A Cheat Controlled Protocol for Centralized Online Multiplayer Games

Bei Di Chen
School of Computer Science
McGill University
Montreal, Canada
bei.di.chen@mail.mcgill.ca

Muthucumaru Maheswaran
School of Computer Science
McGill University
Montreal, Canada
maheswar@cs.mcgill.ca

ABSTRACT
Ordering of command messages from the clients at the game servers is an important issue that impacts fairness, response times, and smoothness of the game play. Recently, protocols based on "reaction" times were proposed to order the command messages. This paper presents a protocol that can be used to control cheating in reaction time based message ordering schemes. We examine the performance of the proposed protocol by emulating wide-area game play scenarios on the Planet-Lab. The results from the experiments indicate that the proposed protocol is able to dramatically reduce the cheating opportunities that exist for the clients.

Categories and Subject Descriptors
C.2 COMPUTER-COMMUNICATION NETWORKS
C.2.4 Distributed Systems
Distributed applications

General Terms
Algorithms, Performance

Keywords
Cheat prevention, multiplayer online games, time cheating

1. INTRODUCTION
Online multiplayer games have tremendously increased in popularity over the last several years. Game cheating is an issue that will be faced by game servers belonging to all six categories [5, 7, 8]. Successful prevention or controlling strategies are essential to retain the trust of the clients who use the gaming services. In this paper, we examine one particular category of cheating known as time-cheating, which gives a cheat player an unfair advantage by allowing him to see into the future, having additional time to react to honest players' actions.

This paper considers a centralized game model, which is the most popularly deployed game model today. One advantage of this model is that it offers a single point for game coordination. On the negative side, it does create a bottleneck of processing and communication when the number of players increases in the online virtual world. We consider the centralized game model as the first step. Extending the study to a distributed game model will be the focus of a future study.

One important operation performed by a game server is the ordering of command messages that arrive from the clients. The ordering should be performed while maintaining fairness and low response times of game play. The fairness requirement is difficult to implement because different clients have different network latencies and processing abilities. One message ordering scheme that provides fairness in this respect is [3] that orders messages by the responded game states and the reaction times of the clients. One concern with this protocol is the opportunity for cheating by malicious players that can present false information. This paper presents a cheat controlling scheme that can improve the above fair-order message deliver protocol such that the opportunities for time-based cheating are severely limited.

There are other works about cheat-proof multiplayer game protocol on P2P platform. The lockstep and sliding pipeline protocols, proposed in [1, 2] both rely on message commitments to prevent time-cheating.

Section II presents the terminology used in this paper and the general assumptions. Section III presents the fair-ordered message deliver model that is used as the basis to develop our protocol. Section IV examines the CCP in detail. Section V provides the experimental results and a discussion of the results.

2. TERMINOLOGY AND ASSUMPTIONS
The set of state information needed to describe the game at any time is referred to as the *game state*. Under the *client-server* architecture, the *game state* is maintained by the server, which computes game state based on *command messages* from clients. The server informs the clients of the current game state by sending *update messages*, which are rendered at the client side and referred to as the *frames*. Finally, we define an online game as *fair* if the game state as perceived by every player is consistent with every other player's expectations, including the server, as defined by game rules.

We assume that both the involved application software and protocols are available to players such that cheating players are able to read, insert, modify, and block messages transmitted in the game protocols. Our goal is to develop a scheme that detects malicious activity and limits the damage by selectively penalizing the attackers. Protecting against potential attacks at transport-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCOMM'04 Workshops, Aug. 30 & Sept. 3, 2004, Portland, OR, USA.
Copyright 2004 ACM 1-58113-942-X/04/0008…$5.00.

layer protocols, network-layer protocols, and operating system operations are beyond the scope of this work.

3. FAIR-ORDERED MESSAGE EXCHANGE PROTOCOL

In [3], a fair-ordered message exchange protocol (FMEP) that does not depend on the actual network latencies from the server to the client was developed. Our protocol builds on this model to provide a cheat controlled environment where cheating opportunities for the clients do not depend on the network latencies.

3.1 Message Exchange Scheme

Let us now examine the message exchanges between the server and the players and their effect on the state of the game. Figure 1 shows a timing diagram of an instance of message exchange between the server and player P_j. Let U_i denote the server's local time at which the server sends an update message UM_i. Player P_j receives UM_i at its local time R^i_j. After receiving an update message, the player generates a command message as a response. We refer to the duration between reception of an update message and transmission of the command message by a player as *reaction time*. CM^i_{jk} denotes the kth command message sent by player j at its local time C^i_{jk} after acting on UM_i from the server. Let $\delta^i_{jk} = C^i_{jk} - R^i_j$ denote the corresponding reaction time.

3.2 State and State Transition

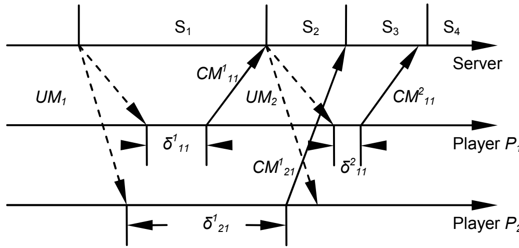

Figure 1. Message exchange scheme.

When the state of the game is S_1, UM_1 is sent by the server and received by both players. Players may receive UM_1 at different instants of local time (that is, $R^1_1 \neq R^1_2$) due to variability in network conditions. P_1 sends a command message CM^1_{11}, which is received at the server with reaction time δ^1_{11}. P_2 also sends a command message CM^1_{21} with reaction time δ^1_{21}. After the server receives both action messages, and the inspection of the reaction times reveals that player P_1 has acted on state S_1 of the game *quicker* than player P_2. Therefore, the command message CM^1_{11} is delivered for processing before CM^1_{21} regardless of the relative arrival order of CM^1_{11} and CM^1_{21}. Moreover, even CM^2_{11} has a smaller reaction time than CM^1_{21}, CM^1_{21} is delivered before CM^2_{11} because CM^1_{21} corresponds to a prior state. Note that Figure 1 depicts the delivery instances of command messages according to fair order which is that command messages respecting to a same update message must be delivered in the increasing order of player's reaction time and command messages related to a prior update message must be delivered before those related to a late update message.

4. CHEAT CONTROLLED PROTOCOL

4.1 Time Cheating

In the example in Figure 1, if P_2 was a malicious player, he could intentionally alter the value of δ^1_{21}, of CM^1_{21}, or tag CM^1_{21} referring to the previous update message such that CM^1_{21} would be delivered for processing before the command message CM^1_{11} from P_1, from which he benefits unfair situation. In other words, through time cheating a player can obtain more time to react to a game world event than an honest player. This kind of time-cheats is the problem this paper is going to address.

4.2 Cheat Controlling

The difference between the time the server sends out UM_i and the time the corresponding CM^i_{jk} arrives includes the *Actual Round Trip Time* ($ARTT^i_{jk}$) between the server and the player P_j, the *Actual Update Message Processing Time* (AS^i_j) at the player and the reaction time δ^i_{jk}. Please note that we assume that the time to issue a command message is negligible. $ARTT^i_{jk}$ associated with each command message varies with time depending on the network condition. The server estimates the *Estimated Current Round Trip Time* ($ERTT_j$) for the player P_j for a given time instance. In addition, the server tolerates a certain variation between $ERTT_j$ and $ARTT^i_{jk}$, called the *Round Trip Time Tolerance* ($RTTT_j$). Similarly, the time to process an update message by the player depends on the work load on the player's machine. Additionally, the server estimates the *Estimated Update Message Processing Time* (ES^i_j) for P_j for UM_i. The *Update Message Processing Time Tolerance* (ST^i_j) specifies the acceptable variance level by the server.

Therefore, for given CM^i_{jk} and δ^i_{jk}, the server independently computes a *Proposed Arrival Time* (*PAT*) for command message CM^i_{jk} as follows:

$$PAT^i_{jk} = U_i + \delta^i_{jk} + ERTT_j + ES^i_j + RTTT_j + ST^i_j. \quad (1)$$

Let AAT^i_{jk} denote the actual arrival time of CM^i_{jk} and be expressed as below, where $A\delta^i_{jk}$ is the actual reaction time, $ARTT_j$ is the actual RTT and AS^i_j is the actual update message processing time.

$$AAT^i_{jk} = U_i + A\delta^i_{jk} + ARTT_j + AS^i_j. \quad (2)$$

If $AAT^i_{jk} \leq PAT^i_{jk}$, we say that P_j is honest, so $A\delta^i_{jk}$ is equal to claimed δ^i_{jk} and accepted by the server. On the other hand, if $AAT^i_{jk} > PAT^i_{jk}$, there can be several causes for the delay: network congestion between the server and the client where player P_j is located (i.e., $ARTT_j > ERTT_j + RTTT_j$), computational congestion at the client machine (i.e., $AS^i_j > ES^i_j + ST^i_j$), or a cheating attempt by player P_j (i.e., $A\delta^i_{jk} > \delta^i_{jk}$). The rule used in the CCP to distinguish between the first two causes and the third cause is simple: if there is no network congestion observed by the server, a delayed message is a cheating message which will be penalized.

4.3 Network and Computational Latency Measurement

To measure the RTT between the server and a client the server does the following. First of all, it examines the incoming messages for a fixed *Message Monitoring Interval* (MMI) for the percentage of late messages. When this percentage rises above a threshold given by the *Ping Threshold* (PT) and there is no outstanding ping in the wire for that client, we ping the client. The

latest ping operation is used to update the $ERTT_j$ value. In addition, in order to be posted with the current network conditions, if P_j doesn't have an outstanding ping, the server decreases the current value of $ERTT_j$ by a constant rate, the *Declining Rate* (*DR*), with time, which correspondingly decreases the *PAT* value. Thus, the server will be triggered to ping P_j regularly.

Furthermore, the lifted $ERTT_j$ due to network congestion needs to be brought down when the network recovers. Only depending $ERTT_j$ to decline by itself is not efficient. Hence, the server does the following. It keeps track of two round trip times during the game play of P_j: *smallest RTT* ($SRTT_j$) and *last ping RTT* ($LRTT_j$). Every time after P_j is pinged, a watermark is set based at ($PAT - \alpha \cdot LRTT_j$) where α is a predefined percentage. When the actual arrival time of a message is less than the watermark it definitely indicates a network recovery from congestion. Based on this observation, the NLM aggressively makes the $ERTT_j$ equal to $SRTT_j$ if $LRTT_j > SRTT_j$, otherwise ping the player to get the current network latency.

The computational latency depends purely on the client's machine resources. Measuring the precise resource availabilities at the client side which is completely controlled by the player is not possible because a cheating player can influence the outcome of the measurement process. We handle this problem by providing a maximum limit on the processing time at the client-side machine. The player is expected to use a machine that is faster enough to complete the processing within this limit. We call this limit as the *Estimated General Update Message Processing Time* (*EGS*). With the maximum limit of *EGS* for the client side processing times, we can argue that a malicious player's cheating chances are limited to the laxity in the *EGS* value. We can use this maximum value to simplify Equations (1) and (2) as below.

$$PAT^i_{jk} = U_i + \delta^i_{jk} + ERTT_j + RTTT_j + EGS. \quad (3)$$

$$AAT^i_{jk} = U_i + A\delta^i_{jk} + ARTT_j + EGS. \quad (4)$$

4.4 Pseudocode

```
Initialization {
    SRTTj = LRTTj = ERTTj = initial Pinged RTT;
}
Upon receiving a command message {
    if AAT^i_jk ≤ PAT^i_jk then { //Considered as an honest message;
        if (AAT^i_jk < PAT^i_jk - α·LRTTj) then
            if (LRTT^i_jk > SRTTj) then {
                ERTTj = SRTTj;
                LRTTj = SRTTj;
            }
            else
                Ping Pj;
    }
    else{ //Considered as a cheating message;
        if (For Pj, Percentage of late messages in MMI > PT
        AND Pj is not being pinged) then
            Ping Pj;
    }
}
Upon receiving a ping result (PRj) {//PRj:Ping Result
    ERTTj = PRj;
    if (SRTTj > PRj) then
        SRTTj = PRj;
    LRTTj = PRj;
}
Upon every execution cycle {
    For j∈Γ
        if Pj is not being pinged then
            decrement ERTTj by DR;
}
```

4.5 Example

Figure 2 depicts a sample game run for an honest player P_j. Each point represents a command message arrival at the server. For simplicity, command messages arrive every 250 milliseconds. As P_j is honest (i.e., $A\delta^i_{jk}=\delta^i_{jk}$), based on Equation (3) & (4) the difference between AAT^i_{jk} and PAT^i_{jk} equals the difference between $ARTT_j$ and ($ERTT_j$+$RTTT_j$). Therefore, in the figure PAT^i_{jk} is represented by ($ERTT_j$+$RTTT_j$), the blue line, and AAT^i_{jk} is represented by $ARTT_j$, the pink line. The red little triangles sitting on the x-axis denote the ping times by the server. And the black solid line is the watermark. First of all, the server pings P_j at the beginning to gather the initial $ERTT_j$. Afterwards, $ERTT_j$ declines for every execution cycle. The server will not ping the player unless the percentage of late messages in MMI is greater than PT. The sharp spike which happens at time 500 doesn't trigger the ping because it is smoothed out as noise by the server. As the blue line goes down due to the declining $ERTT_j$, before the pink line touches the watermark, PT is finally exceeded at time 3000 to cause the second ping which in term updates $ERTT_j$ and sets $LRTT_j$ respectively. As usual, the $ERTT_j$ value declines after the second ping. At time 4000, the network congestion sets in and the pink line increases dramatically.

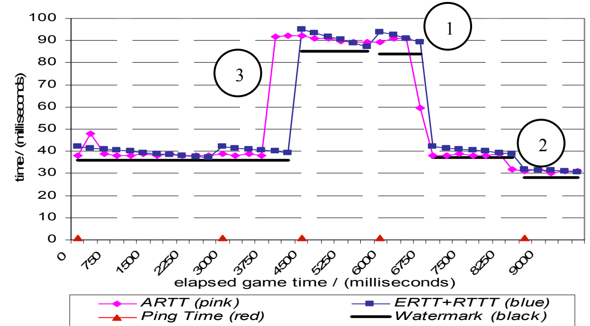

Figure 2. Example of an honest player in network congestion.

The server doesn't ping P_j until PT is exceeded. The new $ERTT_j$ increases the blue line to adapt to the current network condition and the watermark is also set accordingly, as well as $LRTT_j$. As the declining rate is set at 1% of $ERTT_j$, with higher $ERTT_j$ values the blue line declines faster. At time 6000, the fourth ping is triggered before the pink line hits the watermark. After the fourth ping, at time 6750 the network congestion disappears such that the pink line drops below the watermark, which makes the server believe the resolution of the network traffic congestion and brings $ERTT_j$ to be $SRTT_j$ and set $LPTT_j$ to be $SRTT_j$. A similar event occurs at time 8750. Since $SRTT_j$ is the same as $LRTT_j$, the server

pings the player to probe for the current network latency. The new pinged result is used to set $ERTT_j$, $SRTT_j$ and $LRTT_j$.

On the figure where the pink line is greater than the blue line is the moment a malicious player can cheat. The maximum cheating time has to be equal or smaller than the difference between the two lines, otherwise the cheating message will be detected It's easy to find out that the biggest difference happens immediately after the network congestion is gone, labeled 1 and 2 on the figure. On the other hand, where the pink line is smaller than the blue line is when the honest messages are miss-detected as cheating messages. The most apparent interval is labeled 3. We refer those labeled intervals as difficult intervals. Hence a bigger declining rate during network traffic time will shorten the cheating interval due to more frequently pinging; and the smaller values of MMI and PT will make the server more sensitive to network condition changes to reduce the number of miss-detected honest messages. However, both of them will cause the network to be loaded with more ping packets (especially critical during a congestion time). It is the tradeoff game designers have to decide to meet their game requirements.

Please note the way the CPP probes network status is more efficient compared to a regular ping scheme (pinging in a constant time interval) because under the CPP the server pings a client loosely when no traffic is involved and tightly otherwise.

5. EXPERIMENTS AND PERFORMANCE ANALYSIS

5.1 Testbed & Game Simulation

We implemented a game server for a shooting game and robot players to test the CCP proposed in this paper and deployed the game server and robot players on the PlanetLab [4], a wide-area experimentation platform. Due to source code unavailability of commercial online games, we choose XPilot [6], a popular open source first person shooting game in 2-dimensional space, as our testing game model. The XPilot server was modified to output a descriptive game trace file. The automated players (robots) and the server in the CCP emulation were driven by the trace file to mimic the real game interactions. Therefore, the results presented here are based on experiments that are realistic. Although the results are not generic, they are representative of the class of game interactions for which the CCP should be effective. To obtain the traces, we hosted the instrumented XPilot server running at 14 frames per second on a machine at the Advanced Networking Research Lab of McGill University. During the hosting period, there were 56 game sessions being played on our server. As we wanted the game simulation to last 10 minutes, for those players who played fewer than 10 minutes their trace files were repeated. For purpose of this study, we implemented a computational latency monitor that resided at each remote site. The data sensed by this monitor was used for analysis only.

5.2 Key Parameters

For the experiments reported here the following default parameters should be assumed. The declining rate (*DR*) was set to 1.0 percent of the $ERTT_j$ per frame. For the watermark α was set at 10.0 percent of $LRTT_j$. MMI was set at 1 second. PT was set at 40 percent. And *EGS* was set at 3 milliseconds. Please note *RTTT* is a global value which generally applies to all the automated players. In addition, in our experiments, the meaning of a cheating robot with the *cheating time* (*CT*) is that a robot always reports its reaction time in the value of *CT* milliseconds smaller than the actual reaction time for every command message. If the actual reaction time for UM_i is smaller than *CT*, the cheater will calculate the dishonest reaction time referring to the previous update message UM_{i-1} to ensure *CT* is taken.

5.3 Experiments and Results

Based on the trace file from XPilot, we had 56 robots and 1 central server running on 57 PlanetLab nodes to emulate game play scenarios. The 56 robots were separated into two types: non-cheating and cheating robots. Based on the network latency and processing power, the robots are divided into three groups for individual performance analysis. The first group was composed of 45 stable network latency and processing power robots. The second group was composed of 5 stable network latency but unsteady processing power robots. The third group was composed of 6 fluctuating network latency but stable processing power robots. Fortunately both unsteady processing power and fluctuating robots didn't appear in our experiments.

5.3.1 Results with stable network latency and processing power robots

This group composed of 45 stable network latency and processing power robots of which 35 were non-cheating and 10 were cheating. Figure 3 shows the result of the success rate of cheating message detection over *RTTT*, ranging from 1 to 12, and *CT*, ranging from 2 to 11. The result clearly shows that the success rate of cheating message detection is very satisfying and discloses that the success rate of cheating message detection is proportional to (*CT* / *RTTT*).

5.3.2 Results on stable network latency but unsteady processing power robots

The robots in this group all suffered from the lack of processing resource problem, which caused the actual update message processing time (*AS*) to increase way beyond our proposed *EGS* value. There are 5 robots in this group with 3 non-cheatings and 2 cheatings. Let us look at the node of `nodea.howard.edu`, where a non-cheating robot was running with *RTTT* equal to 7 milliseconds. The actual consumed CPU time of each update message is shown in Figure 4. By observing the graph, apparently there was a process periodically executing task on `nodea.howard.edu` which delayed the update message computation. The consequence of unpredictable increase of update message processing time is the growth of AAT^i_{jk}, which in turn will result in a dropping non-cheating message detection success rate because the server just simply considers these delayed command messages, due to $AAT^i_{jk} > PAT^i_{jk}$, as cheating messages in the absence of observed network latency changes.

5.3.3 Results with fluctuating network latency but stable processing power robots

Let us move on to the results with fluctuating network latency robots. There are a total of 6 robots falling into this group, 2 non-cheating and 4 cheating. Figure 5 exhibits that the success rate of cheating message detection approaches to 93% while the *CT* increases and *RTTT* decreases. There are 7% cheating messages

undetected at 10 milliseconds *CT* or beyond due to the difficult intervals. It's not hard to find out if the network latency was less volatile the difficult intervals would be smaller and result in higher success rates. In order to show this, we calculate the sample variance S^2 for every cheating robot in this group and plot them separately using different lines in the Figure 6. By observing the figure, the robot with the smaller network latency deviation (*S*) has the better success rate at most of the time.

6. CONCLUSION

In this paper, we presented a cheat controlled protocol that can work with fair-ordered message delivery scheme for online multiplayer games. Currently, our study focuses on centralized multiplayer games. This paper completely defines the cheat controlled protocol and presents the results obtained from a deployment of the protocol on a wide-area, realistic, game playing scenario on the PlanetLab.

From the experimental results, we can conclude that the proposed protocol is able to adapt to changing network conditions and still retain high success rates in detecting the cheating conditions. One challenge is when the client machine itself is unpredictably loaded with highly varying or large processing times for the game rendering messages. We expect players to choose machines with sufficient idle resources as gaming clients. Further, the experimental results indicate that the protocol is able to achieve high success rates for cheating times larger than 7 milliseconds (i.e., the proposed protocol is able to detect with high success when a player is trying to cheat by more than 7 milliseconds).

7. REFERENCES

[1] N. E. Baughman and B. N. Levine, "Cheat-proof playout for centralized and distributed online games," *IEEE INFOCOM,* 2001, pp. 104-113.

[2] E. Cronin, B. Filstrup and S. Jamin, "Cheat-Proofing Dead Reckoned Multiplayer Games (Extended Abstract)," *ADCOG* 2003, Jan. 2003.

[3] K. Guo, S. Mukherjee, S. Rangarajan and S. Paul, "A Fair Message Exchange Framework for Distributed Multi-Player Games," *NetGames'2003*, May 2003.

[4] The PlanetLab home page, 2004

http://www.planet-lab.org/.

[5] M. Pritchard, "How to hurt the hackers: The scoop on Internet cheating and how you can combat it," http://www.gamasutra.com/features/20000724/pritchard 01.htm.

[6] The XPilot home page, 2004, http://www.xpilot.org/.

[7] J. Yan and H. Choi, "Security Issues in Online Games," *The Electronic Library: international journal for the application of technology in information environments, Vol. 20 No.2*, 2002.

[8] J. J. Yan and H. Choi, "Security Issues in Online Games", *ACSAC'03*, 2003.

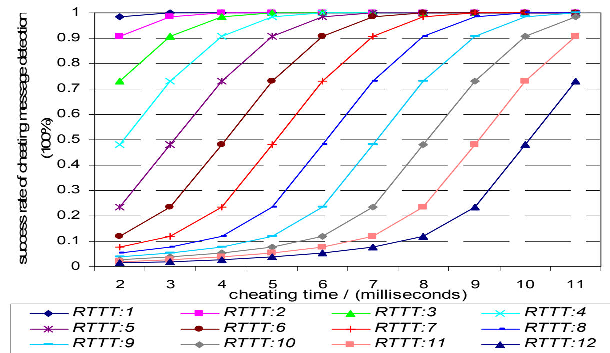

Figure 3. Success rate over *CT* and *RTTT*.

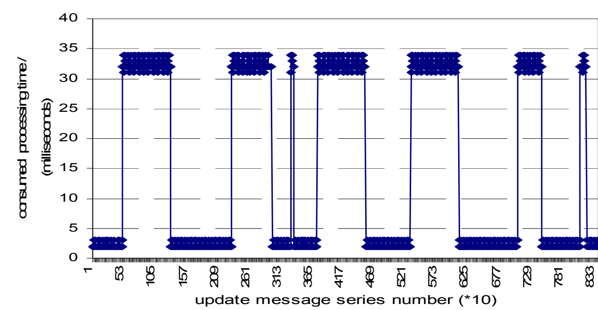

Figure 4. Consumed processing time of update message `@nodea.howard.edu`.

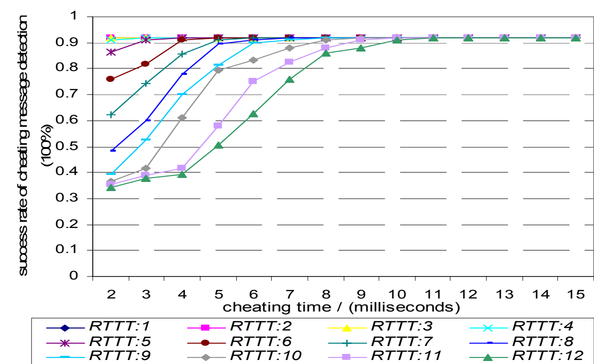

Figure 5. Success rate over *CT* and *RTTT*.

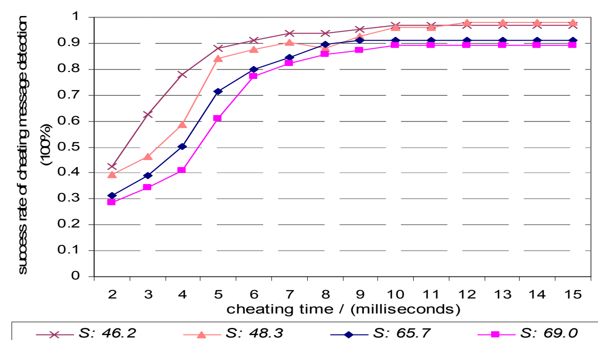

Figure 6. Success rate over *CT* and *S*