Journal of Universal Computer Science, vol. 11, no. 1 (2005), 83-103
submitted: 15/7/04, accepted: 6/12/04, appeared: 28/1/05 © J.UCS

ProtoMon: Embedded Monitors for Cryptographic Protocol Intrusion Detection and Prevention

Sachin P. Joglekar
(University of North Texas, Denton, TX, USA
spj0004@cs.unt.edu)

Stephen R. Tate
(University of North Texas, Denton, TX, USA
srt@cs.unt.edu)

Abstract: Intrusion Detection Systems (IDS) are responsible for monitoring and analyzing host or network activity to detect intrusions in order to protect information from unauthorized access or manipulation. There are two main approaches for intrusion detection: signature-based and anomaly-based. Signature-based detection employs pattern matching to match attack signatures with observed data making it ideal for detecting known attacks. However, it cannot detect unknown attacks for which there is no signature available. Anomaly-based detection uses machine-learning techniques to create a profile of normal system behavior and uses this profile to detect deviations from the normal behavior. Although this technique is effective in detecting unknown attacks, it has a drawback of a high false alarm rate. In this paper, we describe our anomaly-based IDS designed for detecting malicious use of cryptographic and application-level protocols. Our system has several unique characteristics and benefits, such as the ability to monitor cryptographic protocols and application-level protocols embedded in encrypted sessions, a very lightweight monitoring process, and the ability to react to protocol misuse by modifying protocol response directly.

Key Words: Computer Security, Intrusion Detection, Cryptographic Protocol Abuse
Category: D.4.6, C.2.0, K.6.5

1 Introduction

Cryptographic protocols are communication protocols that rely upon cryptography to provide security services across distributed systems. Applications are increasingly relying on encryption services provided by cryptographic protocols to ensure confidentiality, integrity, and authentication during secure transactions over the network. However, the security provided by these encryption services might be undermined if the underlying security protocol[1] has design or implementation flaws. In fact, research has revealed numerous weaknesses in security protocols [Dolev and Yao 1983, Millen et al. 1987, Meadows 1992, Lowe 1996, Mitchell et al. 1997, Song 1999] with results ranging from the misuse of encryption [Syverson 1994] to compromising the private encryption key [Brumley and Boneh 2003]. Much research in this area has focused on applying formal methods for analysis and verification of security protocols, and although much of this research was successful in detecting flaws in and improving the protocols, it remains a

[1] The terms cryptographic protocol and security protocol are used interchangeably throughout this paper.

fact that complete security of cryptographic protocols is still a work in progress. Given the imperfect nature of security protocol design, it can be concluded that in order to improve the confidence in security protocols, detecting intrusions in those protocols is important.

Intrusion detection is a well-studied field, and two main detection approaches have been in use for several years: signature-based and anomaly-based. The signature-based approach is effective for detecting known attacks, since it matches the monitored data with a database of known attack signatures to detect suspicious activities. However, due to its reliance on attack signatures, it is ineffective against previously unknown attacks and modified versions of known attacks. In addition, available but imprecise signatures may cause increased false positives. Anomaly-based detection tries to ameliorate these problems by focusing on modeling normal system behavior in order to be able to detect behavioral deviations. A normal system behavior profile is created by observing data over a period of time, and then intrusions are detected as deviations in the monitored behavior, enabling anomaly-based detection systems to detect novel attacks. Although this is a distinct advantage over signature-based systems, anomaly-based systems have demonstrated difficulty in selecting system features in order to characterize normal behavior in such a way that any subtle deviation caused by malicious activities is not missed, and at the same time expected deviations do not generate alerts. Further, imprecise normal behavior profiles may increase false alerts, limiting the applications of anomaly-based systems in practice. However, recent research has introduced a new approach, specification-based detection, which has been applied to address the problem of high false positives. The specification-based intrusion detection approach uses manually developed specifications to characterize the legitimate system behavior, rather than relying on machine-learning techniques to learn the normal behavior, thus eliminating false positives caused by legitimate but previously unseen behavior. The advantages of specification-based and anomaly-based approaches were combined by [Sekar et al. 2002] in their specification-based anomaly detection system, which greatly simplifies feature selection while being able to detect novel attacks.

Given the success of current intrusion detection technology, applying it to detecting intrusions in cryptographic protocols seems to be an attractive choice. However, most network intrusion detection systems inspect network packet fields in order to match them with attack signatures or to generate a high-level model of interactions between communicating principals to detect suspicious activity. These activities are infeasible at the network level when protocol sessions are encrypted, which suggests that application level techniques must be employed in such a setting. Indeed, recently, Yasinsac [Yasinsac 2000] demonstrated that dynamic analysis of security protocols, rather than a static analysis as in the case of formal methods, enables detection of certain class of attacks on cryptographic protocols. Yasinsac's technique is based on protocol-oriented state-based attack detection, which reconstructs protocol sessions in terms of state models and matches these with previously generated attack state models to detect attacks. However, the attack behavior is modeled as state-machine representations of execution traces of known protocol attacks, which is essentially a signature-based technique, and hence has a drawback of not being able to detect novel attacks.

1.1 ProtoMon

Our technique derives inspiration from the specification-based anomaly detection system of [Sekar et al. 2002] and inherits its benefits, such as reduced false alarm rate, simplified feature selection and unsupervised learning. In this paper, we propose a novel approach of instrumenting shared libraries for cryptographic and application-level protocols to be able to detect and prevent intrusions in those protocols. Our approach detects attacks on protocols embedded in encrypted sessions since we integrate the monitoring into processes taking part in the protocols. Monitoring at a gateway or even another process on the same machine will not be able to detect these attacks. The framework that we propose has the ability to move data collection and analysis off the host to make the performance impact minimal. Also, moving the analysis from the hosts to a central protocol monitor process makes it possible to correlate alerts in order to further reduce the false alarm rate and to detect network-wide attack patterns, but we did not explore this in our research prototype, and we leave this possibility for future work. In addition to describing the system design, we present experimental results to demonstrate the effectiveness of our approach in detecting some of the recent attacks on OpenSSL, such as timing attacks and password interception [Brumley and Boneh 2003, Canvel et al. 2003].

2 Related Work

Intrusion detection is a heavily studied topic, and so in this section we only highlight the work that is most directly related to the techniques we propose in this paper.

The most important characteristic of an anomaly-based IDS is its technique for learning "normal behavior." One of the most common approaches is to use probabilistic techniques, learning models of particular features of normal network traffic (as is done in the SPADE [Staniford et al. 2002], an anomaly detection plugin for Snort) or of other measurable characteristics such as system calls (such as the work of Forrest and Somayaji [Forrest et al. 1996, Somayaji and Forrest 2000]). Our approach is somewhat like a combination of these techniques, where we model normal network protocol characteristics, but do so in a host-based system so that we can peer into encrypted packets.

More advanced modeling techniques have been applied to anomaly detection as well, including cluster analysis and data mining. Both NATE [Taylor and Alves-Foss 2001] and CLAD [Arshad and Chan 2003] are anomaly detection systems based on cluster analysis; however, both are based on sampling packet fields to determine protocol abuse, which is not possible for protocols wrapped in encrypted sessions. An example of data mining for anomaly detection is the work of Lee and Stolfo, who propose a system in which normal usage patterns are formed by learning (mining) large amounts of audit data [Lee and Stolfo 1998]. While this is an interesting direction, the complexity of such a system makes it currently unsuitable for real-time intrusion detection unless a distributed detection framework is designed. It is possible for these techniques to be applied indirectly in our setting, by incorporating them in the analysis engine, but we leave this possibility to future work.

All of the approaches to anomaly-detection discussed above have one common problem: choosing the correct features to be learned in order to create an accurate behavior profile of the system. Consequently, attacks that manifest themselves as anomalies in the features not included in the normal behavior profile are likely to be missed. This problem is ameliorated by a new variant of anomaly detection, specification-based anomaly detection, which uses specifications of protocols given in standard documentation like RFCs to select appropriate features. Use of protocol specifications enables manual characterization of legitimate behavior rather than learning the normal behavior by observing data over a period of time. As another advantage, this approach alleviates the problem of high false positives evidenced in the anomaly detection, by eliminating the false positives caused by behaviors that are legitimate but absent in the training data.

Sekar *et al.* proposed a method, specification-based anomaly detection, that uses state-machine specifications of network protocols and statistical learning to map packet sequence properties to properties of state-machine transitions for detecting network intrusions [Sekar et al. 2002]. The effectiveness of their approach is evidenced at a network gateway by successful detection of most attacks by simply monitoring the distribution of frequencies with which each state transition is taken. However, as noted by Sekar *et al.*, current intrusion detection techniques rely primarily on inspecting network packet fields to gather information about the system behavior, which becomes infeasible when sessions are encrypted. With the advent of IPv6 and and wider use of IPSec this problem will get worse, as there will be more end-to-end encrypted connections making network-level detection techniques ineffective for detecting application-level attacks. Our approach addresses this problem by embedding the monitoring into the protocol process, thus eliminating the need for inspection at the network level.

The problem addressed by our work and others in detecting abuse of security protocols is complementary to the problem of designing secure protocols. Extensive work has been done to formally verify and test the security protocols to prove that they are secure. Several works have been published on the topic after [Needham and Schroeder 1978] expressed the need for techniques for cryptographic protocol analysis, and Dolev and Yao [Dolev and Yao 1983] developed a polynomial-time algorithm for deciding the security of restricted class of protocols. Based on these foundations, several tools for formally analyzing security protocols were developed including special-purpose model-checkers such as the Interrogator [Millen et al. 1987] and the NRL Protocol Analyzer [Meadows 1992], and general-purpose model-checkers such as FDR [Lowe 1996] and Mur [Mitchell et al. 1997]. More recent work by Song has addressed the problem of state space explosion experienced by model checkers, promising more efficient formal analysis tools [Song 1999]. However, given that the protocol security problem is undecidable [Cervesato 1999, Heintze 1996], these techniques have limits to their completeness, meaning that the analysis tools will not be successful all the time. In addition, turning a protocol definition into a concrete implementation often introduces vulnerabilities that are not present in the protocol specification. Therefore, supplemental to the formal methods, a mechanism such as ours for detecting misuse of cryptographic protocols is highly desirable in the production environment. This argument is further strengthened by recent research results that continue to uncover serious flaws in cryptographic protocols [Brumley and Boneh 2003, Canvel et al. 2003].

More recently, this issue was addressed by Yasinsac who applied classic knowledge-based and behavior-based intrusion detection techniques for detecting intrusions in security protocols [Yasinsac 2000, Yasinsac 2002]. This technique develops signatures of protocol attacks by gathering information from three sources:

- Known attacks identified in the literature.
- Attack taxonomies identified in the literature.
- Flaws and suspicious activity gathered during execution.

The attack signatures are developed from protocol execution traces modeled as state machines, which are then used by the detection system for matching with the state machines that are dynamically built during actual protocol execution. Unlike our approach, this approach requires explicit knowledge of attack signatures in terms of protocol execution traces, making it a signature-based system. Our approach, using the anomaly-based methodology, needs no prior knowledge of attack signatures, thus enabling it to detect novel attacks as well as modifications of known attacks.

3 Overview of ProtoMon

Identifying protocol implementation libraries is our first step for instrumenting embedded monitors in the protocol process. We implement the embedded monitors inside shared libraries for security protocols, giving those libraries the capability of acting as sensors within our monitoring framework. The benefit is that intrusion detection capabilities are inherited by any application which uses the libraries. For example, multiple applications on a Linux host may use libssl to protect communication, so instrumenting this one library gives monitoring functionality to all these applications. While we found one small problem in providing completely application-transparent intrusion detection sensors (see the end of section 3.1.3 for a description), this design proved easy to implement and very flexible. It should be noted that unlike traditional IDS sensors which extract data from passive sources such as audit logs, our sensors are directly integrated inside the process and generate signals as the process executes.

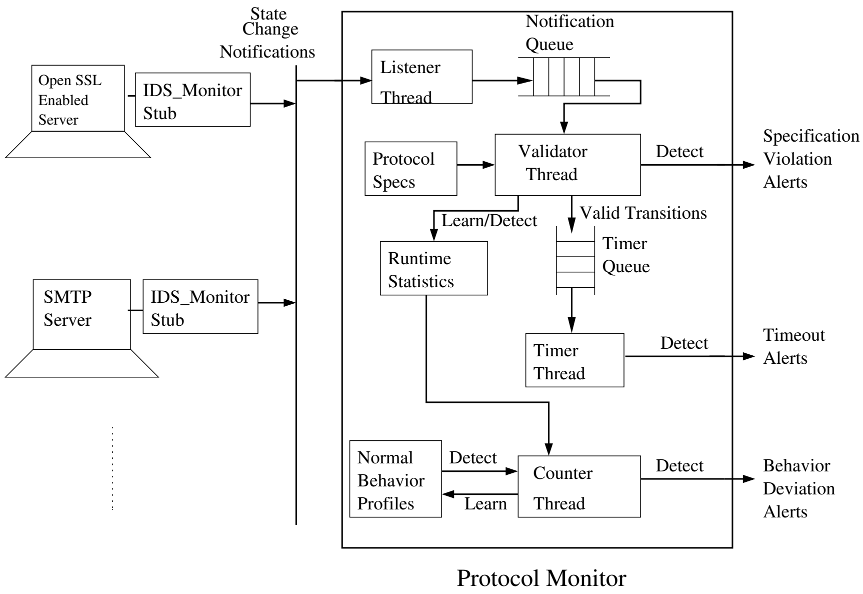

Figure 1: Generic Protocol Monitoring Framework

3.1 Generic Protocol Monitoring Framework

[Fig. 1] shows the basic architecture of our system, which we describe more fully in the remainder of this section. Our system is a generic protocol monitoring framework that can be configured as a host-based intrusion detection system or as a central protocol monitor for all hosts on a network, moving data collection and analysis off the hosts to minimize the performance impact, as illustrated in [Fig. 1]. Our framework consists of three main components: the protocol monitor, pluggable protocol behavior profiles, and monitor stubs. We now briefly describe each of these components.

3.1.1 The Protocol Monitor.

The protocol monitor is primarily responsible for collecting and analyzing the protocol state transition notifications sent by the stubs and detecting anomalies in the protocol behavior. The anomalies are detected by building a short-term profile of the protocol usage and comparing it with the normal behavior profile built during the learning phase. We show in section 4 that most of the attacks that are within the scope of our system are detected by monitoring the state change notifications. The protocol monitor can be operated in three modes, learn, detect, and prevent, which are described next.

Learn mode. In the learn mode, the protocol monitor creates normal behavior profiles for target protocols by taking manually developed protocol state-machine specifications and then observing training data for a period of time to be able to perform statistical analysis of state transitions reported by monitor stubs. It is important that the training data reflects the expected usage pattern of the protocol so that the resultant normal behavior profile captures the correct expected behavior, thus reducing the false positives. However, it should be noted that since we use the specification-based approach, the scope of the learning phase is limited to statistical analysis of legitimate protocol use, rather than learning the legitimate use itself, so this system is less susceptible to false alarms generated by legitimate behavior that is unseen in the training data. Further, the usage statistics of the network services, and hence the use of the protocols, can vary significantly depending on day and time, so ProtoMon can gather statistics specifically for different time periods. In our experiments we use 6 different time slots for each day, but this is of course easily adaptable to other usage patterns. The statistics collected by the monitor in learn mode are:

- Maximum number of protocol sessions for each time slot.
- Maximum number of broken protocol sessions for each time slot.
- State transition statistics. These include the average of the number of times each state transition is taken. The averages are calculated per second, per minute, and per hour to be able to detect attacks that create sudden and short-term anomalies as well as attacks that cause slow and relatively long-term anomalies.

At the end of the learning phase the protocol monitor creates a normal behavior profile for each monitored protocol by recording and averaging the statistics mentioned above.

Detect mode. The normal protocol behavior profiles created in the learning phase are subsequently used by the protocol monitor in the detect mode as a protocol usage baseline, both in terms of legitimate protocol use and expected protocol usage statistics. We define a tolerance limit for the deviation of protocol behavior as the maximum deviation from the normal behavior that is acceptable and is not flagged as anomalous. The protocol monitor creates a short-term protocol usage profile at regular intervals by observing the data and comparing it with tolerance limits of the baseline provided by the normal behavior profile to be able to detect specification violations and statistical anomalies. To be able to successfully execute an undetected attack, the attack must not create any protocol behavior anomalies and must strictly follow the protocol state transition specifications. Any attempts that cause specification violations or behavior anomalies are immediately detectable.

Prevent mode. Intrusion response is an active field of research and various approaches are currently in use, with the most common using techniques such as blocking traffic from offending IP addresses and forcefully resetting connections (e.g., as done by PortSentry [Psionic Technologies]). However, these techniques could be used by an adversary to make an IDS block the traffic from non-offending IP addresses causing a denial of service. We address this issue, as it pertains to responding to protocol misuse, by using the prevent mode of operation. In prevent mode, upon detecting a protocol behavior that rises above the upper tolerance limit, the protocol monitor coordinates with the monitor stub and slows down the protocol response. This is possible because of our positioning the monitor stubs inside the protocol process, which allows the monitor stub to introduce a delay in each protocol state transition as long as the protocol monitor signals anomalous behavior. This is similar to some previous systems, including pH's technique of introducing system-call delays in processes that show abnormal behavior [Somayaji and Forrest 2000], and LaBrea's technique of deliberately slowing the connection from suspected scanner machines [LaBrea]. Our embedded stubs force the protocol behavior to remain within the upper tolerance limit, causing attacks that need a large number of sessions to be slowed down to the point where they are no longer effective. Further, elongating the time needed for the attack to be successful allows for human intervention. Also, as a supplemental response to the detected attack, alerts generated by the protocol monitor could be used to invoke other response mechanisms, such as at the router, to stop an attack. Since the stubs remove the delays once the behavior returns back in the tolerance limit, the protocol is not completely halted and our mechanism cannot be used by an attacker to cause an indefinite denial of service. However, it should be noted that the prevent mode introduces increased network traffic overhead due to the increased communication between the protocol monitor and the monitor stub, as illustrated in Section 4.

Architecture of the Protocol Monitor. The protocol monitor process consists of four threads: listener, validator, timer, and counter. [Fig. 1] shows the interaction between these threads and the alerts that they generate. The listener collects the state change notifications generated by the monitor stubs and inserts them into a notification queue. The validator thread picks up the notifications from the notification queue and validates them against the protocol specification to detect any specification violations. It also generates statistics of protocol usage which are later used by the counter thread to create a behavior profile. Valid state change notifications are inserted in the timer queue for timing each state, which allows the protocol monitor to generate timeout alerts if the protocol state machine is aborted before the final state is reached. Observing the number and frequency of timeout alerts detects some of the side channel attacks described in Section 4. The counter thread constantly compares the normal behavior profile with the short term profile, referred to as "*Runtime statistics*", that it generates using the state transition statistics, collected by the validator thread. Significant difference between the state transition statistics in the short term profile and the ones in the normal behavior profile are detected as anomalies by the counter thread.

3.1.2 Pluggable Protocol Behavior Profiles.

At the end of the learning phase, protocol behavior profiles are generated for all target protocols. A behavior profile has a specification component, which is manually developed in terms of a protocol state-machine by studying implementations of the protocol and standard documents like RFCs. The specification component is complimented by a statistical component, which is built during the learning phase, in which training data is used to learn normal protocol usage statistics. The specification component characterizes legitimate protocol usage in terms of valid states, start states, final states, and all valid state transitions. The rationale behind incorporating these two components is that some of the attacks cause the protocol to directly violate the protocol specification and are thus detected immediately by validations performed against the specification component, whereas attacks that may not generate specification violations but merely are manifested as traffic or usage anomalies are detected due to the usage baseline provided by the statistical component. It is important that the training data resembles the use of the protocol in the target environment to ensure increased precision of behavior profiles, and, as stated earlier, we can use different profiles for different time periods. Profiles generated in this way are then loaded in to the protocol monitor and are used to be able to enforce legitimate and expected protocol usage and to detect attacks as anomalies.

The following is a sample simplified entry in the learned profile of OpenSSL for a weekday afternoon. In this example we only show a subset of 13 states, rather than a full specification of SSLv3, which would require 70 states (and a combined SSLv3 and SSLv2 specification requires 108 states). However, even with the simplified example, the basic structure can be seen.

```
Statistics weekday afternoon{
  8464 {8496 1 64 3779}
  8496 {8512 1 64 3779} {8656 0 0 0}
  8512 {8528 1 64 3779}
  8528 {8544 1 64 3779}
  8544 {8560 1 64 3779}
  8560 {8448 1 64 3779}
  8448 {8576 1 64 3779} {8640 0 0 0} {3 1 64 3779}
  8576 {8592 1 64 3779}
  8592 {8608 1 64 3779}
  8608 {8640 1 64 3779}
  8640 {8656 1 64 3779} {3 0 0 0}
  8656 {8672 1 64 3779}
  8672 {8448 1 64 3779}
  3
  maxSessions=9983
  maxBrokenSessions=57
}
```

Each line begins with a number of a current state followed by all states reachable from that state and the statistics on that particular state transition for three different time intervals: 1 second, 1 minute and 1 hour. This enables detection of attacks that create short-term anomalies which might not get manifested as an anomaly in relatively long term statistics, as well as attacks that create anomalies in the long-term statistics.

3.1.3 Monitor Stubs.

Stubs provide a generic interface between the protocol process and the protocol monitor and are integrated into shared libraries of the protocol implementation. Integrating the stubs with the protocol libraries allows them to capture the protocol activity dynamically, rather than relying on passive methods such as audit logs, and to monitor encrypted protocol sessions. Monitor stubs perform a handshake with the protocol monitor to check network availability and to know the mode in which the protocol monitor is running (learn, detect, or prevent). The stubs send protocol state change notifications to either the local protocol monitor functioning as a host-based monitoring system, or to a central protocol monitor for network-wide analysis and to reduce the performance impact on the host. During the experiments, the process of integrating the stubs with the protocol libraries required a very minimal change in the protocol implementation files since the state transitions were clearly and directly implemented in the original libraries, which we conjecture would be the case with any well-written protocol library.

We did experience one problem in creating completely application-independent monitor stubs. Knowing the IP address of the source of packets is important for tracking sessions, and for identifying the source of attacks, but this information is not necessarily present in the protocol library. In our specific case, we inserted monitor stubs in the OpenSSL library, which uses an abstraction for I/O handles. In our initial tests, using sample OpenSSL applications, these I/O handles were sockets, and the source IP address could be obtained by calling `getpeername()` on the socket (albeit at the cost of a system call). However, when we experimented with the Apache web server using the OpenSSL library, we discovered that the I/O handles in that case actually refer to general memory buffers, and Apache itself handles the I/O, converting inputs through a series of filters before actually reaching the OpenSSL library. Because of this, the actual I/O socket is several levels removed from (and above) the OpenSSL library, and it wasn't clear how to access the socket to obtain the peer IP address. This reveals a problem with our application-independent approach in situations where the application hides the source of packets from the security protocol engine. We did not fully fix this problem in our research prototype. Our OpenSSL modifications determine whether the I/O handles refer to a socket, in which case the source IP address can be determined – in other situations, we simply use a generic IP address for all sessions, losing the ability to identify the attacker. We suggest that a production system would include a way for an application to register a callback function with the IDS monitor for giving the source IP information, so that if it handles I/O itself it can provide the appropriate information. Unfortunately, this means a change must be made in the application as well as in the protocol library, but there doesn't seem to be any way to avoid this.

3.2 Benefits of Approach

The approach taken by ProtoMon has several distinct benefits and advantages, which we discuss in this section.

Detection of attacks on cryptographic protocols and application-level protocols embedded in encrypted sessions. As a primary contribution of our work, our system is able to detect attacks on the application protocols that use encryption to provide authentication, key distribution, and other services necessary for secure communication between participating principals. Current intrusion detection systems which detect attacks on application-level protocols primarily analyze the application-level payload in network packets to match patterns of known attacks. Use of encryption by the application-level protocol makes this analysis infeasible since the payload is encrypted and can only be decrypted at the application-level. We have addressed this issue by positioning our sensors inside the application-level protocol process. Further, application of the specification-based anomaly detection approach enables detection of known and unknown attacks and should keep the false positive rate low.

Generic framework. Although we demonstrate the ability of our system to detect attacks on application-level and cryptographic protocols, the design of our framework is generic, meaning it could be applied to any arbitrary protocol with clearly defined specifications. As noted earlier, this will allow our system to act as a central protocol monitor for the entire network enabling correlation of alerts generated due to behavior deviations in protocols running on different hosts.

Response ability. Intrusion response systems aim at preventing or minimizing the damage caused by detected intrusions. Detection systems that are based on analyzing information from passive sources such as audit logs rely on peripheral response actions such as isolating the target host or changing network firewall rules to block traffic from offending IP addresses. However, the response of the application which is under attack does not change to defend itself. In our approach, due to their positioning inside the protocol process, the monitor stubs enable changing the response of the protocol that is being attacked.

Lightweight. Our system is lightweight meaning it adds minimal overhead to the host that is running the monitored protocol. The prevent mode introduces increased overhead, as compared to the detect mode, due to the increased communication between the protocol monitor and the stub; however, in this case the entire purpose is to slow down protocol response, so additional overhead doesn't seem to be a particular burden.

4 Experimental Results

For evaluation purposes we used attacks against OpenSSL, and the OpenSSL version 0.9.6 library under Linux was our primary experimental target.

4.1 Experimental setup

We carried out the experiments on a private LAN with no significant background traffic and with the configuration shown in [Fig. 2]. During the learning phase, normal OpenSSL protocol usage was simulated by using a four week access log of a real web

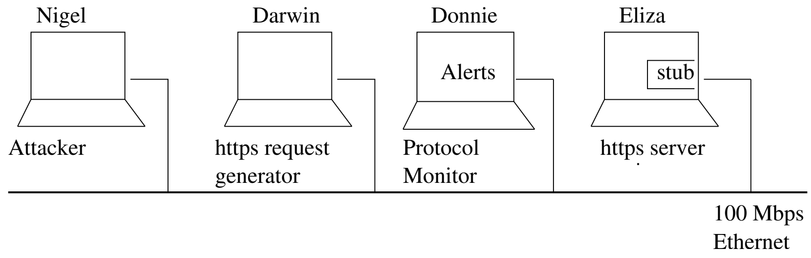

Figure 2: Experimental setup

site. Any instances of attacks in the access log were removed in order to exclude the behavior of the protocol during the attack. Statistics collected during the learning phase were then used in the detect phase to demonstrate the ability of the system to detect attacks not observed during the learning phase.

4.2 Attack Prevention

We categorize attacks that are within the scope of our system into two general classes: *atomic*, and *non-atomic*. We denote attacks which need to use the protocol once or for very few times as atomic. On the other hand, attacks that are based on using the protocol for a large number of times, such as timing attacks, are referred to as non-atomic. Non-atomic attacks are preventable in the prevent mode of the protocol monitor, whereas atomic attacks are only detectable. This follows from the fact that we use an approach of slowing down the protocol response if the protocol monitor detects the protocol behavior going beyond the upper tolerance limit. Atomic attacks do not generate this type of anomaly and hence cannot be prevented by the slow-down mechanism. However, our results show that this mechanism is effective in considerably elongating the time taken by an ongoing non-atomic attack to complete. The *slow-down-factor* can be tuned to a value that practically prevents the attack from becoming successful.

4.3 Attack Detection

In this section, we consider 3 different specific attacks on OpenSSL. We emphasize that our intrusion detection system is general-purpose and not designed or tuned specifically for these attacks, which are chosen simply as representative attacks on SSL that can be found in the literature.

Side-channel attacks. Side-channel attacks have been shown to be possible and practical on cryptographic protocols based on RSA encryption routines [Brumley and Boneh 2003, Canvel et al. 2003, Klima et al. 2003, Bleichenbacher 1998]. These attacks deduce the private key, or invert the encryption with the help of information that is unintentionally leaked by the protocol. In the case of a particular timing attack against OpenSSL

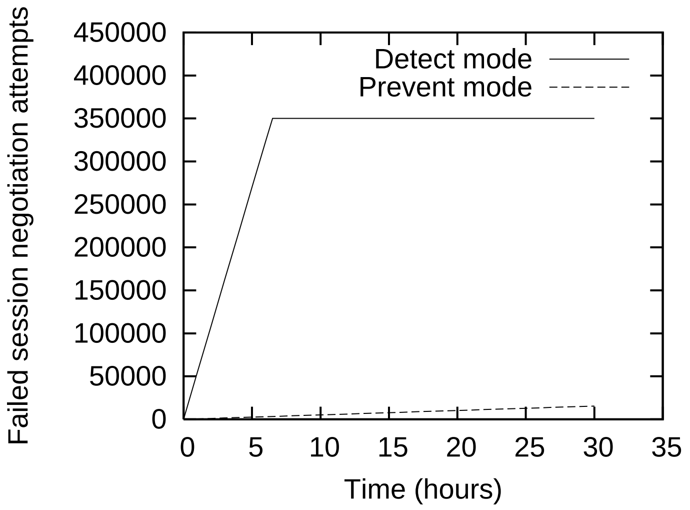

Figure 3: Detection of Timing Attack on OpenSSL

implementations, it takes around 350,000 accurately timed failed session negotiation attempts to break a 1024-bit RSA private key [Brumley and Boneh 2003]. As a general characteristic of these attacks, each failed attempt to negotiate a session results in aborting the protocol state-machine before the final handshake state is reached. Our protocol monitor times each state and observes the increased number of timeouts, which detects these attacks. We simulated the timing attack by tweaking the OpenSSL client library and using a dummy SSL client application to repeatedly abort the handshake at a particular state. [Fig. 3] demonstrates the increased number of timeouts during the time when the attack is in progress. The attack simulation took approximately 6 hours to complete in the detect mode, whereas in the prevent mode, with 1 second delay introduced in the state transitions after the threshold number of sessions were aborted, the stub slowed down the protocol and it took an average of 7 seconds per session. As seen, this considerably elongates the time taken for the attack to be successful. While the original attack simulation completed all 350,000 probes in approximately 6 hours, the prevent mode protocol monitor allowed only about 15,420 probes in the first 30 hours. Not only would the required 350,000 probes take 680 hours at this rate, but also note that the main characteristic of this attack, the timing of protocol events, is disrupted by the protocol delays, rendering the attack completely ineffective.

Rollback Attacks. Wagner and Schneier pointed out possible vulnerabilities in the specification of SSL version 3 [Wagner and Schneier 1996], and we specifically demonstrate detection of version rollback and cipher-suite rollback attacks. SSLv3 is susceptible to a version rollback attack in which the *client hello* message sent by the client during session resumption can be intercepted by an intruder to change the version number to 2, thus forcing the server to downgrade the version used in the session with that client. If the server finds the session id supplied in the client hello message in its cache, it assumes resumption of a previous session, but with a lower SSL version and directly proceeds to the *finished* message. Further, unlike SSLv3, the *finished* message in SSLv2 does not include the version number making this attack undetectable. The downgrading of the SSL version by the server exposes the server to several vulnerabilities in SSLv2. To be able to detect this attack, the protocol monitor maintains separate session caches for SSLv2 and SSLv3 as suggested by Wagner and Schneier. This attack is detected by simply observing a transition from the *SSL2_ST_SEND_SERVER_HELLO_A* state to the *SSL2_ST_SERVER_START_ENCRYPTION* state reported by a monitor stub embedded in SSLv2 for a session id that was previously registered with the protocol monitor by a stub in SSLv3.

The cipher-suite rollback attack exploits the fact that the *change cipher spec* message is excluded from the calculation of a MAC on the previous handshake messages, which is used as an authentication code for the *finished* message. The attacker can drop the *change cipher spec* message causing the server and the client to never change the pending cipher-suite to the current cipher-suite, potentially disabling encryption and message authentication at the record layer. Although the SSLv3 protocol specification documents the *change cipher spec* message as an optional message, it is most likely a cipher-suite rollback attack if this message is excluded from the initial handshake since the initial configuration provides no encryption or authentication, with the exception of the session resumption scenario. The protocol monitor requires the *change cipher spec* message during the initial handshake, and dropping this message by the attacker results in a specification violation that is immediately detected.

Buffer overflow and denial of service attacks. Buffer overflow vulnerabilities that exist in the implementations of SSL protocol routines [CERT 2002b, CERT 2002a] could be exploited to execute arbitrary code on the server. For example, an Apache mod_ssl worm exploited a buffer overflow during the SSLv2 handshake process. Although the attacker can gain full control of the server by executing this arbitrary code, it should be noted that the attacks that fall in this category most likely result in aborting the state-model of the protocol at an arbitrary state. The protocol monitor detects this behavior by using the timeout mechanism described earlier. The timeout alert could then be correlated with the anomaly in protocol statistics reported to the protocol monitor to detect the exploit or denial of service that may be caused by the executed code.

4.4 Behavior comparison

[Fig. 4] demonstrates the behavior of OpenSSL during the two operating modes: *detect* and *prevent*. As seen, alerts are generated in both modes when the protocol behavior deviates from the normal behavior beyond the tolerance limit. However, in the prevent mode, the protocol monitor signals the monitor stub to insert delays in the protocol state transitions upon detection of significant deviation, effectively slowing down the

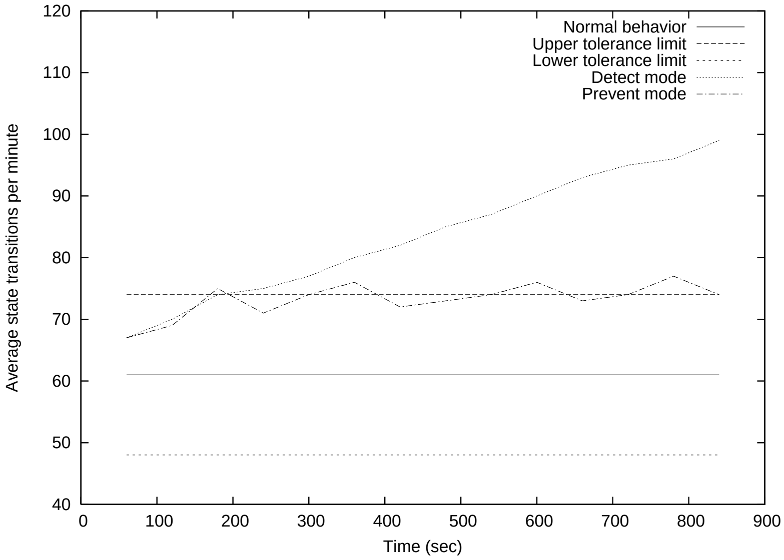

Figure 4: Protocol behavior comparison

protocol and forcing the protocol behavior to be within the upper tolerance limit of the normal behavior. If the protocol behavior deviates toward the lower tolerance limit, alerts are generated in both the modes.

[Fig. 5] illustrates variation in the protocol response time caused by the delays introduced by the stub as a result of the protocol monitor signaling the deviation of the protocol behavior beyond the upper tolerance limit. The behavior is sampled every second, every minute and every hour. [Fig. 4] shows the behavior sampled every minute with the behavior crossing the upper tolerance limit at around 180 seconds into the experiment. As a result the protocol response time plotted in [Fig. 5] shows the corresponding increase for the period of 90 seconds.

In the preliminary report of these results [Joglekar and Tate 2004], the protocol monitor slowed down all clients connecting to a particular service whenever an anomaly was detected. Since that time, ProtoMon has been enhanced to slow down only particular clients that seem to be causing problems. It is not entirely obvious how to do this for statistical anomalies, since the statistics give a total picture of the system behavior, and the connection that puts the statistics out of the acceptable range may in fact not be the malicious client. Our approach was to use the following heuristic: any client responsible for at least 30% of the recent requests would be slowed down. This heuristic may miss a malicious client, and it may incorrectly slow down a non-malicious client, but worked well in our tests. We leave the problem of more reliable identification of a malicious client as an open problem.

To test this enhancement to ProtoMon, we simulated two clients simultaneously on

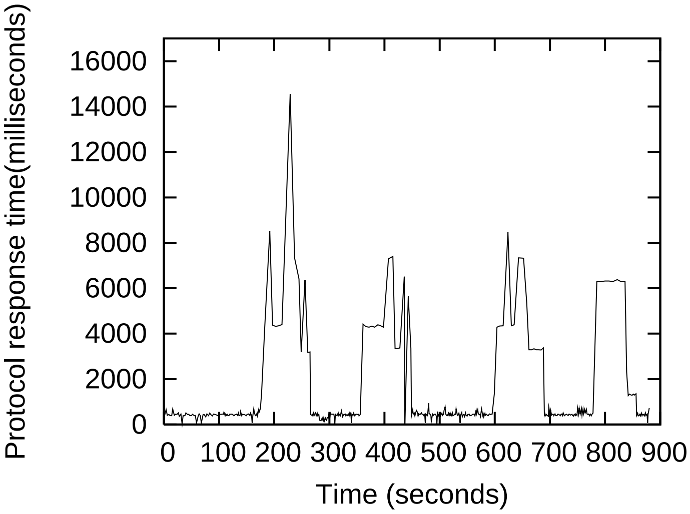

Figure 5: Protocol response time variation

two separate machines, with one client providing malicious behavior and the other client performing within the tolerance limits of the behavior profile. [Fig. 6] shows the result of our tests. While the malicious client is slowed down to approximately 14 seconds per SSL session when the behavior exceeded the defined tolerance limits, the good client continued to operate at full speed.

4.5 Performance Overheads

Given that cryptographic computations based on public key algorithms such as RSA are complex and resource intensive, it is desirable to minimize the overhead added by any mechanism that aims at detecting intrusions in protocols based on these algorithms. This section shows overheads in the performance of the OpenSSL protocol implementation caused by our detection and prevention mechanism, as measured with our experimental setup of [Fig. 2].

4.5.1 Network Bandwidth.

The monitor stub sends state transition notifications to the protocol monitor over the network when they are running on different hosts, resulting in additional network traffic. The following table shows the average network traffic overhead over 3409 SSL protected HTTP requests, with requests sampled from a production web server. Measured sizes indicate the size of complete IP datagrams, including all IP and TCP or UDP header information, and measurements were taking by analyzing a “tcpdump” sniffer

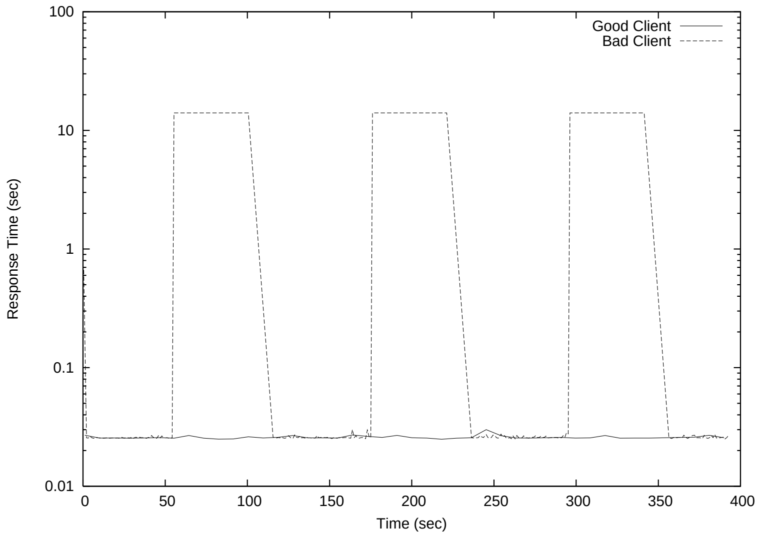

Figure 6: Two client response time experiment

log of actual traffic during the entire experiment. The request sequence had an average data response size of 7344 bytes, with an average of 2610 additional bytes in overhead due to SSL messages, HTTP headers, and TCP/IP headers. The ProtoMon overhead, including IP and UDP headers, was approximately 728 bytes per SSL session in detect mode. In prevent mode, additional packets are required to notify the protocol engine to slow down, giving a slightly higher overhead, but only when malicious behavior has been detected. These results are summarized in the following table, where the percentages reflect the percent of total traffic (SSL and ProtoMon) due to the ProtoMon traffic.

	Detect mode	Prevent mode
Network overhead per SSL session	728 bytes (6.8%)	760 bytes (7.1%)

Reducing Network Overhead. Since our initial implementation and test results with ProtoMon were reported, we have implemented a "traffic aggregator" to reduce network overhead. This is simply a process that sits on the monitored machine, and collects multiple state change notifications into a single, larger UDP packet. Since the notification data itself is quite small, much of the ProtoMon overhead reported above is actually IP and UDP header information, and by accumulating several of these notifications into a single packet, we can significantly reduce the amount of bandwidth required. In particular, the traffic aggregator collects state change notifications until one of two things happens: either a long packet is filled up, or one second of time has elapsed. At this point, the collected data is sent to the protocol monitor in a single packet. In addition, since much information is repeated between successive state change notifications (the source and destination address information is repeated, and the destination state of one transition is the source state of the next), additional space can be saved by carefully coding the notifications. These two changes taken together significantly reduced the amount of network overhead required by ProtoMon, as illustrated by the measurements shown in the following table.

	Detect mode	Prevent mode
Network overhead per SSL session	140 bytes (1.4%)	172 bytes (1.7%)

While this network overhead is very modest, we note that network impact can be basically eliminated by using a second network card and routing ProtoMon traffic over a different network.

4.5.2 Protocol Response Time.

As described in Section 4.1, we used the normal behavior profile of OpenSSL to detect anomalies in the detect and prevent modes of operation. The following table shows average response time of the OpenSSL enabled web server per request after anomalies have been detected.

	Response time per request (ms)	Percentage slow-down-factor
OpenSSL	385.66	-
Learn mode	404.81	4.9%
Detect mode	406.02	5.2%
Prevent mode	2080.60	439.4%

Prevent mode slows down the protocol response by the factor of approximately 4.4 for the purpose of this experiment. This *slow-down-factor* can be tuned to a desired value by adjusting the delays introduced by the monitor stubs. Consequently, the time taken by any attack that spans across multiple protocol sessions can be elongated when the protocol monitor is running in the prevent mode.

4.5.3 Processor overhead.

When the protocol monitor is used as a host-based detection mechanism it uses processor time on the server. Our experiment involving measurements of processor overhead were carried out by using the protocol monitor as a host-based detection system. We used a host with a Pentium4 2GHz processor to run the protocol and the protocol monitor. The results, in the following table, show the percentage increase in the processor usage by the protocol process when the protocol monitor is used in learn, detect, and prevent mode.

	Processor usage per session (ms)	Percentage overhead
Standard OpenSSL	22.2	-
Learn mode	25.5	14.8%
Detect mode	25.7	15.7%
Prevent mode	26.3	18.4%

This overhead can be minimized by placing the protocol monitor process on a separate, central host, which has the added benefit of enabling network-wide alert analysis and correlation.

5 Future Work

Although our results indicate that our approach of embedding monitors inside the executing protocol process detects and prevents attacks on those protocols, there remain several open research issues that need further attention.

First, it is possible to apply more advanced learning techniques and models to building comprehensive behavior profiles. Further research is needed in applying such models to enhance the precision of the normal behavior profile. Adaptive learning strategies could be employed to be able to continuously modify the normal behavior profile to take into account legitimate and expected changes in the protocol usage patterns.

In addition, the high false alarm rate experienced by anomaly-based intrusion detection systems is a current research topic. We believe that alerts caused by protocols running on various nodes in the network could be correlated to reduce false alarms generated by a centralized protocol monitoring process. This argument is strengthened by the fact that a successful attack performed by an active attacker often involves compromising various network services on the network to cause maximum damage. Consequently, the protocols used by these network services, if monitored at a centralized protocol monitor, should cause alerts that could be correlated for a more comprehensive and correct attack detection.

As discussed in Section 4, our current system is able to prevent non-atomic attacks on cryptographic protocols. Further research is needed in order to be able to include atomic attacks in the scope of the response mechanism proposed in this paper. It could be possible to implement the monitor stub as a wrapper around the protocol process, synchronously validating all the state transitions before passing them on to the protocol process. This should allow detection of atomic attacks: for example, the cipher suite rollback attack could be prevented by first validating the state transition to detect the dropping of the *change cipher spec* message and then aborting the session to prevent the unprotected session from continuing. However, this process of complete mediation of all state transitions would almost certainly slow down the protocol response significantly, even under non-attack conditions.

6 Conclusion

The main issue addressed by this paper is real-time detection of intrusion attempts in application level protocols encapsulated inside encrypted sessions. The results shown in this paper illustrate that embedding the monitor stubs inside the cryptographic protocol process restricts the avenues available to the attacker to those attacks that create neither protocol specification violations nor protocol usage behavior anomalies, both of which are detectable by our proposed system. The impact on both the machine being monitored and the network usage is minimal, if the protocol monitor analysis engine is on a separate system. As another contribution, this paper demonstrates that slowing down the protocol response is an effective way to prevent certain types of protocol attacks and minimize the damage to the system.

Acknowledgments

The second author gratefully acknowledges the support of the National Science Foundation through NSF award 0208640.

References

[Arshad and Chan 2003] M. Arshad, P. Chan, Identifying Outliers via Clustering for Anomaly Detection, Technical Report CS-2003-19, Dept. of Computer Science, Florida Institute of Technology, 2003.

[Bleichenbacher 1998] D. Bleichenbacher, Chosen Ciphertext Attacks against Protocols based on the RSA Encryption Standard PKCS#1, Proceedings of CRYPTO'98, pp 1-12, 1998.

[Brumley and Boneh 2003] D. Brumley, D. Boneh, Remote Timing Attacks are Practical, 12th USENIX Security Symposium, pp. 1–14, 2003.

[Canvel et al. 2003] B. Canvel, A. Hiltgen, S. Vaudenay, M. Vuagnoux, Password Interception in a SSL/TLS Channel, in Proc. of CRYPTO 2003, pp. 583–599, 2003.

[CERT 2002a] CERT, "OpenSSL servers contain a buffer overflow during the SSL2 handshake process", CERT Vulnerability Note #102795, July 2002.

[CERT 2002b] CERT, "OpenSSL servers contain a remotely exploitable buffer overflow vulnerability during the SSL3 handshake process", CERT Vulnerability Note #561275, July 2002.

[Cervesato 1999] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, A. Scedrov, A Meta-Notation for Protocol Analysis, Proc. 12th IEEE Computer Security Foundations Workshop, pp. 55–69, 1999.

[Dolev and Yao 1983] D. Dolev, A. Yao, On the Security of Public Key Protocols, IEEE Transactions on Information Theory, 29(2):198-208, March 1983.

[Forrest et al. 1996] S. Forrest, S.A. Hofmeyr, A. Somayaji, A Sense of Self for Unix Processes, IEEE Symposium on Security and Privacy, pp. 120–128, 1996.

[Heintze 1996] N. Heintze, J.D. Tygar, A Model for Secure Protocols and their Composition, IEEE Transactions on Software Engineering, 22(1):16-30, January 1996.

[Joglekar and Tate 2004] S. P. Joglekar and S. R. Tate, ProtoMon: Embedded Monitors for Cryptographic Protocol Intrusion Detection and Prevention, *2004 IEEE Conference on Information Technology: Coding and Computing (ITCC)*, pages 81-88.

[Klima et al. 2003] V. Klima, O. Pokorny, T. Rosa, "Attacking RSA-based Sessions in SSL/TLS," in Proc. of Cryptographic Hardware and Embedded Systems (CHES), 2003, pp. 426–440.

[LaBrea] The LaBrea Project, Sourceforge, http://labrea.sourceforge.net.
[Lee and Stolfo 1998] W. Lee, S. Stolfo, Data mining Approaches for Intrusion Detection, USENIX Security Symposium, 1998, pp. 79–93.
[Lowe 1996] G. Lowe, Breaking and Fixing the Needham-Schroeder Public Key Protocol using FDR, in Proc. of the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, pp. 147-156, 1996.
[Meadows 1992] C. Meadows, Applying Formal Methods to the Analysis of a Key Management Protocol, Journal of Computer Security, 1:5-53, 1992.
[Millen et al. 1987] J.K. Millen, S.C. Clark, S.B. Freedman, The interrogator: Protocol Security Analysis, IEEE Transaction Software Engineering, SE-13(2):274-288, Feb. 1987.
[Mitchell et al. 1997] J.C. Mitchell, M. Mitchell, U. Stern, Automated Analysis of Cryptographic Protocols using Murφ, Proc. 1997 IEEE Symposium on Security and Privacy, 1997, pp. 141–151.
[Needham and Schroeder 1978] R.M. Needham, M.D. Schroeder, Using Encryption for Authentication in Large Networks of Computers, Communications of ACM, 21(12):993-999, December 1978.
[Psionic Technologies] PortSentry, by Psionic Technologies.
[Sekar et al. 2002] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, S. Zhou, Specification-based Anomaly Detection: A New Approach for Detecting Network Intrusions, ACM Computer and Communication Security Conference, 2002, pp. 265–274.
[Somayaji and Forrest 2000] A. Somayaji, S. Forrest, Automated Response using System-Call Delays, 9th USENIX Security Symposium, 2000, pp. 185–198.
[Song 1999] D. Song, Athena - An Automatic Checker for Security Protocol Analysis, Proc. Computer Security Foundation Workshop, 1999, pp. 192–202.
[Staniford et al. 2002] S. Staniford, J. Hoagland, J. McAlerney, “Practical Automated Detection of Stealthy Portscans,” Journal of Computer Security, 10(1/2): 105–136, 2002.
[Syverson 1994] P. Syverson, A Taxonomy of Replay Attacks, Proceedings of the Computer Security Foundations Workshop VII, 1994, 187–191.
[Taylor and Alves-Foss 2001] C. Taylor, J. Alves-Foss, NATE-Network Analysis of Anomalous Traffic Events, A Low-cost Approach, Proceedings of the New Security Paradigms Workshop ’01, pp 89-96, September 2001.
[Wagner and Schneier 1996] D. Wagner, B. Schneier, Analysis of the SSL 3.0 Protocol, Proceedings of Second USENIX Workshop on Electronic Commerce, USENIX Press, pp. 29-40., November 1996.
[Yasinsac 2000] A. Yasinsac, Dynamic Analysis of Security Protocols, Proceedings of New Security Paradigms 2000 Workshop, Sept 18-21, Ballycotton, Ireland, pp. 77-87, 2000.
[Yasinsac 2002] A. Yasinsac, An Environment for Security Protocol Intrusion Detection, Journal of Computer Security, 10(1/2):177–188, 2002.