IEICE TRANS. INF. & SYST., VOL.E94–D, NO.5 MAY 2011

PAPER

A State-Aware Protocol Fuzzer Based on Application-Layer Protocols

Takahisa KITAGAWA[†a)], **Miyuki HANAOKA**[†], *Nonmembers*, *and* **Kenji KONO**[†,††], *Member*

SUMMARY In the face of constant malicious attacks to network-connected software systems, software vulnerabilities need to be discovered early in the development phase. In this paper, we present AspFuzz, a state-aware protocol fuzzer based on the specifications of application-layer protocols. AspFuzz automatically generates anomalous messages that exploit possible vulnerabilities. The key observation behind AspFuzz is that most attack messages violate the strict specifications of application-layer protocols. For example, they do not conform to the rigid format or syntax required of each message. In addition, some attack messages ignore the protocol states and have incorrect orders of messages. AspFuzz automatically generates a large number of anomalous messages that deliberately violate the specifications of application-layer protocols. To demonstrate the effectiveness of AspFuzz, we conducted experiments with POP3 and HTTP servers. With AspFuzz, we can discover 20 reported and 1 previously unknown vulnerabilities for POP3 servers and 25 reported vulnerabilities for HTTP servers. Two vulnerabilities among these can be discovered by the state-awareness of AspFuzz. It can also find a SIP state-related vulnerability.

key words: *fuzzing, software vulnerability testing, protocol specification*

1. Introduction

Internet servers are constantly exposed to malicious attacks launched remotely. For example, attackers may send malicious messages that cause buffer overflows to execute arbitrary malicious code. To defend against malicious attacks, eliminating all vulnerabilities in software systems before they are shipped is essential. In spite of recent advances in software development/testing techniques, product-quality software systems are far from vulnerability-free. Vulnerabilities in software systems have instead increased in number. Symantec [1] documented 5491 vulnerabilities in 2008; this is a 19% increase over the 4625 vulnerabilities documented in 2007.

Fuzzing is a software testing technique to reduce the number of vulnerabilities in software systems. Fuzzing is a kind of black box testing; it prepares random or semi-random inputs to target programs. Since these inputs are random, they are likely to be unexpected and incorrect inputs to the target programs. If the target program does not reject the incorrect inputs, it will hang or crash during fuzzing. Since critical security flaws often lie in incorrect or insufficient checking of program inputs, software systems that cannot survive fuzzing have a high probability of security holes. Fuzzing is known to be a quick and cost-effective approach to finding security flaws; according to the "Month of Browser Bugs" project, a new vulnerability was discovered each day of July 2006 [2] by use of fuzzing.

To apply fuzzing to networked software, network messages must be created carefully because random inputs may not conform to the specifications of application-layer protocols and, if so, are rejected immediately. State-of-the-art fuzzers create input messages from the specifications of application-layer protocols such as POP3 and HTTP. These specifications detail 1) the format and syntax of each message and 2) the order in which messages are exchanged. Usually, the message order can be described in a state automaton. Each state determines a set of messages that can be accepted in that state. When a message is received or sent, the protocol state is changed to another. For fuzzing to be performed on networked software, fuzzers must be aware of protocol states. Since networked software handles different sets of messages in different states, fuzzers should generate anomalous messages according to the current state of the target protocol.

To the authors' knowledge, most existing fuzzers use *message-level* fuzzing [3]–[7]; it adds anomalies at the message level and preserves the correct order of message exchanges. Although message-level fuzzing has been proven effective, it cannot find exploitable vulnerabilities if the message order is anomalous. Suppose a protocol specification says message A must be followed by exactly one message B. Message-level fuzzers generate anomalous versions of messages A and B but cannot generate an incorrect sequence of messages such as B→A and A→B→B. Some networked software has vulnerabilities if the message order is not correct. For example, Apache HTTP Servers 2.0.37 through 2.0.45 are susceptible to remote attacks causing denials of service [8]. This vulnerability is exploited if an HTTP request is followed by multiple `Host` headers, which is prohibited by HTTP/1.1; it defines that there should be only one `Host` header.

To mitigate this shortcoming of message-level fuzzers, scenario-based fuzzers have been proposed [9], [10]. To use scenario-based fuzzers, the user must describe a scenario that specifies the order in which messages are sent during fuzzing. However, writing a scenario is a tedious and daunting task. To increase the coverage of fuzzing, the user must carefully design a scenario that covers multiple patterns of message exchanges.

Manuscript received March 29, 2010.
Manuscript revised November 27, 2010.

[†]The authors are with Keio University, Yokohama-shi, 223–8522 Japan.
[††]The author is with CREST, Japan Science and Technology Agency.
a) E-mail: kita@sslab.ics.keio.ac.jp

DOI: 10.1587/transinf.E94.D.1008

In this paper, we present AspFuzz, a state-level (or state-aware) protocol fuzzer that sends network messages in an anomalous order as well as in an incorrect format or syntax. AspFuzz targets networked software that implements application-layer protocols such as POP3, IMAP, SIP, and HTTP. AspFuzz is a superset of conventional protocol fuzzers; it generates anomalous messages similar to the conventional anomalous messages. In addition, it sends them in an anomalous order. Instead of describing scenarios for fuzzing, AspFuzz automatically re-orders messages to discover vulnerabilities. For effective generation of anomalous messages, AspFuzz leverages the protocol specifications usually published as RFC. To use AspFuzz, a protocol specification is described in a special language for each application-layer protocol. From this specification, AspFuzz automatically generates anomalous attack messages in both correct orders and anomalous orders. Preparing protocol specifications would not be a problem because previously prepared specifications can be reused.

We implemented a prototype of AspFuzz. To demonstrate the effectiveness of AspFuzz, we searched for vulnerabilities of POP3, HTTP, and SIP reported in SecurityFocus [11]. A closer examination showed that AspFuzz can discover 20 vulnerabilities in POP3 servers, 25 vulnerabilities in HTTP servers, and 1 SIP vulnerability. Three vulnerabilities among these are state-related ones. We collected 4 POP servers and 19 HTTP servers with vulnerabilities and tested them using AspFuzz. AspFuzz discovered all 4 reported and 1 unknown vulnerabilities for POP3, and 16 reported vulnerabilities for HTTP. The 3 HTTP vulnerabilities are not exploitable even with exploit code provided by SecurityFocus and neither discovered AspFuzz.

The remainder of this paper is organized as follows. We discuss the related work in the next section. Section 3 presents AspFuzz, and Sect. 4 describes its implementation. Section 5 presents our experimental results. Finally, we conclude the paper in Sect. 6.

2. Related Work

Fuzzing is a software testing technique that provides random data as inputs for software. It is particularly useful for testing network protocols or file formats. The work most related to ours is AJECT [3]. Like AspFuzz, AJECT uses an application-layer protocol specification to generate well-formed input messages. The specification represents the definition of a state, flow graph of the protocols, and message syntax that can be sent in each state. Inputs are generated based on heuristics to find specific vulnerabilities, such as buffer overflow or format string vulnerabilities. In each state, AJECT generates messages that are acceptable in that state but with invalid arguments. Like AJECT, AspFuzz generates messages that violate the specified format or syntax of messages. However, AspFuzz also generates messages in anomalous orders. Some reported vulnerabilities can be exploited by messages in incorrect orders and therefore can be discovered only by AspFuzz. Other fuzzing tools, such as PROTOS [4], Peach [5], Spike [6], and Autodafé [7], use a similar approach to that of AJECT. SNOOZE [9] and KiF [10] are scenario-based stateful protocol fuzzers. The former targets network applications, and the latter targets SIP applications. To generate well-formed input messages, these fuzzers require a scenario in addition to a protocol specification. A scenario specifies how to perform the fuzzing, such as using mutated fields and the order in which messages are sent. Thus, they do not automatically generate inputs from the protocol specification. Because the tester must carefully design a scenario that covers multiple patterns of message exchanges, writing it manually is a tedious and daunting task. Sparks et al. [12] propose a fuzzing approach based on a genetic algorithm to increase code coverage. A tester specifies the grammar that encodes message syntax and input production rules. Since this approach does not generate inputs by considering the protocol state, however, it cannot discover vulnerabilities which are related to the protocol state.

Pistachio [13] performs static source code analysis to check the C implementation of a network protocol. Pistachio uses rules that describe acceptable messages and sequences of messages during each communication, which are defined in RFCs or similar standards document. Pistachio can find bugs in the protocol implementations, but it requires the source code of the program.

Vulnerability assessment tools, designed for discovering known vulnerabilities, include Nessus [14], McAfee Vulnerability Manager [15], COPS [16], Internet Scanners [17], and QualysGuard [18]. These tools collect known vulnerabilities and regularly scan a system to determine whether or not these vulnerabilities exist. Although these tools are useful for improving the security of a system, they cannot discover unknown vulnerabilities.

To automatically generate test inputs, some approaches use symbolic execution and constraint solving [19]–[25]. For example, EXE [19] and KLEE [20] perform symbolic execution at a source code level. Godefroid et al. [21], [22] propose a whitebox fuzzing that performs symbolic execution of binary programs. These techniques automatically generate inputs for finding bugs in an application, but they do not focus on finding bugs that are related to the protocol state.

Protocol reverse engineering can help specify protocol definitions. It is the process of extracting application-level specifications for network protocols. For example, Prospex [26] and the research by Wondracek et al. [27] automatically infer specifications for stateful network protocols by giving sample protocol messages. We believe that they are complementary to AspFuzz. If AspFuzz generates inputs using a specification that is extracted by these techniques, AspFuzz could perform fuzzing without manually specifying protocol definitions.

3. AspFuzz

3.1 Overview

In this paper, we present AspFuzz, a state-aware protocol fuzzer. AspFuzz targets network-connected software that implements application-layer protocols. The key observation behind AspFuzz is that most attack messages do not conform to the *application-layer protocol specification*. The specification consists of the state, format, and syntax of messages, which are defined in RFCs. For example, according to SecurityFocus BID 11705, an attack message for a buffer overflow vulnerability in the Digital Mappings Systems POP3 server exceeds the POP3 command length limitation of protocol specification. According to RFC 1939, the POP3 argument length is up to 40 characters. In addition, some vulnerabilities are exploited by messages in incorrect orders. For example, Apache HTTP Servers 2.0.37 through 2.0.45 have a denial of service vulnerability [8]. This vulnerability can be exploited if an HTTP request is followed by multiple `Host` headers. Because HTTP/1.1 does not allow multiple `Host` headers, this attack message does not conform to HTTP/1.1.

To automatically generate attack messages that exploit possible vulnerabilities, AspFuzz leverages the application-layer protocol specification. Figure 1 shows an overview of AspFuzz. AspFuzz generates various types of attack messages that deliberately violate the format or state specification of the protocol. By doing so, AspFuzz discovers several types of vulnerabilities, such as buffer overflow, denial of service, and format string vulnerabilities. For example, POP3 defines that a response should be an alphabetic string of up to 512 characters. AspFuzz generates messages that violate this protocol format. AspFuzz also generates messages in an anomalous order. For example, HTTP/1.1 defines that there be only one `Host` header. AspFuzz generates attack messages that have multiple `Host` headers. The vulnerabilities exploited by attack messages conforming to the format or order specification are out of the scope of AspFuzz.

3.2 Application-Layer Protocol Specification

The application-layer protocol specification consists of the state, format, and syntax of messages, which are defined in RFCs. We exemplify the specification of POP3. The state transition of POP3 servers is shown in Fig. 2. A POP3 session starts in the *authorization* state. A POP3 server receives a user name and password in the POP3 *authorization* state, and then it makes a transition to the *transaction* state. In this state, the client can get mail lists or the body of mail. If the server receives the `QUIT` command, it makes a transition to the *update* state and then the connection is terminated.

A POP3 message from the client to the server consists of a command that represents the desired operation and corresponding argument. The main POP3 command, syntax, format, and acceptable state are shown in Table 1. For example, the `USER` command is used to send a user name to the server for authentication; it can be used only in the *authorization* state and takes one argument (a user name). The argument length should be less than or equal to 40 characters, the responses may be up to 512 characters long, and messages should consist of printable ASCII characters. The `RETR` command is used to retrieve a message that is specified in its argument. The argument is a message number that should be a positive integer.

To use AspFuzz, an application-layer protocol specification is described as a protocol definition. A protocol definition is described in the `tsfrule` language that was developed in TCP Stream Filter [28]. By use of the `tsfrule` language, the primary application-layer protocol definitions can be described; the authors described four application-layer protocol definitions for POP3, HTTP, SMTP, and FTP [28]. Protocol definitions are divided into two parts: definitions of messages and definitions of states. Definitions of messages define the syntax, length limitation, character restriction, and sequence of messages. These definitions are described as a regular expression. Definitions of states define messages that are sent or received and the state transition in each protocol state. These definitions are described as a finite state machine.

Part of the POP3 protocol definition is shown in

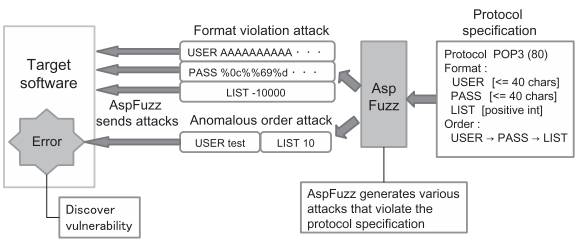

Fig. 1 Fuzzing process in AspFuzz.

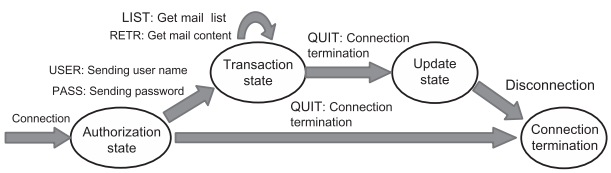

Fig. 2 State transition of POP3 servers.

Table 1 Main POP3 message specification. SP means space, and CRLF means carriage return/line feed as termination of command.

Command syntax and format	*State*
`USER` SP *Name* CRLF *Name*: up to 40 printable ASCII characters	Authorization
`PASS` SP *Pass* CRLF *Pass*: up to 40 printable ASCII characters	Authorization
`QUIT` CRLF	Any
`LIST` CRLF	Transaction
{`LIST` | `RETR`} SP *Message_number* CRLF *Message_number*: positive integer	Transaction

```
1: regexpr SP          = ' '
2: regexpr CRLF        = "\r\n"
3: regexpr User_Cmd    = "USER" SP (['\033'-'\126']+)[<=40] CRLF
4: regexpr Pass_Cmd    = "PASS" SP (['\033'-'\126']+)[<=40] CRLF
5: regexpr List_Cmd    = "LIST" SP CRLF
6: regexpr List_Arg    = "LIST" SP (['0'-'9']+)[<=40] CRLF
7: regexpr Retr_Cmd    = "RETR" SP (['0'-'9']+)[<=40] CRLF
8: regexpr OK_Reply    = "OK" SP (['\033'-'\126']+)[<=507] CRLF
9: regexpr Quit_Cmd    = "QUIT" CRLF
10: initial state authorization {
11:    <: User_Cmd -> auth_reply;
12:    | Quit_Cmd -> terminate;
13: }
14: state auth_reply {
15:    :> OK_Reply -> pass_auth;
16: }
17: state pass_auth {
18:    <: Pass_Cmd -> pass_reply;
19:    | Quit_Cmd -> terminate;
20: }
21: state pass_reply {
22:    :> OK_Reply -> transaction;
23: }
24: state transaction {
25:    <: List_Cmd -> list_reply;
26:    | List_Arg -> list_reply;
27:    | Retr_Cmd -> retr_reply;
28: }
```

Fig. 3 Part of POP3 protocol definition.

Fig. 3. Lines 1 to 9 describe the definitions of messages. `(['\033'-'\126']+)` in line 3 means that the character range of the ASCII code should be between the decimal values of 33 and 126 (i.e., printable characters) and should repeat more than once. `[<=40]` in line 3 means that the length of the string should be less than or equal to 40 characters. In lines 10 to 28, the definitions of states are expressed. Lines 10 to 13 define the *authorization* state, in which the `USER` or `QUIT` commands can be used. A rule for a state transition is represented by "*message* `->` *next state*", which means that if a *message* is received or sent, a transition is made to the *next state*. The symbols `<:` and `:>` denote the direction of the message; `<:` means a message from client to server, and `:>` means a message from server to client. Line 11 defines that if a client sends a *User_Cmd*, a transition is made to the *auth_reply* state. Line 12 defines that if a client sends a *Quit_Cmd*, a transition is made to the *terminate* state.

Some messages must contain dynamically determined information, such as HTTP cookies and SIP `Call-ID`, to continue valid communication. For example, if a cookie is needed in a HTTP communication, HTTP software rejects the message that do not contain the cookie. Since a cookie is dynamically supplied by the server to preserve state information, AspFuzz should dynamically include the cookie when AspFuzz sends generated messages. We can describe this in a protocol definition by extracting and including dynamically determined information in the attack messages. If a protocol definition defines that the cookie is provided in the `Set-Cookie` header in the HTTP reply message and the following request message contains it in the `Cookie` header, AspFuzz analyzes the reply messages to extract the cookie and generates an attack messages that contain it.

Describing a protocol definition is not difficult if a tester has knowledge about the protocol. Since developers of networked software normally have enough knowledge about the protocol specification, it is not difficult for them to describe the protocol definition to test software in development. In fact, we could describe the POP3 and HTTP definitions, which have about 120 and 130 lines of code, respectively, by extracting application-layer protocol specifications based on RFCs. In addition, once someone writes a protocol definition, other people can reuse it to test software that implements the same application-layer protocol. Furthermore, the approach to automatically extracting application-layer specifications [26], [27] can be helpful for AspFuzz to perform fuzzing without manually specifying protocol definitions.

3.3 Crafting Attack Messages

To discover potential vulnerabilities, AspFuzz automatically generates attack messages that exploit possible vulnerabilities by leveraging the application-layer protocol specification. AspFuzz uses two types of attacks, *format violation* and *anomalous order*, and also generates attack messages that combine the two.

3.3.1 Format Violation Attack

A format violation attack deliberately violates the format or syntax specification. We introduce several kinds of fuzzing methods by using real examples of POP3 and HTTP. AspFuzz also crafts attacks that combine two kinds of fuzzing methods.

- **Contains long string**: If a message has a length limitation in the protocol specification, AspFuzz generates an attack message that has a string longer than the defined length limitation. In addition, although RFC does not specify the length limitations, the URI of HTTP messages normally does not have an exceedingly long string (e.g., more than 1000 characters). AspFuzz generates a message that has an exceedingly long string in the URI part of an HTTP message.
- **Violates character restriction**: If acceptable characters are restricted in the protocol specification, AspFuzz generates an attack message that contains prohibited characters. For example, since messages used in POP3 should consist of printable ASCII characters, AspFuzz generates a message that contains non-printable ASCII characters.
- **Contains erroneous value**: If a message field specifies a value for desired operation, AspFuzz generates an attack message that contains an erroneous or unexpected value. For example, since an argument of a POP3 `LIST` command should be a positive integer, AspFuzz generates an attack message that has a negative value or huge value as an argument.
- **Contains format string**: A format string vulnerability is common in the software of text-based protocols such as POP3, HTTP, and SIP. AspFuzz generates an attack

message that contains a format string such as "%s" and "%d".

- **Changes number of fields**: If the number of message fields is limited, AspFuzz generates an attack message by adding or eliminating fields. For example, since the POP3 `RETR` command takes one field as its argument, AspFuzz generates an attack `RETR` message that has two or more fields or no field.
- **Removes delimiter**: Many application-layer protocols use delimiter bytes to divide a message into individual fields. AspFuzz generates an attack message that does not have delimiter bytes. For example, since the POP3 command uses a carriage return/line feed (CRLF) as the termination of a command, AspFuzz generates attack messages without CRLFs.

3.3.2 Anomalous Order Attack

An anomalous order attack deliberately violates the state specification. SecurityFocus BIDs 7723, 13873, and 14174 report vulnerabilities that are exploited by messages that have an incorrect order. As explained in Sect. 3.2, a protocol specification defines a set of message types that can be handled in a particular protocol state and the order in which message are exchanged. To automatically generate an anomalous order of messages, AspFuzz uses a message that cannot be used in the current protocol state. For example, since the POP3 `USER` command can be used only in the *authorization* state, AspFuzz sends the `USER` command in the *transaction* state. In some protocols, including multiples of the same headers in a message is prohibited. HTTP/1.1 defines that there be only one `Host` header or `Content-Length` header. AspFuzz generates messages that have multiple `Host` or `Content-Length` headers. AspFuzz generates anomalous message ordering using the following heuristics.

- **Exchanges messages**: AspFuzz exchanges the order of messages to be sent to a target server. If a protocol definition says messages A and B must be sent in the order A→B (A is followed by B), AspFuzz exchanges the order and sends them like B→A. Many protocols have a request/response pattern; i.e., request A is sent, response B is received, and another request C is sent. In this case, AspFuzz sends request C before request A to exchange the order of messages.
- **Duplicates a message**: AspFuzz duplicates messages to generate anomalous message orders. When AspFuzz identifies a pattern of sending two messages A and B successively, it duplicates messages A or B; AspFuzz tries A→A→B and A→B→B. When AspFuzz identifies a request/response pattern, it duplicates the request message. For example, if a protocol definition specifies message A is sent, message B is received, and another message C is sent, AspFuzz duplicates message A; i.e., AspFuzz sends A, receives B, and sends A again.
- **Eliminates a message**: AspFuzz eliminate some messages. When AspFuzz identifies a pattern of sending two messages A and B successively, it eliminates either message A or B. If a protocol definition specifies the messages A and B must be sent in the order A→B, AspFuzz sends message A or B exclusively. Note the difference from the heuristics of exchanging message orders in which both messages are sent in the probably incorrect order. When AspFuzz identifies a request/response pattern, it eliminates the request and response pair entirely.
- **Sends a message in another state**: As described in Sect. 3.1, many application-layer protocols have states which define a set of valid messages. To check for vulnerabilities, AspFuzz selects a message invalid in the current state but valid in another state.
- **Unexpected termination**: AspFuzz closes a connection at random to simulate unexpected termination of the current session. AspFuzz closes a connection when a message is being transmitted.

The current implementation does not guarantee the incorrectness of the generated messages. For example, AspFuzz exchanges the order of two successive messages without checking a protocol inhibits the exchanged order. AspFuzz tries the exchanged order (B→A) whenever it finds successive messages (A→B). If the protocol allows B→A, the generated messages are not anomalous. AspFuzz uses the heuristics to generate anomalous orders. Since there are many formal methods to describe state-based protocols, we believe model checking can be used to generate anomalous message orders.

In addition, AspFuzz can automatically craft an attack message whose format and order are incorrect. For example, AspFuzz crafts an attack message that uses a `USER` command whose arugment is over 40 characters and then sends it in the *transaction* state instead of the *authorization* state.

4. Implementation

The architecture of AspFuzz is shown in Fig. 4. We implemented AspFuzz in Java. AspFuzz consists of two systems: the attack generation system and the attack management system. The attack generation system generates attack messages based on a protocol definition. The attack management system sends attack messages and receives response messages. In addition, it tracks the target software's protocol state by analyzing both sent and received messages to send attacks in a certain protocol state.

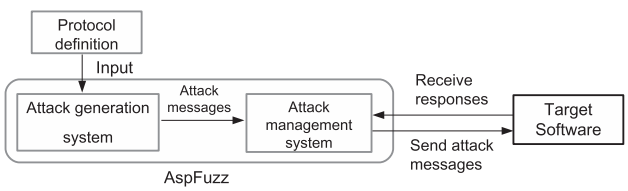

Fig. 4 Architecture of AspFuzz.

4.1 Attack Generation System

The attack generation system analyzes a given protocol definition and generates attack messages based on the protocol definition. We implemented it based on a `tsfrule` compiler, which was developed in TCP Stream Filter.

An example of attacks that are automatically generated using the POP3 `USER` and `LIST` command definitions is shown in Table 2. Suppose AspFuzz uses a protocol definition shown in Fig. 3. The attack generation system analyzes `(['\033'-'\126']+)[<=40]` defined in line 3. `(['\033'-'\126']+)[<=40]` means that the character range of the ASCII code should be between the decimal values of 33 and 126 (i.e., printable character) and the length of the string should be less than or equal to 40 characters. The attack generation system records this character and length limitation. Then the attack generation system generates attack messages that contain non-printable characters (the second row in Table 2) or over 40 characters, such as 41, 255, or 1024 characters, which are possibly the boundary length of buffer overflows in the target software (the first row in Table 2). Lines 10 to 13 in Fig. 3 defines that `USER` command can be used in the `authorization` state, and lines 24 to 28 define that `USER` command can not be used in the *transaction* state. Then the attack management system analyzes these definitions, and records that `USER` command can be used only in the `authorization` state. Then the attack management system generates anomalous order attack messages that contain `USER` command in the *transaction* state instead of the *authorization* state (the eighth row). The last row shows combination of anomalous order and format violation attack message that contains over 40 characters.

To send attack messages in a specific protocol state, AspFuzz generates not only attack messages but also normal messages that transitions to a specified protocol state by leveraging the target protocol's state machine. State machine is extracted from the protocol definition. For example, the third row in Table 2 shows an example of attack message that contains `LIST` command with erroneous value. Since `LIST` command should be sent in the *transaction* state, the attack generation system generates valid `USER` and `PASS` command to make a transition to the *transaction* state of the target software.

Table 2 Example of attack messages using POP3 `USER` and `LIST` command definitions. 'A' x *N* means that character 'A' repeats *N* times.

Attack message	*Attack type*
`USER` SP 'A' x *N* CRLF *N*= {41, 255, 1024 ···}	Long string
`USER` SP *binary* CRLF *binary*: non-printable ASCII characters	Violated character restriction
`LIST` SP *value* CRLF *value*= {-10000, -0, 1111111111 ···}	Erroneous value
`USER` SP *string* x 20 CRLF *string*= {"%s","%d", "%s%s%s" ···}	Format string
`USER` SP 'A' x 20 SP 'A' x 20 CRLF	Duplicated fields
`USER` SP CRLF	No fields
`USER` SP 'A' x *N* (without CRLF) *N*= {41, 255, 1024 ···}	Long string and no delimiters
`USER` SP "test" CRLF sent in transaction state	Anomalous order
`USER` SP 'A' x 255 CRLF sent in transaction state	Long string in anomalous order

To generate attack messages flexibly, AspFuzz uses attack generation rules. Attack generation rules are the configurations for generating attack messages, and a tester can easily add site-specific rules. The attack generation rules include many variables to generate attacks, such as the number of characters in a long string or strings that may exploit a format string vulnerability. The default rules are defined based on our heuristics. The rules also include information for successfully sending attack messages, such as a user name and password for authenticating POP3 servers. If a new type of attack is discovered or a tester wants to add site-specific rules, he/she can write his/her own rules. For example, if a tester wants to contain specific format string, the tester describes a specific format string in the attack generation rules and the attack generation system generates attack messages that contain it. Suppose a tester wants to test HTTP requests that access a CGI program in the target software. If the tester describes a specific CGI program name in the attack generation rule, AspFuzz generates HTTP requests that access it.

If necessary, AspFuzz contains dynamically determined information such as HTTP cookie and SIP `Call-ID` in the generated message.

4.2 Attack Management System

The attack management system sends attack messages to the target software and receives response messages. The attack management system automatically tracks the target software's protocol state by using the state tracking system, which was developed in TCP Stream Filter. The state tracking system monitors both sent and received messages and tracks the state based on the protocol definition. If an attack message is sent in a certain protocol state, AspFuzz sends some messages to make a transition to the protocol state before sending the attack message. For example, since the attack message shown in the third row in Table 2 should be sent in the *transaction* state, the attack management system sends valid `USER` and `PASS` command and checks the protocol state. If the target software transits to the *transaction* state, the attack management system sends attack message to the target software.

In the current implementation, we manually decide whether the attack succeeds or not. To help manual analysis, the attack management system checks responses against attack messages and the network socket condition. If target software do not respond, target software disconnects the attack management system, or an error occurs in the network socket, AspFuzz may exploit a vulnerability in the target software. In that case, the attack management system automatically issues the warning message. We decide whether the attack succeeds or not by leveraging warning messages, behavior of the target software, and the target software's

Table 3 Discovered POP3 vulnerabilities.

BID	*Software*	*First successful attack*	*Attack type*
4295	QPopper 4.03	USER SP 'A' x 2048 (without CRLF)	Long string and removes delimiter
7724	BaSoMail 1.24	LIST SP -10000 CRLF (sends 100 times successively)	Erroneous value
11256	YahooPOPS! 0.6	USER SP 'A' x 2000 CRLF	Long string
11705	Digital Mapping Systems POP3 Server	USER SP 'A' x 1024 CRLF	Long string
–	BaSoMail 1.24	USER SP 'A' x 2048 CRLF	Long string

log, and so on. In the future, we plan to detect successful attacks automatically by using buffer overflow runtime detection systems such as StackGuard [29], LibSafe [30], TaintCheck [31], and MemSherlock [32], intercepting signals such as SIGSEGV (which indicates an illegal memory reference), or monitoring resource usage of the target software.

5. Experiments

5.1 Experimental Setup

We conducted experiments to show that AspFuzz can discover vulnerabilities. First, we searched for vulnerabilities of POP3, HTTP, and SIP implementations published in SecurityFocus [11]. We analyzed the details of the collected vulnerabilities to determine whether our approach could in principle find the vulnerabilities. Then we collected the real POP3 and HTTP vulnerable software and sent attack messages generated by AspFuzz to determine whether AspFuzz could actually find the vulnerabilities. We also used the Common Vulnerabilities and Exposures database [33], and several other hacker and security sites to check the vulnerability information and find exploit codes for the vulnerabilities.

We described POP3 and HTTP protocol definitions and specific attack generation rules, such as authentication information for POP3 servers and specific file name to some software for the experiment. We used a testbed machine that ran AspFuzz on Fedora 8 with an Intel Pentium 4 3.40 GHz and 1 GByte of main memory. We installed each target software in WindowsXP or Fedora Core 6 on a VMware virtual machine that was allocated 256 MBytes of memory in the testbed machine. AspFuzz sent generated messages to each software in the VMware virtual machine and received response messages from them.

Most vulnerabilities described in this section are caused by format violation. In our investigation, there are three vulnerabilities triggered by anomalous ordering of messages. This fact implies that AspFuzz can discover vulnerabilities that may be overlooked by traditional state-unaware fuzzers even though there are not so many vulnerabilities caused by anomalous message ordering.

5.2 POP3 Results

We searched for all the POP3 server vulnerabilities reported from January 2002 to March 2009 in SecurityFocus. Reports of 61 POP3 vulnerabilities were made, and 40 reports had details of the vulnerabilities. Since the rest of reports did not have enough information about vulnerabilities, we could not decide whether AspFuzz can discover these vulnerabilities or not. A closer examination of the reports showed that AspFuzz can discover 20 vulnerabilities. They include 14 buffer overflow vulnerabilities, 5 denial of service vulnerabilities, and 1 format string vulnerability. The remaining 20 vulnerabilities are not target vulnerabilities of AspFuzz because these vulnerabilities are not exploited by the attack messages that do not conform to the application-layer protocol.

We attempted to collect POP3 servers that contained the 20 discoverable vulnerabilities and finally obtained 4 of them. As a result of sending the messages generated by AspFuzz, we successfully exploited all four known vulnerabilities. The discovered vulnerabilities and successful attacks are shown in Table 3. The table indicates the report identifiers assigned by SecurityFocus (BID), the name of target software, the attack message that AspFuzz exploited vulnerability for the first time (first successful attack), and generated attack message type. We explain the details of the Qpopper vulnerability as a successful attack example. According to SecurityFocus BID 4295, Qpopper 4.03 has a remote denial of service vulnerability. If a string longer than 2048 characters is sent to the qpopper process, a denial of service condition occurs. The first successful attack generated by AspFuzz contains 2048 characters as a USER command argument and does not contain CRLF which represents the command termination. Because this attack did not contain CRLF, the server did not reply. After sending the attack, we found that the Qpopper process had gone into an infinite loop by monitoring the CPU usage and Qpopper's log. As a result, the Qpopper process could not provide POP3 services.

In addition to the reported vulnerabilities, we discovered one previously unknown vulnerability. This vulnerability was found when we conducted an experiment with BaSoMail 1.24. The successful attack message has 2055 characters that comprise a USER command, space, a 2048-character user name, and CRLF. When AspFuzz sent it to BaSoMail 1.24, a denial of service condition occurred, such as the software crashing, multiple access violation windows being shown, or the reply message not being sent. After manual testing, we found that if a string longer than 2049 characters is sent to BaSoMail, a denial service condition occurs. This message is different from the successful attack for SecurityFocus BID 7724 of the same software. Note that this vulnerability differs from the one reported in CVE-2004-2168, which causes denial-of-service when an attacker

Table 4 Discoverable POP3 vulnerabilities.

BID	*Software*	*Example of attack*	*Attack type*
4055	Delegate 7.8.0	USER SP 'A' x 242 CRLF	Long string
4427	FTGatePro 1.0 5	APOP SP "user" SP 'A' x 1000 CRLF	Long string
5285	SmartMax MailMax 4.8	USER SP 'A' x 247 CRLF	Long string
5327	T.Hauck Jana WebServer 1.46	DELE SP Value CRLF Value: large message number	Erroneous value
6053	Alt-N Mdaemon 6.07	DELE SP -1 CRLF	Erroneous value
6074	SmartMail Server 2.0	Sends "AAAA" and closes the connection	Sends anomalous message
6183	IISPop 1.181	'A' x 289999	Long string
6197	MailEnable 1.5018	USER SP 'A' x 2009 CRLF	Long string
7519	SLMail 5.1.0.4420	PASS SP 'A' x 4654 CRLF	Long string
8473	vhost-3.05r3	USER SP "%s%s" x 5000 CRLF	Long format string
9794	1st Class Mail Server 4.0	APOP SP "user" SP 'A' x 626 CRLF	Long string
10728	Gattaca Server 2003 1.1.10.0	LIST SP 9999999999999999 CRLF	Erroneous value
12711	Foxmail Email Server 2.0	USER SP 'A' x 5000 CRLF	Long string
15972	Floosietek FTGate 4.4	USER SP "%n" x 20 CRLF	Format string
19651	Alt-N MDaemon 8.1.3	USER SP 'A' x 3370 CRLF	Long string
25496	Hexamail Server 3.0.001	USER SP 'A' x 1024 CRLF	Long string

Table 5 Discovered HTTP vulnerabilities.

BID	*Software*	*First successful attack*	*Attack type*
13873	Squid 2.5 stable 4	Content-Length: SP 0 CRLF (sends multiple times)	Erroneous value and sends anomalous order
22791	Apache Software Foundation mod_jk 1.2.19	POST SP "/A" x 3000 SP HTTP/1.1 CRLF	Long string
23341	Wserve 4.6	HEAD SP 'A' x 2000 SP HTTP/1.1 CRLF	Long string
23445	KarjaSoft Sami HTTP Server 2.01	HEAD SP "%s%s%s" x 20 SP HTTP/1.1 CRLF	Format string
23545	3proxy 0.53g	Host: SP 'A' x 20000 CRLF	Long string
23713	Pi3Web 2.0.3	HEAD SP "/A" x 1024 SP HTTP/1.1 CRLF	Long string
24649	Apache 2.24	Cache-Control: SP max-age= CRLF	Removes field
28757	Novell eDirectory 8.8	Connection: SP *String* CRLF *String*: 100 random characters that include ','	Long string and violates character restriction
29661	BitTorrent 6.0.1	Range: SP bytes'1' x 20000-'1' x 20000 CRLF	Erroneous value and removes field
30273	BEA Systems WebLogic Server 8.1 SP6	HEAD SP /weblogic/index.jsp SP HTTP'A' x 2048/'A' x 2048 CRLF	Long string
30869	Fedora Directory Server 1.1.1-3	Accept-Language: SP 'A' x 255 CRLF	Long string
31416	CCProxy Server 6.6	CONNECT: SP 'A' x 2000 SP HTTP/1.1 CRLF	Long string
33898	Steamcast 0.9.75	Content-Length: SP -100000 CRLF	Erroneous value
34134	HP OpenView Network Node Manager 7.53	Cookie: SP aaaaa='A' x 10000 CRLF	Long string
34135	HP OpenView Network Node Manager 7.53	Accept-Language: SP 'A' x 500 CRLF	Long string
34294	HP OpenView Network Node Manager 7.53	Cookie: SP aaaaa='A' x 10000 CRLF	Long string

creates multiple connections to ports 25 and 110.

Other vulnerabilities that are discoverable but cannot be tested by AspFuzz are shown in Table 4. In Table 4, there is one notable vulnerability that is triggered by anomalous message ordering. According to SecurityFocus BID6074, SmartMail Server 2.0 is prone to a denial of service if a client sending data does not follow the normal protocol and closes the connection unexpectedly to quit the session. In this table, examples of attack are decided by vulnerability reports and the results of manual analysis of public exploit code. AspFuzz can generate these attack messages. However, they were unable to be tested, because the software was not available or could not be installed in our testbed environment.

5.3 HTTP Results

We searched for all the vulnerabilities of HTTP implementations reported from January 2007 to March 2009 and several vulnerabilities reported before 2007 in SecurityFocus. Reports of 71 HTTP vulnerabilities were made, and 54 reports had details of the vulnerabilities. Since the rest of reports did not have enough information about vulnerabilities, we could not decide whether AspFuzz can discover these vulnerabilities or not. A closer examination of the reports showed that AspFuzz can discover 25 vulnerabilities. They include 11 buffer overflow vulnerabilities, 10 denial of service vulnerabilities, 2 memory corruption vulnerabilities, 1 HTTP request smuggling vulnerability, and 1 arbitrary code execution vulnerability. The remaining 29 HTTP vulnerabilities are not target vulnerabilities of AspFuzz because they are not exploited by attack messages that do not conform to the application-layer protocol.

We attempted to collect HTTP software of the 25 discoverable vulnerabilities and obtained 19. AspFuzz successfully exploits 16 of the known vulnerabilities. Table 5 shows the discovered vulnerabilities. First successful attack shows part of attack message that is believed to exploit vulnera-

Table 6 Discoverable HTTP vulnerabilities.

BID	*Software*	*Example of attack*	*Attack type*
27701	IEA Software RadiusX 5.1.38	`Content-Length`: SP 2147483647 CRLF	Erroneous value
28572	Novell eDirectory 8.82	`HEAD` SP '<' x 2048 SP HTTP/1.1 CRLF	Long string
28610	SmarterTools SmarterMail 5.0	`TRACE` SP 'A' x 8784 CRLF	Long string
28795	BigAnt IM Server 2.23	`GET` SP 'A' x 2048 CRLF	Long string
30652	OmniSwitch OS9000 Series	`Cookie`: SP `Session`= 'A' x 2932 CRLF	Long string
32560	Rumpus FTP Server 6.0	'A' x 2908	Long string

bility. We explain here the details of the Steamcast vulnerability as an example of a successful attack. According to SecurityFocus BID 33898, Steamcast is prone to multiple memory-corruption vulnerabilities. If a `Content-Length` header that contains a negative value is sent to Steamcast, a denial of service condition occurs or an attacker can execute an arbitrary code in the context of the application. A successful attack message generated by AspFuzz consists of a `POST` request, `Host` header, `Content-Length` header that has "-100,000", and a message body. Steamcast crashed after this attack message was sent to it.

We also discovered a HTTP/1.1 RFC violation bug in a HTTP proxy that leads to the HTTP Request Smuggling vulnerability [34]. HTTP Request Smuggling is a hacking technique against HTTP devices/entities (e.g., cache server and proxy server). HTTP/1.1 RFC does not allow multiple `Content-Length` headers, and a Watchfire technical report [34] discloses that violation of this rule leads to the HTTP Request Smuggling vulnerability. The technical report notes that Squid 2.5.stable4 is observed to process a HTTP request with anomalous order as a valid HTTP request. This implies that Squid 2.5.stable has an HTTP request smuggling vulnerability that is triggered by anomalous message ordering. In our experiment, AspFuzz sent generated attack messages that contained multiple `Content-Length` headers whose value were "0" to Squid 2.5.stable4. Squid responded with an "HTTP/1.1 200 OK" response message. Since the attack messages did not conform to HTTP/1.1 RFC, Squid should have sent error response codes. This behavior shows that Squid 2.5.stable4 processes an HTTP request with anomalous order that violates HTTP/1.1 RFC as a valid HTTP request.

AspFuzz could not exploit three vulnerabilities: Apache memory corruption vulnerability (BID 7723), xserver HTTP request buffer overflow vulnerability (BID 25030), and SW-HTTPD HTTP request denial of service vulnerability (BID 34188). Even though we sent the known exploit code for these vulnerabilities, the vulnerabilities could not be exploited. We believe this was because the obtained software had already been fixed or some conditions for exposing the vulnerability were not satisfied.

Other vulnerabilities that are discoverable but cannot be tested by AspFuzz are shown in Table 6. The column of "Example of attack" shows part of the attack messages that are believed to exploit vulnerability. AspFuzz can generate these attack messages. However, they were unable to be tested, because the software was not available or could not be installed in our testbed environment.

5.4 SIP Results

We searched for SIP vulnerabilities in SecurityFocus. According to SecurityFocus BID 14174, the multiple vendor VoIP phones handle spoofed SIP status message in an improper manner, and this vulnerability can be exposed only by using an anomalous order attack. According to RFC 3265, the SIP `NOTIFY` message is used to notify an event that has been requested by an earlier `SUBSCRIBE` message. If the notified event is not previously subscribed to by the `SUBSCRIBE` message, the client has to respond with a "481 Subscription does not exist" error message. The vulnerable phones process `NOTIFY` messages on events that are *not* previously subscribed instead of rejecting them. This is because they do not correctly verify the `Call-ID`, `tag`, and `branch` headers of `NOTIFY` messages. AspFuzz can automatically generate an anomalous order attack message that exploits this vulnerability.

6. Conclusion

We presented AspFuzz, a state-aware protocol fuzzer based on the specifications of application-layer protocols. AspFuzz targets network-connected software that implements application-layer protocols. The key observation behind AspFuzz is that most of attack messages exploit networked software vulnerabilities, which violate the strict definitions of application-layer protocols. AspFuzz automatically generates a large number of attack messages in incorrect format or syntax as well as in anomalous orders by leveraging application-layer protocols of the target software. We investigated the reported vulnerabilities for POP3, HTTP, and SIP software and sent generated messages to them. With AspFuzz, we discovered 20 reported and 1 previously unknown vulnerabilities for POP3 and 25 reported vulnerabilities for HTTP. Two vulnerabilities among these can be discovered by the state-awareness of AspFuzz. It can also find a SIP state-related known vulnerability.

References

[1] Symantec Corporation, "Internet Security Threat Report XIV," http://www.symantec.com/business/theme.jsp?themeid=threatreport, 2009.

[2] "Month of Browser Bugs," http://browserfun.blogspot.com/, July 2006.

[3] N. Neves, J. Antunes, M. Correia, P. Veríssimo, and R. Neves, "Using attack injection to discover new vulnerabilities," Proc. 36th

IEEE International Conference on Dependable Systems and Networks (DSN '06), pp.457–466, 2006.
[4] "PROTOS-Security Testing of Protocol Implementations," http://www.ee.oulu.fi/research/ouspg/protos/
[5] "Peach Fuzzing Platform," http://peachfuzzer.com/, 2008.
[6] D. Aitel, The advantages of block-based protocol analysis for security testing, Immunity Inc., 2002.
[7] M. Vuagnoux, "Autodafé: An act of software torture," tech. rep., Swiss Federal Institute of Technology (EPFL), Cryptography and Security Laboratory (LASEC), 2006.
[8] "Apache APR_PSPrintf Memory Corruption Vulnerability." http://www.securityfocus.com/bid/7723
[9] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: Toward a stateful network protocol fuzzer," Proc. 9th Information Security Conference (ISC '06), pp.343–358, 2006.
[10] H.J. Abdelnur, R. State, and O. Festor, "KiF: A stateful SIP Fuzzer," Proc. 1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm '07), pp.47–56, 2007.
[11] "SecurityFocus," http://www.securityfocus.com/
[12] S. Sparks, S. Embleton, R. Cunningham, and C. Zou, "Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting," Proc. 23rd Annual Computer Security Applications Conference (ACSAC '07), pp.477–486, 2007.
[13] O. Udrea, C. Lumezanu, and J.S. Foster, "Rule-based static analysis of network protocol implementations," Proc. 15th USENIX Security Symposium, pp.193–208, 2006.
[14] Tenable Network Security Inc., "Nessus vulnerability scanner," http://www.nessus.org/nessus/
[15] McAfee Inc, "McAfee Vulnerability Manager," http://www.mcafee.com/
[16] D. Farmer and E.H. Spafford, "The COPS security checker system," Proc. USENIX Summer Conference, pp.165–170, 1990.
[17] IBM Inc, "Internet Scanner," http://www.iss.net/
[18] Qualys Inc, "QualysGuard," http://www.qualys.com/
[19] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler, "EXE: Automatically generating inputs of death," Proc. 13th ACM Conference on Computer and Communications Security (CCS '06), pp.322–335, 2006.
[20] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08), pp.209–224, 2008.
[21] P. Godefroid, M.Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," Proc. 15th Annual Network and Distributed System Security Symposium (NDSS '08), 2008.
[22] P. Godefroid, A. Kieżun, and M.Y. Levin, "Grammar-based whitebox fuzzing," Proc. 2008 ACM Conference on Programming Language Design and Implementation (PLDI '08), pp.206–215, 2008.
[23] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," Proc. 2005 ACM Conference on Programming Language Design and Implementation (PLDI '05), pp.213–223, 2005.
[24] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," Proc. 5th Joint Meeting of the European Software Engineering Conference and the ACM Symposium on Foundations of Software Engineering (ESEC/FSE '05), pp.263–272, 2005.
[25] D. Molnar, X.C. Li, and D.A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," Proc. 18th USENIX Security Symposium, pp.67–81, 2009.
[26] P.M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," Proc. 30th IEEE Symposium on Security and Privacy (S&P '09), pp.110–125, 2009.
[27] G. Wondracek, P.M. Comparetti, C. Kruegel, and E. Kirda, "Automatic network protocol analysis," Proc. 15th Annual Network and Distributed System Security Symposium (NDSS '08), 2008.
[28] K. Kono, T. Shinagawa, and M.R. Kabir, "Improving internet server security by filtering on TCP streams," IPSJ Trans. Advanced Computing Systems, vol.46, no.SIG4 (ACS 9), pp.33–44, 2005.
[29] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," Proc. 7th USENIX Security Symposium, pp.63–78, 1998.
[30] A. Baratloo, T. Tsai, and N. Singh, "Libsafe: Protecting critical elements of stacks," Tech. Rep., Bell Labs, Lucent Technologies, 1999.
[31] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," Proc. 12th Annual Network and Distributed System Security Symposium (NDSS '05), pp.123–130, 2005.
[32] E.C. Sezer, P. Ning, C. Kil, and J. Xu, "Memsherlock: An automated debugger for unknown memory corruption vulnerabilities," Proc. 14th ACM Conference on Computer and Communications Security (CCS '07), pp.562–572, 2007.
[33] "Common Vulnerabilities and Exposures," http://cve.mitre.org/
[34] C. Linhart, A. Klein, R. Heled, and S. Orrin, HTTP REQUEST SMUGGLING, Watchfire Inc., 2005.

Takahisa Kitagawa received the B.E. degree from Keio University in 2008. He is currently a master student in the Graduate School of Science and Technology, Keio University. His current research interests include system software and network security.

Miyuki Hanaoka received the B.E. degree from the University of Electro-Communications in 2005, and M.E. from Keio University in 2007. She is currently a Ph.D. candidate in the Graduate School of Science and Technology, Keio University. Her research interests include network security and system software. She is a student member of IEEE, ACM, and IPSJ.

Kenji Kono received the B.Sc. degree in 1993, and M.Sc. degree in 1995, and Ph.D. degree in 2000, all in computer science from the University of Tokyo. He is an associate professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software and Internet security. He is a member of IEEE/CS, ACM, and USENIX.