenario 2 reaches more than 50%, which means 50% of messages transferred cannot be used. In such scenarios, SAP is preferred. As shown in Fig. 11, the accuracy of SAP with strong noises in the three OSes reaches 90%. A statistic of average decode error rate on the TCTP and SAP is shown in Fig. 12. The SAP achieves much lower decode error rate in all OSes. Fig. 11 and Fig. 12 demonstrate the redundancy check and segment retransmission technologies in SAP can ensure the correctness of information transferred in a covert channel. Fig. 12 also reveals that Xen gets higher error rate than the other two platforms in both of the TCTP and SAP.

6.4 Capacity Estimation

One important application of covert channel models is to estimate the capacity of the channel. We introduce the simulation of capacity estimation of TCTP in Linux by using the HLPN model.

Fig. 10. Decode error rate of TCTP and Classic; the lower two lines represent a normal environment; the upper two lines represent an environment with mitigating mechanism.

Fig. 11. Decode error rate of SAP in mitigating environment.

Fig. 12. Average decode error rate of TCTP and SAP in Linux, Xen, and Fiasco.OC in mitigating environment.
Recall that there were nine transitions \( T \) = \( \{ \text{Start, Get1bit, send0, send1, send_s_ack, read0, read1, send_r_ack, store1bit} \} \) in TCTP HLPN model. Among them, \( \text{Start, send_s_ack, and send_r_ack} \) are mapped to \( \text{create}() \), a system command to create temporary files; \( \text{send0, send1, read0, and read1} \) are mapped to \( \text{fork}() \), a system command to change the value of \( \text{last_pid} \). \( \text{Get1bit and store1bit} \) are operations to write the memory, which can be ignored when compared with the other two operations. We use \( T1 \) to \( T9 \) to denote the time of each transition. Assume that there are \( N \) bits to transfer, \( m \) bits 0 and \( N-m \) bits 1. The capacity of TCTP can be estimated by following formula:

\[
C = \frac{N}{T1 + N \times (T2 + T3 + T8) + m \times (T3 + T6) + (N - m) \times (T4 + T7)}
\]

(19)

In our implementation in Linux, \( T1=5=8 = 1 \text{us} \) (the time of one single \( \text{create}() \)), \( T3=T6=19 \text{us} \) (the time of two successive \( \text{fork}() \)), \( T4=T5=10 \text{us} \) (the time of one single \( \text{fork}() \)). We have:

\[
T1 + N \times (T2 + T5 + T8) \approx 2N(\text{us})
\]

\[
m \times (T3 + T6) \approx 38 \times m(\text{us})
\]

\[
(N - m) \times (T4 + T7) \approx 10N - 10m(\text{us})
\]

Finally we get:

\[
C = \frac{N}{12N + 28m}
\]

(20)

The distribution of bit 1 is equal to bit 0 in large scale statics, so we have:

\[
m = 0.5 \times N
\]

So

\[
C_{estim} = 38.4 \text{ kbits}
\]

As shown in Fig. 9, the max capacity achieved in our experiment is 35kbits, which is very close to the estimated value; the estimation error is 9.7%.

7 RELATED WORK

In 1973, Lampson [4] discussed the ways in which information can be transferred between programs (i.e., legitimate, storage, and covert channels) and defined covert channels as those not intended for information transfer at all, such as the program’s effect on the system load. After the innovative work, covert channels became one of the hottest topics in academia and were discovered in all kinds of computer systems [8], [18], [26] (i.e., desktop operating systems, database, network, cloud and mobile computing).

The subject of this paper is the designing and modeling of covert channels. The most popular work in the designing of covert channel is the L2 cache covert channel attack in EC2 [6] performed by Ristenpart et al. As a follow up, Okamura et al. designed and evaluated a new attack that uses the load of a shared CPU to encode information [21]. On the other hand, Wu et al. identified a new covert channel termed sharing memory covert timing channel (SMCTC) from Xen source code [20]. Xu et al. re-explored L2 cache channel and achieved higher bit rate than previous works [17].

As for covert channel modeling, Millen firstly used the technique of finite state machines to model covert channels that use individual channel variables [9]. Tsai and Gligor [11] improved the finite-state into a Markov model to simulate the use of covert storage channels and to compute their maximum bandwidths under different system loads and program behaviors. Recently, a four-state machine [12] was proposed to depict the scenario with transmission errors.

8 CONCLUSIONS AND FUTURE WORK

Covert channel analysis is a classic topic in information security. Constructing real covert channel communications in new computing platforms is the first step for researchers to learn the threat of covert channels. In this paper, we studied the design and modeling of the covert channel protocols. Two covert channel protocols called TCTP and SAP aiming to provide reliable transmission are proposed. The TCTP avoids mutual inferences in encoding and synchronization operations and the SAP uses sleeping time and redundancy check to ensure correct transmission in an environment where strong noise exists. We used HLNP to model and analyze the three protocols (classic covert protocol, TCTP, and SAP). The HLNP has greater ability on describing covert channel and is easy to extend to depict new covert protocols. Our experiments proved that the TCTP and SAP can achieve high accuracy with stable capacity, and the models perform much better on capacity evaluation. In future, we are planning to use model checking approach to detect covert channels and analyze the decode error rate and channel restrictions by information flow analysis based on the model.

ACKNOWLEDGMENTS

This work is jointly supported by the National Science and Technology Major Project under Grant 2012ZX01039-004, the National Natural Science Foundation of China under Grant 61170072, 61305057, 61305054, the Knowledge Innovation Key Directional Program of Chinese Academy of Sciences under Grant KGX2-YW-174, KGX2-YW-125, and the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDA06010600.

REFERENCES


This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2458862, IEEE Transactions on Computers

Yuqi Lin is currently a PhD student at the Institute of Software, Chinese Academy of Sciences. His primary research interests include covert channel analysis, virtualization techniques, cloud computing and system security.

Saif U. R. Malik is an assistant professor at the COMSATS Institute of Information Technology Islamabad, Pakistan. His research interests include Formal Methods (including modeling, analysis, and verification) and its application in large-scale computing systems.

Kashif Bilal is an assistant professor at the COMSATS Institute of Information Technology Abbottabad, Pakistan. His research interests include energy efficient high speed networks and robustness.

Qiusong Yang is Research Professor at the Institute of Software, Chinese Academy of Sciences. His research interests include operating systems, model checking of concurrent systems and software engineering.

Yongji Wang is Research Professor at the Institute of Software, Chinese Academy of Sciences. His work has appeared in over 150 publications.

Samee U. Khan is associate professor at the North Dakota State University. His work has appeared in over 275 publications. He is associate editor of: IEEE Access, IEEE Cloud Computing, IEEE Communications Surveys and Tutorials, and IEEE IT Pro. For more information, see: http://sameekhan.org/