VCSTC: Virtual Cyber Security Testing Capability – An Application Oriented Paradigm for Network Infrastructure Protection

Guoqiang Shu, Dongluo Chen, Zhijun Liu, Na Li, Lifeng Sang, and David Lee

Department of Computer Science and Engineering, the Ohio State University
Columbus, OH 43210, USA
{shug,chendon,liuzh,lina,sangl,lee}@cse.ohio-state.edu

Abstract. Network security devices are becoming more sophisticated and so are the testing processes. Traditional network testbeds face challenges in terms of fidelity, scalability and complexity of security features. In this paper we propose a new methodology of testing security devices using network virtualization techniques, and present an integrated solution, including network emulation, test case specification and automated test execution. Our hybrid network emulation scheme provides high fidelity by host virtualization and scalability by lightweight protocol stack emulation. We also develop an intermediate level test case description language that is suitable for security tests at various network protocol layers and that can be executed automatically on the emulated network. The methodology presented in this paper has been implemented and integrated into a security infrastructure testing system for US Department of Defense and we report the experimental results.

Keywords: Network Modeling, Network Emulation, Security Testing, Test Automation, Virtualization.

1 Introduction

Security, reliability and interoperability are indispensable in today's distributed heterogeneous information infrastructures. These properties rely on the correct functioning of the increasingly complicated security devices, such as traditional firewall, security switch, intrusion detection system (IDS) and so on [9,11,13,17]. For modern security devices, testing is no longer considered to involve only the vendor because software components such as user configuration and plug-in have become significant [1,21]. On the other hand, since critical secure devices are often to be deployed at sensitive environment (e.g. government or military network), it is not appropriate to test them using the real network. Instead, target system and configuration are tested using special testbed before deployment. The last several decades have witnessed a great number of mature IP network testbed solutions [2,10], with the focus of integration testing and performance testing. In this work we study several key challenges and solutions particularly in testing network security devices.

First, the nature of security testing demands a high level of fidelity between the testbed and the real environment in order to compose realistic test scenario and obtain meaningful assessment. This cannot be achieved by the content-insensitive traffic generation paradigm often adopted in performance testing. Particularly, it is desirable that the testbed mimics the characteristics of the real network including topology, host machine properties and the protocol stack. Fidelity of protocol stack is especially important for (1) generation of realistic background traffic [20], and (2) designing test cases at transport or higher layer, or executing real applications.

While duplicating the real environment or approaches of this nature guarantees fidelity, it could be extremely expensive to scale. An enterprise network normally contains at least hundreds of physical hosts with heterogeneous configurations. Naturally a question to ask here is how many hosts will be involved in a test? While testing for properties like address blocking may require only a few, other features such as resilience against Distributed Deny-of-Service (DDoS) attack could involve much more. A promising direction toward loyal and scalable solution is to employ rapidly developing virtualization techniques [16,19]. Virtual machine solutions such as VMware ESX server can increase the testbed size roughly by an order while still preserving all the applications; and lighterweight protocol stack virtualization methods can scale much better (e.g. virtual Honeynet [14,15]) at the cost of sacrificing some real applications. In this work we propose the integration of both based on the following assumption: even in a test case involving a large number of hosts, the subset that has to run real application simultaneously is often very small. Our experience of developing a firewall/IDS test suite justifies this assumption.

The last but not least notable issue is automation. Security tests usually employ precisely specified sequence of actions from various principals, which essentially requires coordination of the external network, internal network and the device under test itself. The test system should hide this control complicacy to the end user. In addition, as the security features of sophisticated device span over multiple network layers, the test description mechanism should provide corresponding capability and at the same time facilitate automatic test execution.

Motivated by the above insights we propose Virtual Cyber Security Testing Capability (VCSTC) – a novel methodology of testing security device and the associated application solutions with high fidelity, scalability and usability. VCSTC methodology aims at a broad category of target systems that could be deployed at the boundary (gateway) of a local, usually an organizational or enterprise network. The main security-related functionality of such systems includes multi-directional access control, intrusion detection, virus/worm detection, vulnerability analysis and mitigation. Examples of such devices available on the market include Cisco ASA family, Top Layer Secure Command and many lower-end consumer security appliances. Two networks are involved in using and hence testing such devices: an internal network to be protected, and an untrusted external network. Typically there are four steps in testing: create a model for both internal and external network; emulate the two models on a testbed; develop test cases; and execute the test cases on the testbed. VCSTC methodology spans over all four steps, although model construction step is not related to testing and therefore is not the focus on this paper. To the best of our knowledge our approach is the first to integrate network emulation and automated testing, and it is distinguished from the existing security testbed solutions (such as DETER [2]) by the following two key aspects.

Hybrid network emulation: To test a security device deployed at network boundary, both internal and external network are emulated using the mixture of network host and protocol stack virtualization techniques. The emulated network contains two types of nodes: a small number of FAT nodes that are fully featured virtualized network hosts, and a large number of THIN nodes each having only a virtual TCP/IP stack. Configuration of emulated networks is automatically done according to the user network model. Hosts involved in a test case will be mapped to emulated nodes (FAT or THIN) depending on what is executed on that host. By leveraging the advantage of these two virtualization methods, our hybrid testbed achieves the best balance between loyalty and scalability. We show that our methodology can support up to one thousand emulated nodes on a commodity computer.

Test description language: In order to cater the diverse features of security device VCSTC uses an intermediate level test case description language to facilitate the specification of Point of Control and Observation (PCO) at various layers: IP, Socket and Application. PCOs could be deployed at any host and the target device, while the actually deployment and control are automatic. This language is tightly based on a programming language to provide virtually unlimited expressivity. A test case is dynamically compiled into a native executable before execution by the test driver. We evaluate this scheme by both manual test case generation and using the language as the target of model-based formal test generation methods.

The rest of this paper is organized as follows. In section 2 we provide an overview of VCSTC methodology as well as the testing system architecture. Then the two main contributions are discussed in more details. Section 3 introduces our test description language; the hybrid network emulation approach is elaborated in Section 4. The second part of the paper reports our extensive evaluation of the methodology during the development of a testing platform for the U.S. Department of Defense (DoD) [13]. Section 5 presents our experience of manual and automatic test generation, as well as a brief remark on the performance. Finally we discuss some on-going and future work in Section 6.

2 Testing Methodology and System Architecture

In VCSTC methodology there are two essential components in security testing: model and test cases. They are independent and developed separately. The network model should contain sufficient information to emulate a real network, and in the meantime not associated with any special devices and therefore generally reusable. VCSTC supports several methods to build a network model: it could be automatically synthesized using network management protocols (e.g. SNMP) and collected network traces, or using random network topology generation; or manually using prevalent modeling language such as UML with software tool assistance.

Test cases could also be constructed through various ways. A test case mainly specifies two things: (1) a set of PCOs and their deployment (2) a sequence of actions of the PCOs with the expected outcome. Test cases could be made abstract by defining parameters of numerical type or special type like IP address. Such abstract test cases could be concretized by selecting a set of parameter values. To execute a test on a given network model (assuming they are compatible) we first compile it together with all supporting libraries into an executable. Next the network model is automatically emulated and PCOs are deployed at the designated hosts, each of which is controlled by the test driver through a private communication channel. After the test case finishes a log file with all network activities during its execution is returned to the tester along with the test verdict for evaluation. In this framework high fault coverage can be achieved by the combination of three approaches - selecting different network models; generating test cases from a model of the feature under test to cover it more comprehensively; and selecting many combinations of parameter value for an abstract test case.

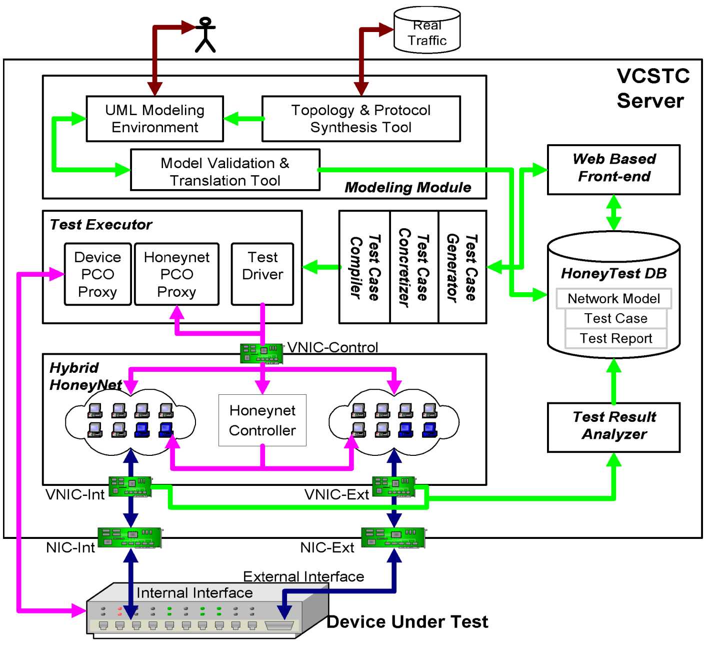

Fig. 1. Architecture of VCSTC platform capsulated within a single server that connects to both internal and external interface of a target device

Figure 1 shows the architecture of a full-featured testing system we developed that realizes the VCSTC methodology. The system is an out-of-box product that could be contained within a single server. The modeling module provides an UML compatible environment for creating and validating network models. Models are stored in a database after validation. The operational environment implements a streamline consisting test preparation, test execution and test result processing, all of which are exposed to the user via a Web-based interface. The test generation module accepts abstract test cases (as textual file) generated by the tester or from a formal model. They are then concretized based on a certain parameter selection policy and eventually compiled into a native binary file. The test executor is responsible of creating emulated network and executing test cases on it. The emulated network is essentially a virtual honeynet with hybrid nodes (details in Section 4) implemented using a pool of virtual machines with a central controller. The whole testing workflow is automated to the extent that testers interact with VCSTC server only by providing network models and test cases from Web interface. The test executor hides the complexity of controlling the network emulator and PCOs from the users.

Now we briefly describe the network configuration on the server. Three types of virtualized network interfaces connect the emulator with other components. There are many virtual Network Interface Cards (NIC) of each type grouped together by virtual networks. The private interface VNIC-Control is used by the test executor to control the honeynet and all PCOs. This type of interface is totally invisible to the test cases and the target device. The other two types of interfaces VNIC-Int and VNIC-Ext are bridged with the physical NICs NIC-Int and NIC-Ext on the server, which are connected to the internal port and external port of the target device, respectively. This setup enables emulation of both the internal and the external network, while all emulated packets generated during testing are transformed into real packets at the corresponding VNIC before delivered to the device. Note that for simplicity Figure 1 only shows one internal path to the device whereas additional NIC and VNIC could be deployed similarly according to the requirements for testing. Network traffic - both real and emulated - passing all VNIC will be monitored during test execution and readable test reports such as Message Sequence Chart (MSC) are generated afterwards for further analysis.

3 Test Specification Language

VCSTC uses its own notation for test case specification. The rationale of introducing the new notation is not to replace the traditional high level test specification language such as TTCN-3 or MSC; instead our main incentive is to provide a flexible way of developing test sequences related to security features at all layers of network protocol stack. Toward this goal our language is tightly based on a native programming language such that any valid statement of the host programming language could be embedded in the test code, providing encoding of an input symbol, for instance. A test case is a textual file with multiple declarative sections (described below) and a test code section. We show later that the proposed intermediate level language could be used to interpret test sequences from more abstract formal models such as EFSM [6].

The most important element in our language is PCO. A test case defines multiple PCOs on various places and controls their behavior. A PCO is deployed on either a host of internal/external network or the target device. Every PCO on network has one of three types: (1) Packet PCO sends and receives raw network packet of TCP, UDP or IP protocol by taking over the network interface of the host. (2) Socket PCO manages one TCP or UDP socket. (3) Application PCO handles one user application. It reads and writes to the application through its standard input/output channel. Table 1 summarizes the three types of PCO. A PCO must be bound on a host (i.e. an IP address) before it can be function, and the mapping could be done by various ways. The PCO definition might supply a fixed IP address if the test case is design for some specific network models, or otherwise the binding could be done in test code by calling run-time API. Note that multiple PCOs with different types could co-exist on the same host except for Packet PCO due to the nondeterministic behavior under the situation of multiple network capturers.

Table 1. Three types of PCOs

Type of PCO	Packet PCO	Socket PCO	App. PCO
Method of control	Send/Receive raw TCP/UDP/IP packet	Read/Write TCP/UDP socket	Execute native application
Number on each node	One	Many	Many
Blocking I/O	No	Yes	No

The actions of PCOs are defined in the test code section, which eventually returns a test verdict. A test case could contain parameters and become abstract. Abstract test case cannot be executed before assigning parameter values. Our language supports parameters of bounded Integer type and IP address type, and a test concretization algorithm implements parameter value selection according to certain coverage criteria such as random sampling, boundary coverage and so forth. After the test case is concretized it is automatically transformed into the host language for compilation. In this phase auxiliary code such as test case initialization and cleanup routines is generated and weaved together with the test code. During compilation the VCSTC runtime library proving essential functionalities and all user defined libraries are linked. In practice, a lot of reusable routines (e.g. Malware simulation, special packet generation) are encapsulated in the form of library so that the test code could focus on the logic, that is, the sequence of actions from PCO. The VCSTC runtime library implements all types of PCO and the proxies used to control them remotely from the test executor.

Now we discuss more details by an example. Figure 2 shows a test case that checks whether a security device provides an outgoing source IP black list of sufficient length. The test code uses Java as host language and implements a rather straightforward logic: a Web server is started on an external host by an Application PCO (line 8). There are many (NCLIENT) internal hosts with a Socket PCO on each (line 7). Both the server and clients are selected randomly from the network (lines 17-18). The device is controlled by a Device PCO (line 9). At the beginning the test case launches the server and clears the black list (line 22), followed by a check (lines 24-28) to see whether all clients can reach the server. Next the black list is configured through Device PCO (line 29), and we retry the connections again, expecting that no client can successfully reach the serer. Any client's success in connecting at this time (line 32) proves the black list useless and therefore the test case returns the verdict failure. The test case contains two Integer parameters (lines 2-5): number of clients and server ports, which are to be assigned according to a user policy. Note that the execution of the test case is fully automated except for Device PCO. Configuration of target device might need manual activity, depending on the interface a specific device is providing. Many venders provide programmable configuration mechanism, which could be utilized by VCSTC runtime to fully automate test execution.

We close this section by a remark on the relationship between test case and network model. Test cases like the one in Figure 2 do not depend on any network-specific properties such as background traffic and therefore could be executed on any network models. The only implicit constraint is that the network must contain enough (in this case NCLIENT) distinct hosts – this will be checked statically by the test executor when loading the test case. On the other hand, the test case might also explicitly specify a list of compatible network model names if necessary.

```
#TESTCASE OBL_Length_TCP
#PARAM {
    int{[0,512]} NCLIENTS;
    int{[0,1024], [50000,51000]} PORT;
}
#PCO {
    SOCKET pco_client[NCLIENTS];
    APP pco_server;
    DEVICE pco_device;
}
#PACKET {}
#VERDICT {
    success, failure, unknown, timeout
}
#TESTBODY
{
   bind_PCO(pco_client, INTERNAL, RANDOM|NONDUP);
   bind_PCO(pco_server, EXTERNAL, RANDOM);

   log("Testing length of black-list using TCP");
   pco_server.execute("httpd", PORT);
   pco_device.config("clear black list");
   Vector black_list = new Vector<InetAddress>();
   for (int x= 0;x<NCLIENTS;x++) {
           if(!pco_client[x].connect
                   (pco_server.getIP(),PORT)) return unknown;
           black_list.add(pco_client[x].getIPAddress());
           pco_client[x].close();
   }
   pco_device.config("add to black list", black_list);
   wait(3000);
   for (int x= 0;x<NCLIENTS;x++) {
           if(pco_client[x].connect(pco_server.getIP(),
                   PORT)) return failure;
   }
   return success;
}
```

Fig. 2. Example of a simplified abstract test case

4 Network Host and Protocol Stack Virtualization

A test case is executed on an emulated network that from the view of the device under test is the same as a real network. Emulated network is created from a network model using hybrid network virtualization approach. As we mention in Section 1, VCSTC mitigates the key challenge of scalability by using a hybrid virtual honeynet. Honeynet [18] is used recently as a best practice of network emulation for the purpose of attack identification. We adapt a hybrid honeynet design where two virtualization techniques are used together to achieve the balance of scalability and fidelity. First we distinguish two terms used in this section: *logical* node and *physical* node. A logical node is a network host, either external or internal, defined in a test case. A logical node is identified by its IP address. A physical node is a network host in the emulated network. The test executor maintains a mapping from logical nodes to physical nodes and deploys the PCOs according to this mapping. In our scheme of hybrid honeynet there are two types of physical nodes in the emulated network:

- **FAT node:** A FAT physical node is emulated by a complete virtualized host machine. Thanks to the advanced virtualization techniques such node can accommodate any application running at the real host. A repository of pre-configured (e.g. with different Operating System and/or applications) virtual machine images are stored at the server while the honeynet controller selects the proper ones to load. Unfortunately, host virtualization is still very expensive and we cannot afford to emulate the whole network using FAT nodes alone.

- **THIN node:** A THIN physical node is emulated by virtualizing only a TCP/IP protocol stack but not the actual resources of a host. Software solutions such as Honeyd [14] accomplish this by overriding the IP protocol stack on a single (possibly virtual) machine and claiming responsibility for a range of IP addresses. Socket based program could be executed on top of the virtualized protocol stack appearing to the outside as running with its own address. This approach is lightweight and therefore very scalable. The cost however, is that the function of PCO deployed on them is limited. Since all programs launch on THIN nodes share the same physical machine and therefore its resources, there is obviously a potential problem of interference. The exact constraints are determined by the virtualization tool used. In our system with Honeyd as protocol stack emulator if a logical node is mapped to a THIN node, then application PCOs on it can only execute a special type of socket-based EFSM simulation program synthesized from user network traces (see Section 6).

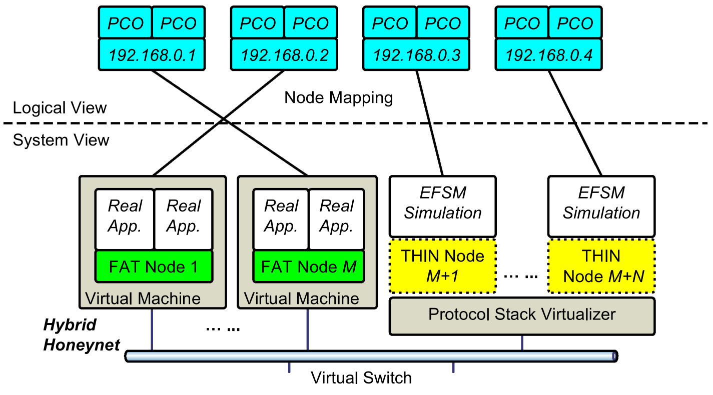

Fig. 3. Internal structure of a hybrid honeynet with two types of nodes

We create and configure a mixture of these two types of physical nodes in the emulated network, as shown in Figure 3. From the view of the emulator, we have a small number *M* of FAT nodes and a protocol stack virtualizer supporting *N* THIN nodes. These heterogeneous physical nodes are connected by a virtual network switch and form a honeynet. On the other hand, the heterogeneity is made transparent to the test cases. That is, all logical nodes are the same in terms of PCO capabilities. The mapping between logical and physical node is first created by the test executor before a test case is loaded, and is adjusted dynamically by network reconfiguration under some circumstances. The separation of logical nodes and physical nodes has two obvious advantages. First the honeynet resource provisioning could be changed anytime - for example adding more FAT nodes - without affecting any test case. Second, in most test cases only a small number of hosts run real applications (and therefore require to be mapped to a FAT node) simultaneously, despite that the total number of hosts is large. In such situations when a logical node does not have any activities we can remap it to a THIN node at runtime. When the current physical node provisioning can no longer support the execution of a test case, the test executor will get a runtime error and hence returns failure. Below we describe some heuristic guidelines practiced by the test executor for static and dynamic mapping.

Guideline 1 (Static): If a logical node does not have Application PCO, always map it to a THIN node, because Packet and Socket PCOs could both be supported.

Guideline 2 (Static): If a logical node has both Socket and Application PCO, map it to a FAT node if there is one available, otherwise map to a THIN node. Nodes with Application and Packet PCO have lower priority of mapping to FAT node. This is because Packet PCO is easier to migrate dynamically than Socket PCO.

Guideline 3 (Dynamic): Before an Application PCO executes a real user application, mapping need to be adjusted if the logical node is currently mapped to a THIN node. If there is Socket PCO with established TCP connections at this time, we report failure because we cannot migrate TCP connection across physical nodes. Otherwise, if there is an unmapped FAT node, it is remapped to the logical node. If all FAT nodes are already mapped, we check whether one of them could be swapped to a THIN node, that is, on the current owning logical node no Application PCO is executing and no Socket PCO is connected. If this condition is satisfied, honeynet controller will reconfigure the network (i.e. IP address) and switch the mapping of two logical nodes, therefore allow the user application to be executed on a FAT node. If no FAT node satisfies this condition, failure is reported.

As an example of test case, in Figure 2 we have an array of logical nodes (client) with only Socket PCO and another node (server) with only Application PCO. The mapping for this test case is trivial since only one FAT node is needed for the server node and all clients are mapped to THIN nodes.

Figure 4 shows a more illustrative example. In this test case we have two client nodes with Application PCO and a server node with Application PCO. Table 2 shows the node mapping at several key timing points when the emulated network contains unlimited THIN nodes but only 2 FAT nodes. Before executing the test case, the first two nodes (client[0] and client[1]) get the FAT nodes and the rest (including server[0]) get THIN nodes. Before the server starts (line 13), it needs to be remapped to a FAT node, and client[0] could be swapped out since it is not active. Similarly when client[0] needs to launch its program reconfiguration happens again, swapping it with client[1]. Finally client[1] launches a program, now since client[0]'s PCO has terminated its application, it could be switched to a THIN node and client[1] gets the FAT node. Note that the jitter of mapping in this example is quite unrealistic since in practice the server contains much more FAT nodes.

```
1.  ……
2.  #PCO {
3.          APP pco_client[4];
4.          APP pco_server;
5.          DEVICE pco_device;
6.  }
7.  ……
8.  #TESTBODY
9.  {
10.  bind_PCO(pco_client, INTERNAL, RANDOM|NONDUP);
11.  bind_PCO(pco_server, EXTERNAL, RANDOM);
12.  ……
13.  pco_server.execute_service("IIS6.0");
14.  ……
15.  pco_client[0].execute("lynx","domain.com/page.cgi");
16.  ……
17.  pco_client[0].terminate();
18.  ……
19.  pco_client[1].execute("iexplore","domain.com/page.cgi");
20.  ……
21.  pco_client[1].terminate();
22.  ……
23. pco_server.terminate();
24. }
```

Fig. 4. Example of test case with dynamic node remapping

Table 2. Node mapping of the test case with 2 FAT nodes

	Line 10	Line 13	Line 17	Line 19
server	THIN	FAT-1	FAT-1	FAT-1
client[0]	FAT-1	THIN	FAT-2	THIN
client[1]	FAT-2	FAT-2	THIN	FAT-2

Dynamic network reconfiguration also involves a reconnection between the PCO proxy (in the test executor) and the physical node through the network interface VNIC-Control of the honeynet (Figure 1). When the new IP address becomes usable on the physical node, the PCO proxy will disconnects the old PCO and connect the new one. On a separate issue, we are currently investigating suitable process migration schemes supporting dynamic remapping including live TCP connections, which fully take the advantage of the hybrid network design.

5 Experiments and Evaluation

The proposed VCSTC methodology has been fully applied in the development of a real security testing platform for the U.S. DoD (Department of Defense). The purpose of this project is to provide critical network infrastructure owners with an effective and easy-to-use mechanism to assess the suitability of a security device or solution with respect to their own infrastructure before investment. In this section we report our experience and evaluation during the development of this platform. We start from a brief overview of the system configuration and some simple practice in Section 5.1; then Section 5.2 summarizes our effort of integrating automatic test generation techniques. Our system supports generating test cases (in our test description language) from two popular formal models – Parameterized EFSM and Simplified Firewall Rule Language. We also present performance evaluation of the system installed on commodity hardware in order to justify its feasibility and scalability.

5.1 System Configuration and Basic Operations

As discussed earlier, the whole system could be deployed on a single machine, i.e. HoneyNet server (Figure 1). We choose a typical hardware configuration: a Dell Precision 690 workstation with two Xeon 3.2 GHz Due Core CPUs and 2GB memory. The server has two Gigabit physical NICs (NIC-Int and NIC-Ext). Both modeling module and test executor are implemented in Java 1.5 and Jpcap (a packet manipulation utility). The hybrid honeynet is composed of 5 VMware virtual machines running Ubuntu Linux as guest Operating System – 4 of them with 256MB virtual memory each are used as FAT nodes and the last one with 512MB virtual memory runs Honeyd 1.5 to emulate up to 1024 THIN nodes. The system is used to test several security devices on the market, and our performance evaluation is conducted using Netgear ProSafe FVS318 VPN Firewall/Switch.

We use both network models synthesized from real network and randomly generated large models. For real network, we derive a model from a testbed of the WAN-in-Lab project [7] developed by Caltech. This testbed has 4 Cisco routers with SNMP capability. The whole model contains 39 subnets and totally about 40 distinguished hosts with services available. We imagine the target device is about to be deployed at the gateway of this network and manually develop a small test suite that covers the classic access control and content filtering features common to typical Firewall and IDS. It takes a Java developer two days after one day's training to write about 50 test cases (Table 3). Using these test cases, the tester is able to verify precisely the details of many features of the device that is stated very informally and vaguely from its user manual. For instance, one of the Anti-virus test cases discovers that the device cannot enforce malicious URL blocking when the URL is encoded in HEX form (e.g. "www.abc.com/x.e%78e" for "www.abc.com/x.exe"), which effectively renders this URL blocking feature useless. Based on this experience we consider our test description language efficient and of good usability.

Table 3. Firewall/IDS Test Suite

Firewall Feature	Inbound filtering	24 Test cases
	Outbound filtering	24 Test cases
	Port Forwarding	4 Test cases
	Dynamic filtering	1 Test cases
Anti-virus features		2 Test cases
Intrusion detection features		3 Test cases

5.2 Automatic Test Case Generation

The applicability of our methodology could be broadened by leveraging the advanced test generation methods. We make an effort to integrate them into our methodology. We investigate automated translation from test sequences derived from formal model to the VCSTC test description language. The first model we implement is EFSM. Our test system provides a GUI to specify a feature of the device using EFSM with parameters in I/O message. Figure 5 shows an example of a single port blocking feature with two states. From the EFSM model test sequences could be automatically derived using various approaches, such as checking sequences from reachability analysis (Figure 5 shows the reachability graph when the range of *port* variable has 3 values). We then translate each test sequence into a test case file and then generate an incomplete user library that defines the I/O symbols of the model. Figure 6 shows a section of the test case corresponding to the sequence {*Set_Block*[0]/-, *Visit*[0]/-, *Set_Unblock*/-, *Visit*[0]/*Resp*} and an empty method definition for input symbol *Set_Block*, for which the test designer is responsible of providing the code to implement this input symbol using the PCOs on internal and external hosts. Note this only needs to be done once and then shared by all test cases for the same model.

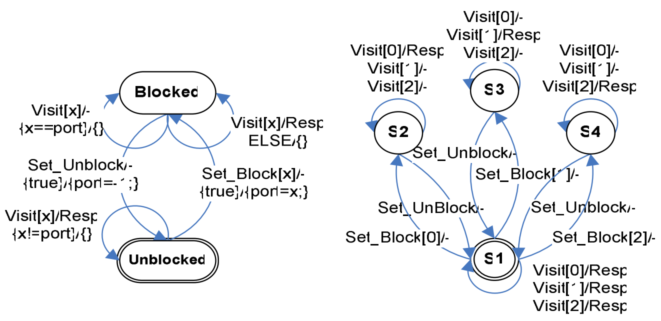

Fig. 5. A simple EFSM model of port blocking feature (left) and the corresponding reachability graph as a FSM (right)

Similarly our system supports generating test cases from firewall configurations. We use a simple grammar to describe firewall rules following classic semantics [1,11]. A rule contains a predicate based packet filter and an action and a configuration is an ordered list of rules. After a user inputs a firewall configuration, test cases with input packets are automatically generated. The elements in the packet filter can be either a value or a wildcard ("*"), and furthermore the user might ask the test case concretization process to select a value by specifying it as a parameter of the rule. For example, a configuration as specified below is composed of two rules *A* and *B*. The test case generated from this configuration will contain three parameters (i.e. Src port

```
#TESTBODY
{
    input_Set_Block(pco_ext, pco_int, pco_device,0);
    ASSERT(output(pco_ext, pco_int, pco_device) == NULL);
    input_Visit(pco_ext, pco_int, pco_device,0);
    ASSERT(output(pco_ext, pco_int, pco_device) == NULL);
    input_Set_Unblock(pco_ext, pco_int, pco_device);
    ASSERT(output(pco_ext, pco_int, pco_device) == NULL);
    input_Visit(pco_ext, pco_int, pco_device,0);
    ASSERT(output(pco_ext, pco_int, pco_device) == Resp);
    return success;
}
#USES LIB_Simple_Port_Blocking

public class LIB_Simple_Port_Blocking
{
    ……
    public void input_Set_Block(PCO pco_ext, PCO pco_int,      PCO
    pco_device) {
            … //user provides implementation of input symbol;
    }
    ……
}
```

Fig. 6. Test code and library generated from EFSM test sequences

in *A*, Protocol in *B* and Src port in *B*) and a pair of Socket PCO binding on internal and external network, respectively.

A: "Allow TCP from [10.0.0.1:$Param_1$] to [*.*] through External"
B: "Deny $Param_2$ from [*:$Param_3$] to [192.168.0.2:80] through External"

The test code first enables this configuration through Device PCO, and then essentially sends a packet enabling a subset of rules to see whether the device under test takes the expected action. Clearly the subset of rules triggered by a particular packet depends on the parameter value of all rules. In our example, A and B could be enabled together if $Param_2$=TCP and $Param_1 = Param_3$. In fact when both *A* and *B* are enabled they conflict with each other, and it is to the interest of the tester how the device will handle. The test case concretization process produces parameter assignments in such a way that most rule subsets are covered. Due to space limit we omit the detail of the algorithm and test case generated. From our experiences of integrating the two formal models, we believe that our methodology is promising for a variety of application domains related to network security testing.

5.3 Performance Evaluation

Finally we remark on the performance evaluation of our system. First we clarify that VCSTC is not targeted for performance testing or load testing therefore it is not designed to meet hard real-time requirements. The purpose of evaluation is instead to justify the feasibility of our design for network model and tests of practical scale. We use a series of micro-benchmarks to measure various aspects of the system, with the focus on the performance penalty incurred by using hybrid network virtualization. The first performance penalty comes from initializing the emulated network. For FAT

# of THIN Node	16	64	256	1024
Startup Time	6.67s	24.13s	30.59s	55.43s

(a)

Message Size	4KB	8KB	16KB	32KB	64KB
Trans. Time	<1ms	1.00ms	2.13ms	3.88ms	7.87ms

(b)

# Connections	1	2	3	4
Bandwidth (each con.)	6.80Mb/s	2.95Mb/s	2.08 Mb/s	1.90 Mb/s

(c)

Fig. 7. (a) Startup time of hybrid honeynet with different network size. (b) Transmission time of PCO control messages. (c) TCP transmission bandwidth between external and internal network nodes with 1-4 simultaneous connections measured by iPerf.

nodes the controller reset/reload loads a virtual machine image which takes constant time; then the Honeyd engine virtualizes the pool of THIN nodes and launch the PCO on each node. The Honeyd start up time is proportional to the number of THIN nodes as shown in Figure 7 (a), for instance a network of 1024 nodes could take up to 1 minute to initialize. Note that under certain situations it is unnecessary to reinitialize network for each test case, specifically when all test cases share a network model and they all cleanup properly. In addition, communication cost between the test driver and the PCO is not neglectable because the control message sent might carry a data portion (e.g. a packet to send from that PCO). We measure the transmission time with various message sizes shown as Figure 7 (b). There is no difference between FAT and THIN nodes since the same control channel is used.

The packet dispatching mechanism used by protocol stack virtualization tools (i.e. Honeyd) also causes delay in data transmission involving a THIN node. Basically all socket function calls are delegated to the tool and go through internal tunneling, which forms a global bottleneck. We use a benchmarking tool iPerf to measure the bandwidth of concurrent TCP connections between external and internal nodes. If both are FAT nodes the bandwidth for a single link is 8.89Mb/s; if one side is THIN node, it is downgraded as shown in Figure 7(c). We believe that this bandwidth limitation is not critical to validity of most security related tests.

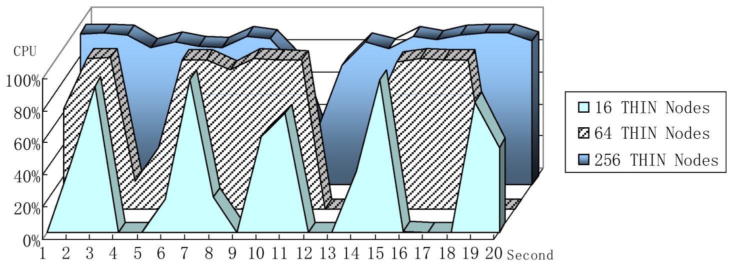

Fig. 8. Honeyd CPU load percentage for three networks of THIN nodes each sending UDP traffic at 2KB/s

Another simple benchmark is designed to evaluate approximately the work load of testing server. The dominating factor here again is large number of THIN nodes virtualized by Honeyd. We create network of different size, then let each node send UDP packet at a given rate to random destination node. This scenario corresponds to a typical test case where all logical nodes carry symmetric tasks. Figure 8 shows the CPU usage of the VMware guest OS running Honeyd during a window of 20 seconds. When the network is small (16 nodes) an average 33.4% CPU time is used while a large network (1024 nodes) is likely to saturate the CPU (85.8%). While admittedly being a coarse measurement, this shows that our system is capable of running fairly large models.

6 Discussion

In this work we present a new security testing approach, VCSTC using network host and protocol stack virtualization. Two main aspects are discussed in detail: (1) designing a scalable and yet loyal network testbed; (2) develop test cases manually or automatically. Compared to existing solutions, VCSTC has a few advantages. Our novel design of hybrid network emulation provides both fidelity (by network host virtualization) and scalability (by lightweight protocol stack virtualization). We also develop an intermediate level test description language that is suitable for security tests at various network protocol layers. In the paper we discuss how test cases are executed automatically on the emulated network model. Extensive experiments have been conducted on our implementation platform, which justify the benefits of our proposed methodology.

On the other hand, we are still at the initial stage of applying network virtualization techniques to testing. Lots of issues remain to be explored in our current approach before its applicability could be further broadened. Our approach aims at security testing at IP layer and above. As a matter of fact, VCSTC does not support routing protocol emulation despite that it generates real IP packets. Consequently, routing related security features cannot be tested under our framework. For similar reason data link layer security features are not supported. Emulating routing in a scalable fashion is a challenging task and it may change the protocol stack virtualization scheme in a drastic way. A promising approach is to use one virtualizer for each routing domain or subnet, and connect them by FAT nodes where routing protocols are implemented. Also the test language is to be augmented to support routing operations at the PCOs.

Protocol synthesis from real network is another challenge where network traffic with high fidelity is desired. This is an issue for both network modeling and testbed design. Since running real user applications on top of all virtualized nodes is clearly not practical, we need to synthesize a model of the protocol from network traces [3,12] and emulate it on the testbed in order to generate (not simply replay) traffic patterns similar to those seen. In our ongoing work we use a state machine minimization approach [5] to obtain EFSM models from field-decoded protocols (e.g. by Ethereal), and implement a special program to simulate EFSM models that could be executed on top of both FAT and THIN nodes. We envision this and the enhancement for routing emulation will render our VCSTC a more powerful and useful tool for testing both hardware and software based network security systems.

Acknowledgement. The VCSTC project was supported by the U.S. Department of Defense under grant award N41756-06-C-5541. Work in this paper is also supported in part by the U.S. National Science Foundation (NSF) under grant awards CNS-0403342, CNS-0548403. We thank Steven Low from Caltech for kindly allowing us to use the WAN-in-Lab testbed.

References

1. Al-Shaer, E., Hamed, H.: Discovery of Policy Anomalies in Distributed Firewalls. In: Proceedings of IEEE INFOCOM (2004)
2. Benzel, T., Braden, R., Kim, D., Neuman, B., Joseph, A., Sklower, K., Ostrenga, R., Schwab, S.: Experience with DETER: A Testbed for Security Research. In: 2nd IEEE Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom) (2006)
3. Cui, W., Kannan, J., Wang, H.: Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In: The 16th USENIX Security Symposium (2007)
4. El-Atawy, A., Ibrahim, K., Hamed, H., Al-Shaer, E.: Policy Segmentation for Intelligent Firewall Testing. In: 1st Workshop on Secure Network Protocols (NPSec) (2005)
5. Gören, S., Ferguson, F.J.: On state reduction of incompletely specified finite state machines. Computers and Electrical Engineering 33(1), 58–69 (2007)
6. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - A survey. In: Proceedings of the IEEE, pp. 1090–1123 (1996)
7. Lee, G.S., Andrew, L.H., Tang, A., Low, S.H.: WAN-in-Lab: Motivation, Deployment and Experiments. Protocols for Fast, Long Distance Networks (PFLDnet), 85–90 (2007)
8. Liljenstam, M., Nicol, D.M., Berk, V., Gray, R.: Simulating Realistic Network Worm Traffic for Worm Warning System Design and Testing. In: Proceedings of the 2003 Workshop on Rapid Malcode (WORM) (2003)
9. Lyu, M., Lau, L.: Firewall security: policies, testing and performance evaluation. In: Proceedings of the COMSAC, pp. 116–121 (2000)
10. Maier, S., Herrscher, D., Rothermel, K.: Experiences with node virtualization for scalable network emulation. Computer Communication 30(5), 943–956 (2007)
11. Mayer, A., Wool, A., Ziskind, E.: Fang: A Firewall Analysis Engine. In: Proceedings of the IEEE Symposium on Security and Privacy (2000)
12. Orebaugh, A., Ramirez, G., Burke, J., Pesce, L.: Wireshark & Ethereal Network Protocol Analyzer Toolkit (Jay Beale's Open Source Security). Syngress Publishing (2007)
13. Pederson, P., Lee, D., Shu, G., Chen, D., Liu, Z., Li, N., Sang, L.: Virtual Cyber-Security Testing Capability for Large Scale Distributed Information Infrastructure Protection (submitted, 2008)
14. Provos, N., Holz, T.: Virtual Honeypots: From Botnet Tracking to Intrusion Detection, 1st edn. Addison-Wesley Professional, Reading (2007)
15. Provos, N.: A Virtual Honeypot Framework. In: Proceedings of the 13th USENIX. Security Symposium (2004)
16. Sabiguero, A., Baire, A., Boutet, A., Viho, C.: Virtualized Interoperability Testing: Application to IPv6 Network Mobility. In: 18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, pp. 187–190 (2007)
17. Sherwood, J.: The Security Certification Criteria Project. In: The 3rd International Common Criteria Conference (2002)
18. Spitzner, L.: The Honeynet Project: Trapping the Hackers. IEEE Security and Privacy 1(2), 15–23 (2003)
19. VMware Inc, http://www.vmware.com
20. Wang, L., Ellis, C., Yin, W., Luong, D.D.: Hercules: An Environment for Large-Scale Enterprise Infrastructure Testing. In: Proceedings of the Workshop on Advances and Innovations in Systems Testing (2007)
21. Wool, A.: Architecting the Lumeta firewall analyzer. In: 10th USENIX Security Symposium, pp. 85–97 (2001)