Utilizing Binary Rewriting for Improving End-host Security

Yougang Song **Brett D. Fleisch**
Distributed Systems Laboratory
Department of Computer Science & Engineering
University of California, Riverside
{ysong, brett}@cs.ucr.edu

Abstract

Conventional methods supporting Java binary security mainly rely on the security of the hosts Java Virtual Machine (JVM). However, malicious Java binaries keep exploiting the vulnerabilities of JVMs, escaping their sandbox restrictions and allowing attacks on end user systems. Administrators must confront the difficulties and dilemmas brought on by security upgrades. On the other hand, binary rewriting techniques have been advanced to allow users to enforce security policies directly on mobile code. They have the advantage of supporting a richer set of security policies and self-constrained written code. However, the high administrative and performance overhead caused by security configuration and code rewriting have prevented rewriters from becoming a practical security tool.

In this paper, we address these problems by integrating binary code rewriters with web caching proxies and build the security system called PB-JARS, a Proxy-based JAva Rewriting System. PB-JARS works as a complimentary system to existing JVM security mechanisms by placing another line of defense between users and their end user systems. It gives system administrators centralized security control and management for mobile code and security policies. We evaluated PB-JARS using a real Java binary traffic model derived from analyzing real web trace records. Our results show that adding binary rewriting to web caching system can be very efficient in improving end host security at low cost.

Keywords: Security and Protection, Binary Rewriting, Web Servers, Traffic Analysis, Performance Evaluation

1. Introduction

Java binaries have two forms: Java applets and application programs [27]. Java applets are typically downloaded from the web and can only be run from within a web browser, while Java applications are stand-alone programs. Both kinds of Java binaries can only be executed within the JVM supported by most current computers and embedded systems. Java binary security is ensured by JVM security mechanisms, which include type-safe language, sandbox models, bytecode verifiers and security managers [7]. However, no system is one hundred percent secure. Flaws and bugs have been frequently found in JVM implementations. For example, in 2002 the Microsoft Virtual Machine (Microsoft VM) was found to allow applets write access to the Java security manager [25]. In 2005, a Sun Java Plugin was found to create temporary files with predictable names [26]. As JVM becomes more complex and new versions of JVM continue to be released, implementation errors and security vulnerabilities are inevitable as history has shown.

In case of security threat events or security upgrades, every client system within a same enterprise environment has to be updated separately because of the JVM’s current isolated security management. However, the discrepancy among each client system and shortage of enough security knowledge often makes a uniform and timely security upgrade difficult. In addition, JVM enforces a sandbox security model, where resource access requests are either granted or prohibited. Therefore, administrators and users often have to disable useful functionalities to prevent a rarely happening but critical security threat. For example, a Java binary’s network access might be disallowed to prevent it from disclosing the content of an important local file.

On the other hand, recent developments in binary rewriting techniques allows arbitrary checking code to be inserted into the binary code to monitor its execution and terminate itself when it is about to violate specified security policies [5]. Its major advantages include: 1) It allows administrators to define more flexible security policies. For the previous example, the policy “no sending of messages after reading specific files” can be enforced and meanwhile preserve the binary’s normal network operation. 2) The code after being rewritten is self-constrained by its internal checking code and does not rely on the host security monitoring system [6]. However, binary rewriting systems were designed to work with standalone computers, the administrative overhead caused by configuring and managing the complicated security policies and the performance overhead [10,11] caused by code rewriting prevent this technique from being used in large scale.

In this paper, we address these problems by integrating binary rewriting into web caching proxies and build the security system called PB-JARS. PB-JARS does not replace JVM security systems but is used as a complimentary system to relieve administrators from security upgrade difficulties to enforce security policies which can’t be supported by the current JVM security model. All Java binaries passing through the proxy server are automatically rewritten with specified security policies and guaranteed to be securer before they reach the recipients. This paper later shows a comprehensive performance evaluation, under the realistic Java binary traffic model, where the binary rewriting overhead is significantly reduced by web caching and PB-JARS adds no significant performance overhead to the proxy server.

The rest of paper is organized as follows: Section 2 gives a survey on binary rewriting techniques to security. Section 3 illustrates, through an example about how PoET rewrites Java binary code to conform to given specifications. Section 4 discusses the issues when applying binary rewriting as a security tool. Section 5 describes PB-JARS, its architecture and implementation details. Section 6 first presents a study of Java binary Internet traffic by analyzing raw Internet access log files and then gives the performance evaluation under the derived traffic model. Section 7 discusses related work, followed by conclusions in Section 8.

2. Binary Rewriting Techniques to Security

A binary rewriting system transforms a binary program into a different but functionally equivalent program [3]. Because it requires no knowledge of the source code, binary rewriting has been widely used in the area of code migrations across different processor architectures, performance instrumentation and program optimization, such as optimizations on code speed, size and power consumption [17,18,19,20,21,22,23]. Applications of binary rewriting techniques in the computer security area mainly focused on dealing with commonly seen and dangerous attacks, such as the buffer overflow problem [16]. Before real binary rewriting security techniques have appeared, there has been substantive work proposed based on compile time analysis and transformation such as RAD [15], which protects buffer overflow attacks by adding protection code into the prologue and epilogue of the program at the compile time. Another technique is based on run-time interception and checking such as Libsafe [12], which is based on a dynamically loadable library that intercepts all function calls made to library functions that are known to be vulnerable. Sandboxing techniques such as Software Fault Isolation (SFI) [13], which use the idea of Address Sandboxing to enforce system security has also been explored.

The Binary-Rewriting RAD [2] extends the work of RAD and first applies the security directly on binary code without requiring access to program source code, symbol tables or relocation information. It uses a combined disassembly technique to identify the boundary of every procedure in the input program. The protection code is appended to the end of the original binary. It inserts the code at the function prolog to save a copy of the return address and the code at the function epilog to check the return address on the stack with the saved copy. Some instructions at the function prolog and epilog are replaced by a JMP instruction to redirect the control to the inserted code at a function's prolog and epilog. Purify [4], detects run-time memory leaks and access errors by inserting checking instructions directly into the object code and before every load or store. To detect memory access errors, Purify maintains a state code for each byte of memory and a run-time check is enforced by the checking instructions whenever the program makes memory access. To make binary analysis and rewriting efficient, SELF [14] proposed a transparent security enhancement to ELF binaries by adding an extra section. The extra section contains information specifically needed for binary analysis and hence it is convenient to perform many security-related operations on the binary code.

Comprehensive binary rewriting security systems appeared when binary rewriting techniques were used to enforce security policies specified by host system requirements. The benefit provided by such security systems is the flexibility to enforce security policies that are not supported by the standard security mechanisms. Notable such systems are Naccio [8], SASI [9] and PoET [6]. Naccio[8] allows the expression of safety policies in a platform-independent high level language and applies these policies by transforming program code. A policy generator takes resource descriptions, safety policies, platform interface and the application to be transformed and then generates a policy description file. This file is used by an application transformer to make the necessary changes to the application. The application transformer replaces system calls in the application to functions in a policy-enforcing library. Naccio has been implemented for both Win32 and Java platforms. Security Automata SFI Implementation (SASI) [9] uses a security automaton to specify security policies and extends the idea of software fault isolation by merging security policy into the application itself. The security automaton acts as a reference monitor for the code. In relation to a particular system, the events that the reference monitor controls are represented by the alphabet, and the transition relationship encodes the security policy enforced by the reference monitor (so called IRM, Inline Reference Monitor [5]). The security automaton is merged into application code by an IRM binary rewriter. It adds code that implements the automaton directly before each instruction. The rewriter is language specific: x86 SASI is the implementation for x86 machine code and JVML SASI is for Java virtual machine.

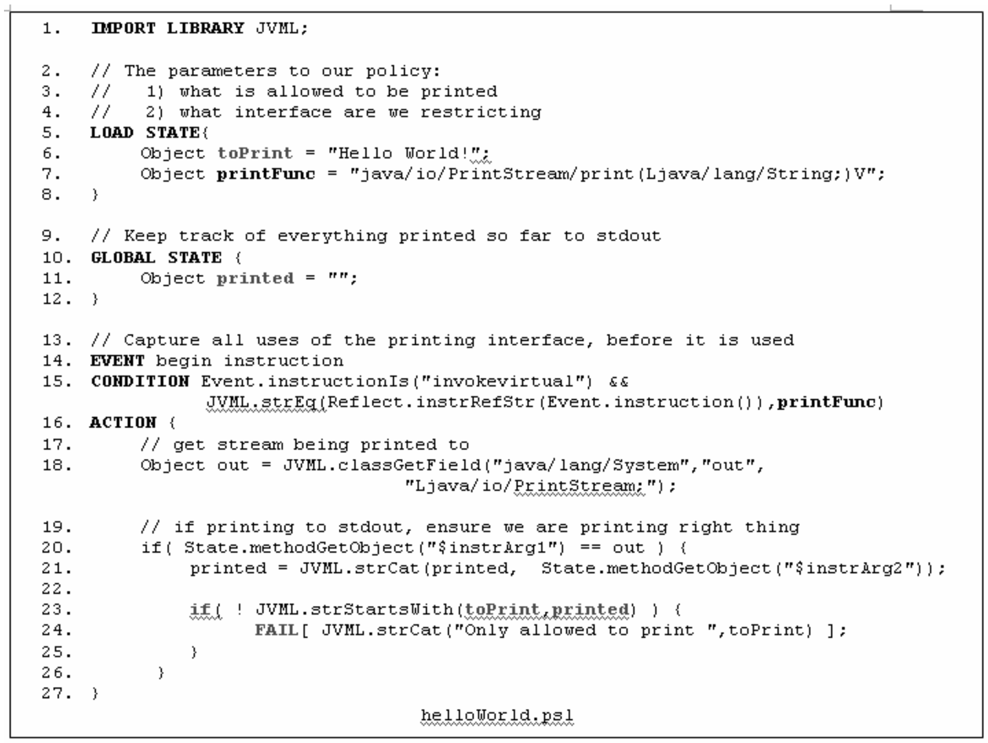
PoET (Policy Enforcement Toolkit) [6] is an implementation of IRM for Java applications. Superior to Naccio and SASI, which requires the source code information or binaries generated with special compiler, PoET does not require any source code information available and the code after being rewritten can be executed on any version of JVM. It uses a relatively higher level, event-oriented, Java-like language to specify security policies, which is called PSLang. Specifying a security policy involve defining [5]: *Security events* to be mediated by the reference monitor; *Security state* that is stored about earlier security events and used to determine which security events can be allowed to proceed; *Security updates* that update the security state, signal security violations and/or take other remedial action when the related security event happens. PoET adds the security enforcement according to the instructions specified in the security policy.

3. An Example: How PoET Rewrites Java Binary Code

We selected PoET in our work because of its availability and intrinsic simplicity. In this section we illustrate through a concrete example how PoET rewrites Java binary code to conform to the given policies.

Figure 1 shows the example policy *helloWorld* taken from PoET source code package. It only allows Java binaries to print strings starting with ‘Hello World!’”. In this policy, the *LOAD*

STATE section (line 5~8) defines the security state *toPrint*, which specifies the starting string the binary call be allowed to print. The *GLOBAL STATE* section (line 10~12) traces the string that has already been printed. The *EVENT ACTION* section (line 14~27) specifies that whenever the print function is invoked and the string to be printed does not start with the security state *toPrint*, the application will be terminated.



```
1.   IMPORT LIBRARY JVML;

2.   // The parameters to our policy:
3.   //   1) what is allowed to be printed
4.   //   2) what interface are we restricting
5.   LOAD STATE{
6.        Object toPrint = "Hello World!";
7.        Object printFunc = "java/io/PrintStream/print(Ljava/lang/String;)V";
8.   }

9.   // Keep track of everything printed so far to stdout
10.  GLOBAL STATE {
11.       Object printed = "";
12.  }

13.  // Capture all uses of the printing interface, before it is used
14.  EVENT begin instruction
15.  CONDITION Event.instructionIs("invokevirtual") &&
              JVML.strEq(Reflect.instrRefStr(Event.instruction()),printFunc)
16.  ACTION {
17.       // get stream being printed to
18.       Object out = JVML.classGetField("java/lang/System","out",
                                   "Ljava/io/PrintStream;");

19.       // if printing to stdout, ensure we are printing right thing
20.       if( State.methodGetObject("$instrArg1") == out ) {
21.            printed = JVML.strCat(printed,  State.methodGetObject("$instrArg2"));
22.
23.            if( ! JVML.strStartsWith(toPrint,printed) ) {
24.                   FAIL[ JVML.strCat("Only allowed to print ",toPrint) ];
25.            }
26.        }
27.  }
                              helloWorld.psl
```

Figure 1. Security policy "helloWorld" (taken from the PoET source code package)

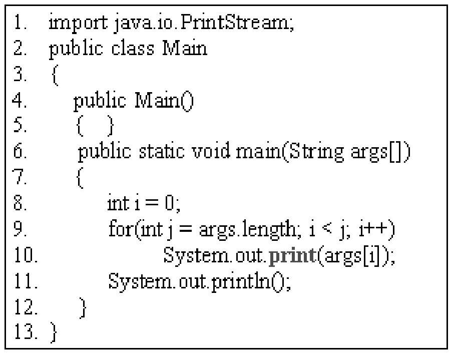
Given a security policy and Java binary code, PoET will perform the following steps to merge the security policy into the Java binary code:

a. preload the specification file, generate the corresponding specification class, and identify the events to be monitored; Events can be categorized into 8 levels: program, classInit, instanceInit, instanceFree, method, exceptionHandler, block, instruction;

b. load and disassemble the Java binary code; Rewrite each disassembled components by inserting constraint code in the order of the above levels.

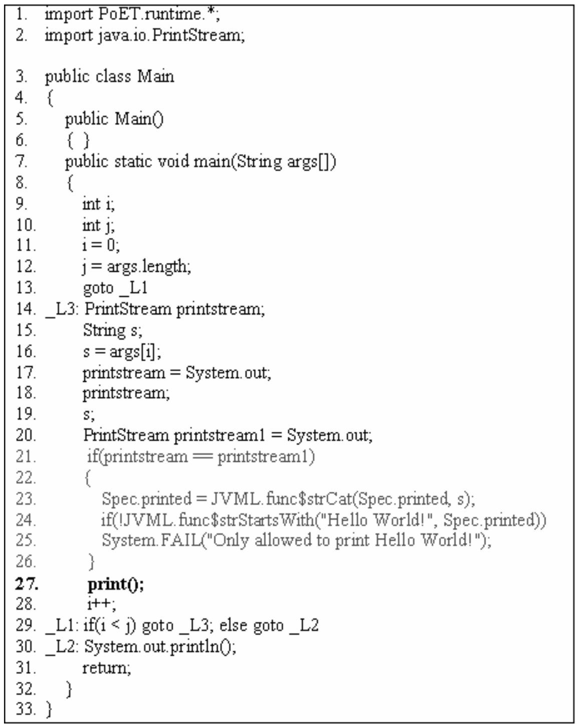
c. Reassemble the disassembled components to form the rewritten code with enforced security policies.

Therefore, given a Java binary code (of which the source code is shown in Figure 2a), Figure 2b shows the code after being rewritten with the security policy *helloWorld*. The code in red is the constraint policy which has been added before the print event.



```
import java.io.PrintStream;
public class Main
{
    public Main()
    {   }
    public static void main(String args[])
    {
        int i = 0;
        for(int j = args.length; i < j; i++)
                System.out.print(args[i]);
        System.out.println();
    }
}
```

a. Source code of which the binary code to be rewritten



```
import PoET.runtime.*;
import java.io.PrintStream;

public class Main
{
    public Main()
    { }
    public static void main(String args[])
    {
        int i;
        int j;
        i = 0;
        j = args.length;
        goto _L1
_L3: PrintStream printstream;
        String s;
        s = args[i];
        printstream = System.out;
        printstream;
        s;
        PrintStream printstream1 = System.out;
        if(printstream == printstream1)
        {
            Spec.printed = JVML.func$strCat(Spec.printed, s);
            if(!JVML.func$strStartsWith("Hello World!", Spec.printed))
            System.FAIL("Only allowed to print Hello World!");
        }
        print();
        i++;
_L1: if(i < j) goto _L3; else goto _L2
_L2: System.out.println();
        return;
    }
}
```

b. The code after being rewritten (generated by JAD Java decompiler)

Figure 2 Code rewriting: before vs after

4. Issues When Applying Binary Rewriting Security System

Binary Rewriting Security Systems have the advantages of supporting a richer set of security policies and self-constrained written code. However, it's hard to apply them directly as one convenient security application system.

The first reason is the high administrative overhead. Typically this includes the complexity of composing security policies and configuring and managing them. PSLang is defined in the high-level language that extends JVML, however in order to identify the security events to be monitored, administrators are required to be very familiar with the JVM Library. For example, Figure 3 shows a part of the security policy “Only allow files in a certain directory to be read”, which is taken from PoET source code package. To monitor the event of *file read*, it requires administrators to determine all the possibilities where the file read can be invoked.

Binary rewriters allow users to define more flexible security policies. Despite this, they can increase the complexity to effectively configure and manage policies. Considering that even the simple “yes-no” security policies under the *access control list* security model [35] can cause a lot of confusion, it’s impractical for administrators to manage those complicated security policies without a convenient security management system. In addition, PoET uses command lines for all the operations, consequently applying security policies is very non user-friendly.

```
……

// When trying to construct a FileInputStream or FileReader make sure it's OK
EVENT begin instruction CONDITION  Event.instructionIs("invokespecial") && JVML.strEq(
Reflect.instrRefName(Event.instruction()), "<init>") && JVML.strStartsWith(Reflect. instrRefStr(
Event.instruction()),"java/io/File") && (JVML.strEndsWith(Reflect. instrRefClassName
(Event.instruction()),"InputStream") ||  JVML.strEndsWith(
Reflect. instrRefClassName( Event.instruction()),"Reader"))
ACTION {
        ……
        // It must be in "that special place"
        if( ! JVML.strStartsWith(realPath, readDir) ) {
            FAIL[ JVML.strCat("Can't read file ",fileName,": not in ",readDir) ];
        }
}
```

Figure 3. Complexity of the security policy

The second concern is the rewriting overhead. Binary rewriting systems were designed to work with standalone computers. The performance overhead not only includes the known rewriting overhead but also the overhead caused by system initialization. In order to run PoET, the JVM must be initialized first and then the security policies must be parsed and loaded into memory. In Figure 4, we evaluated the examples that come with the PoET source code package.

As we can see that in the worst case *LimitMem*, which is used to limit the total memory that an applet can allocate, the rewriting overhead can reach above one second.

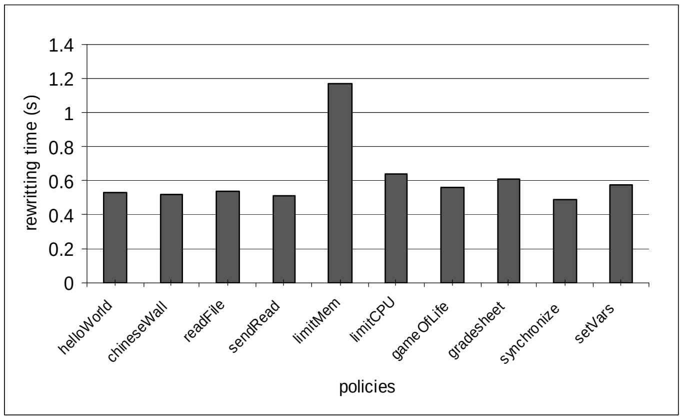


Figure 4. High rewriting overhead

5. PB-JARS: A Proxy-based Java Rewriting System

In this section we describe PB-JARS in detail, focusing on its integration with web proxies.

5.1. Overview and Architecture

A proxy server is positioned between clients and remote Internet servers and transparently mediates all requests for web objects from clients. Proxy servers usually provide caching service, which improves access speed and increases availability by replicating information [30]. PB-JARS cooperates with caching proxy servers to provide Java security for end users (as shown in Figure 5). First, a client sends a Java binary request to the proxy. If the proxy server can not find it in the cache (*cache miss*), the proxy forwards the request to the remote server and PB-JARS intercepts and rewrites the java binary files returned from the remote server. At this point, a small amount of overhead of rewriting is added in the process. The rewritten and hence secured code is then forced to be cached in the proxy cache and distributed to clients. The whole process is transparent to the proxy cache and cache validation is handled by the cache as it used to be.

Subsequent requests for the same Java binary can be satisfied directly by the rewritten code from the cache (*cache hit*) without the intervention of PB-JARS.

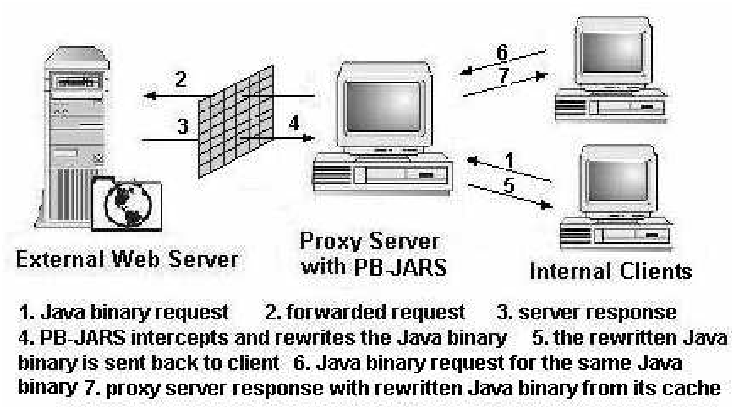

2
3
4
6
7
1
5
External Web Server
Proxy Server with PB-JARS
Internal Clients
1. Java binary request 2. forwarded request 3. server response
4. PB-JARS intercepts and rewrites the Java binary 5. the rewritten Java binary is sent back to client 6. Java binary request for the same Java binary 7. proxy server response with rewritten Java binary from its cache

Figure 5. PB-JARS overview

Therefore, PB-JARS provides the following advantages:

1) Improved security manageability: PB-JARS gives system administrators centralized control of security polices and a uniform way to enforce security policies at the level of administrative domains through proxy servers. There is no specific security configuration required at the recipients' side. In addition, PB-JARS provides convenient tools for administrators to configure and manage the security policies and mobile code.

2) Reduced performance overhead: from our study, we learned that a significant percentage of Java binary requests are repeated. Thus, caching rewritten binary code for repeat requests can efficiently reduce the performance costs associated with binary rewriting.

As shown in Figure 6, PB-JARS has four parts: Database, Application Management, Policy Editor, and Binary Rewriter. *Database* stores security policies. It organizes the security policies into several categories such as file system, network communication and memory usage and differentiates the policies within each category into different security levels. The database also keeps the history of changes made to binaries that are rewritten. *Application Management*, a graphical user interface, is provided for administrators to manage the database and binary rewriter conveniently. The *Policy Editor* reduces the complexity of creating security policies by wrapping the JVM Library (JVML) with an abstract intermediate level. The *Binary Rewriter* is actually a plug-in module for real binary rewriters. It provides two buffers for the binary rewriter. The real binary rewriter gets its input from the receiving buffer and rewrites the code to the output buffer. A *content filter* inserted at the proxy intercepts and filters out the mobile code that pass by to the binary rewriter's receiving buffer. Detailed description about each component can be found in our previous work[10]. In the following section, we will focus on its integration with web proxies.

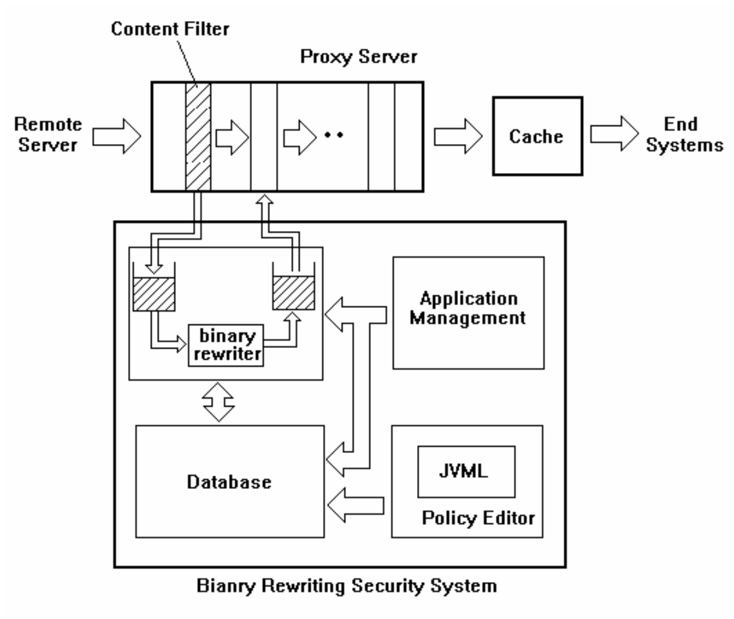


Figure 6. PB-JARS system architecture

5.2. Integration with Web Proxy

We selected Apache [30] as the web proxy server. Although Apache is better-known as a web server rather than a proxy server, the reason that we chose Apache is because it provides a filter mechanism so that we could easily add a content filter that integrates PB-JARS with the proxy server. Also, the performance of Apache as a proxy server is reasonably good. For a prototype system of this kind that integrates a binary rewriter and a web proxy together, the specific web proxy is not as important as testing the research thesis concerning them both being used together.

A skeleton of the content filter is shown in Figure 7. When the proxy gets a request, it first checks if the request can be served from its cache. If it is a cache hit, the orignal *cache_out* filter is added to get the data from the cache, otherwise it adds a *cache_in* filter before all the other output filters to save servers' reponse into the cache. We thus placed our security filter whenever a cache_in filter is needed and made it process the data flow before the cache_in filter. Java binary traffic can be identified by analyzing the URLs. Once it is found, the complete file will be pulled out by the *pull_data_out* function. For Jar files, PB-JARS will verify and unpack it to class files. The function *notify_PoET* will be called then to send the notification to PoET to rewrite the Java binaries. After being rewritten, the Jar files will be repacked and resigned by PB-JARS. The final rewritten Java binary file will be pushed back into the dataflow by the *push_data_in* function.

```
handle_url_request()
  check the url;
  search the proxy cache;
    if cache hit
      add cache_out filter;
    if cache miss
      if java binary requests
        add security_cache_in filter;
      add cache_in filter;

security_cache_in():
   pull_data_out into temporary buffer;
   if jar file
      unpack it;
   notify_PoET;
   wait until Rewriting is done;
   if jar file
      repack and sign it;
   push_data_in;

cache_in():
   cache the rewritten code into cache;
   code is distributed to the client;
```

Figure 7. The skeleton of the content filter added to Apache2.0

6. Performance Evaluation

In this section, we evaluate the performance overhead of adding PB-JARS to an Apache proxy. We chose Web Polygraph [29] as our benchmarking tool because it is a widely accepted proxy performance benchmark [31], and most importantly, it has the capability to simulate real content traffic, which is crucial for our content-filtering type performance evaluation. We analyzed the overhead added by PB-JARS on Java binary requests and then compared the proxy's overall performance with and without PB-JARS under different request rates.

6. 1. Customize Web Polygraph for Content-filtering Type Performance Evaluation

Web Polygraph is designed specially for caching proxy benchmarking. Its latest workload model, Polymix-4, includes many key web traffic characteristics, such as a synthetic workload composed of various content types, specified request rates and inter-arrival times, a mixture of cache hits and cache misses, etc. Table 1 summarizes the content types that the Web Polygraph server uses for benchmarking. The server hosts mainly three content types (i.e. image, HTML, download.) with specific file extensions. All other content types are included in the 'other' type without specifying file extensions. The size models of content types listed in Table 1 are generated by analyzing unique file size transportation from the web proxy log [28].Table 2 lists the default parameters of the polymix-4 workload model. For the other parameters not stated, we use the default values of polymix-4.

Table 1. Content types of Web Polygraph workload

Type	Percentage	Reply Size Distribution	Cachability	Extensions
Image	65%	Exponential (4.5KB)	80.00%	.gif, .jpeg, and .png
HTML	15%	Exponential (8.5KB)	90.00%	.html and .htm
download	0.5%	Lognormal(300KB, 300KB)	95.00%	.exe, .zip, and .gz
Other	19.5%	Lognormal(25KB, 10KB)	72.00%	

Table 2. Default parameters of the workload model

Client request rate	0.4/secs
Number of transactions per connection	Zipf (64)
Request types	IMS: 20% Reload: 5% Basic 75%
Server delay	40 millisecond/packet
Server think time	Normal distribution (2.5, 1)

6. 2. Internet Traffic Model of Java Binaries

In order to test PB-JARS, we need to generate real Java binary file traffic in the experiment. However, to our knowledge, no specific analysis of Java binary traffic has ever been published in the literature. So we conducted our own research on Java binary file traffic.

We used the raw access logs provided by the IRCache project [32]. These logs were collected on a daily basis between July 11th, 2004 and August 11th, 2004 from a number of proxy servers located at various educational and commercial institutions throughout the United States [32]. The access logs recorded the clients' request information for HTTP objects and consist mainly of the entries of time, duration, client address, result codes, bytes, request method, URL, type and etc. A Java JAR file is the digital signed and compressed form of a bunch of java class files. JAR files that contain applets can be executed in the web browser the same way as applets. Therefore, Java binary file requests can be identified by their file extensions (i.e. .class, .jar) in the corresponding URL entries and from the 'result codes' entry we can determine the status of the specific transaction.

Table 3 shows the statistic summary for the raw data set. The *successful requests* mean the transactions whose HTTP result code is "200/OK". The *unique successful requests* mean that repeated requests for the same Java binaries are removed. From this table we can calculate that the class file requests take 0.08% of the total requests, 27.08% of the class file requests are successful requests, 59.6% of the successful requests are unique file requests and 16.4% of the successful requests are directly served by cached classes. Jar file requests take 0.04% of the total requests, 30.4% of the Jar file requests are successful requests, 24.9% of the successful jar file requests are unique file requests, and 41.7% of the successful jar file requests are served by cached jar files.

Table 3. Summary of access log characteristics
Duration: 08/11/ 2004 ~ 09/10/ 2004 (Total Requests Count: 117790462)

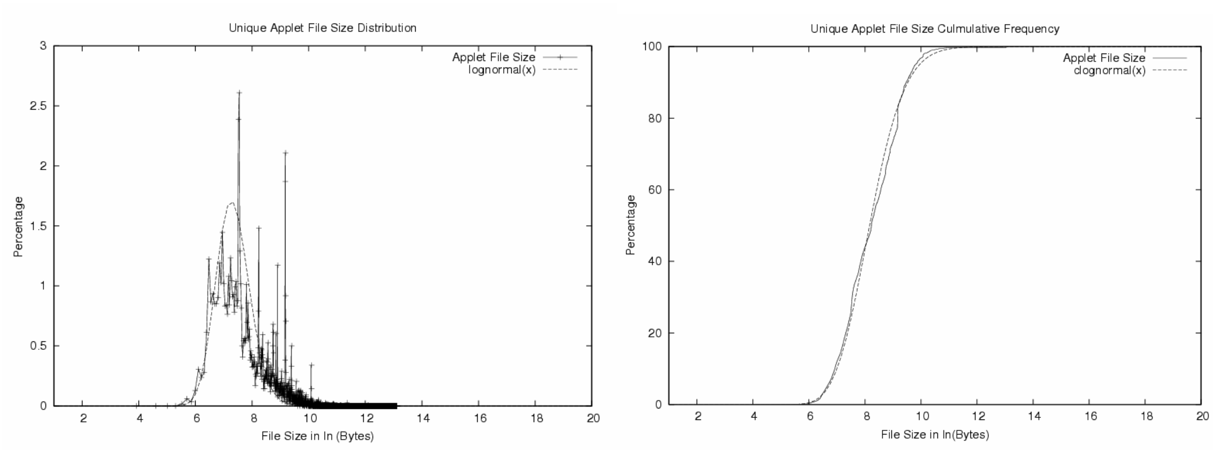
Java Binaries	**Class**	**Jar**
Total request count	95054	48743
Total successful requests count	25737	14826
Total successful requests served directly from cache	4223	6176
Total unique successful requests count	14646	3685
Mean bytes of successful requests	7176.85	38945.23
Median bytes of successful requests	3725	8143
Mean bytes of successful unique file	7085.56	64322.77
Median bytes of successful unique file	3775	15644

Table 4 lists the main reasons that cause the failure of Java binary requests. Strictly speaking, Not Modified (304) is not a failure. It is the response from the server to the client query about whether the file cached in the client cache has been out-of-date since last time the client cached the file.

Table 4. Reasons that Java binaries requests fail

Java Binaries	**Class**	**Jar**
Service Unavailable (503)	32.6%	78.7%
Not Modified (304)	31.3%	9.3%
Not Find (404)	27.1%	4.4%
Others	9.0%	7.6%

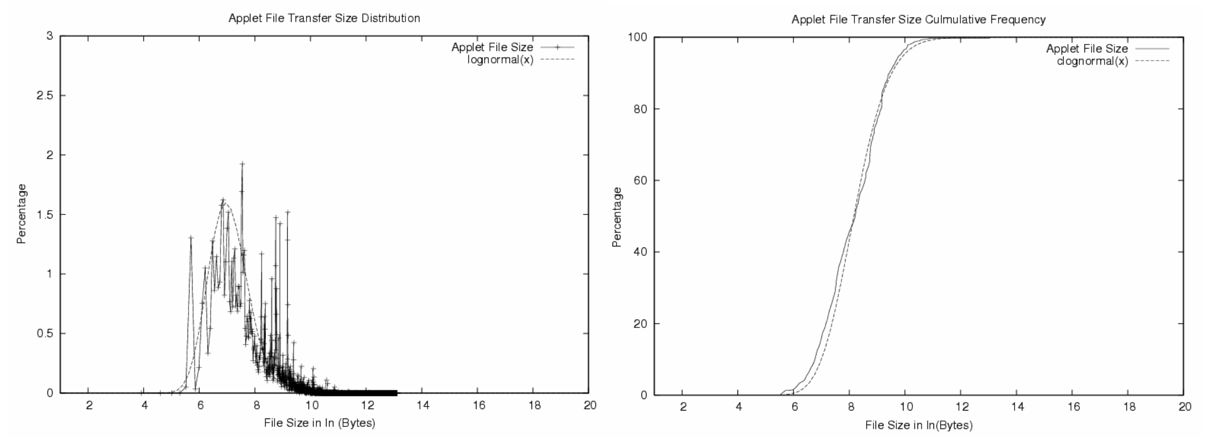
Figure 8 shows the distribution of all successful unique class files in the data set. Figure 8a compares the distribution with the synthetic lognormal distribution (the dashed line) with parameters μ= 7.61 and σ = 0.66. Figure 8b is the corresponding cumulative frequency plot. As we can see, the class file size distribution is close to log normal distribution.



(a) Frequency **(b) Cumulative frequency**

Figure 8. Unique class file size distribution

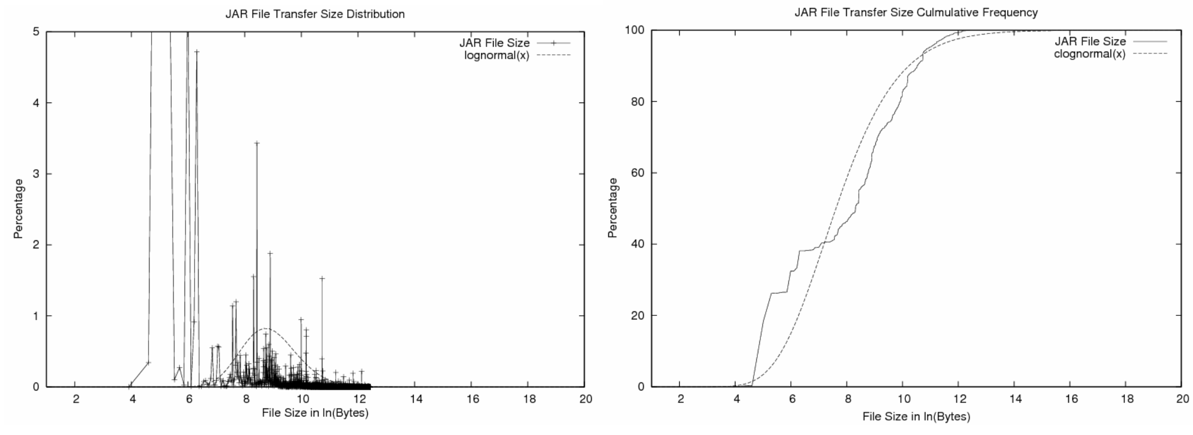
Figure 9 shows the transfer size distribution of all successful class files in the data set. Figure 9a shows the distribution is close to the lognormal distribution (the dashed line) with parameters $\mu = 7.06$ and $\sigma = 0.70$. Figure 9b is the corresponding cumulative frequency plot.



(a) Frequency **(b) Cumulative frequency**

Figure 9. Successful class file transfer size distribution

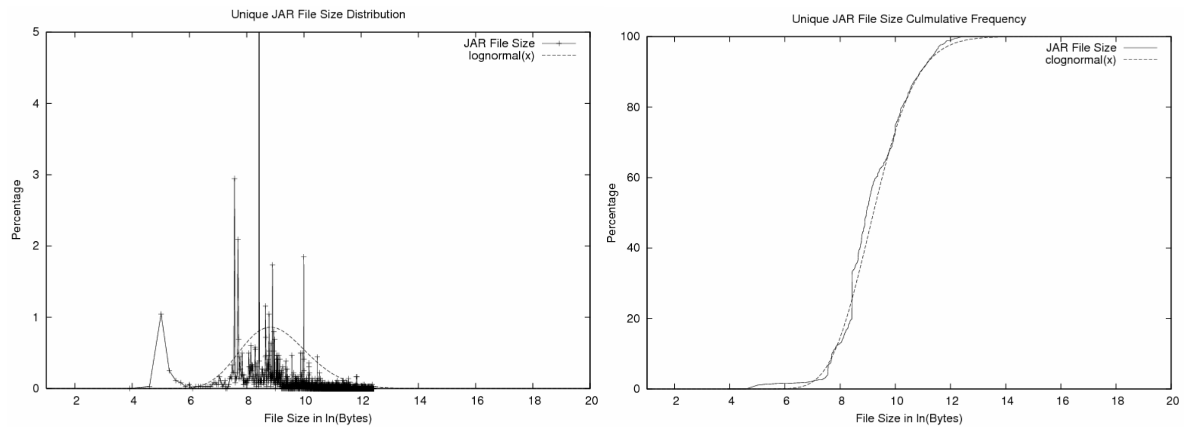
Figure 10 shows the distribution of all successful JAR files in the data set. Figure 10a compares the distribution with the synthetic lognormal distribution (the dashed line) with parameters $\mu= 8.69$ and $\sigma = 0.37$. Figure 10b is the corresponding cumulative frequency plot. As we can see, the JAR file size distribution is not as regular as class files. There are several very popular Jar files with the size between 4k and 5k bytes. The overall trend tends to be a normal distribution.



(a) Frequency (b) Cumulative frequency

Figure 10. Successful JAR file transfer size distribution

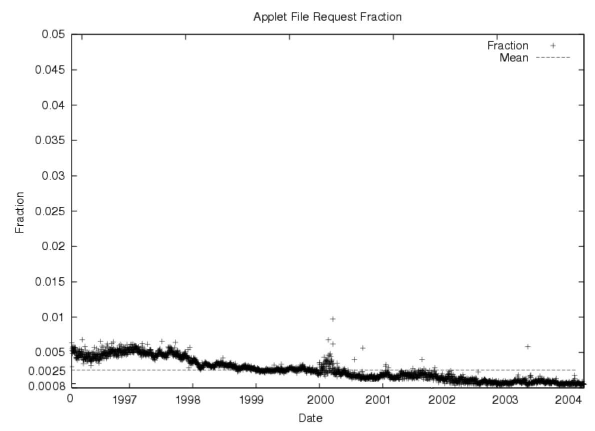
Figure 11 shows the distribution of successful unique JAR files in the data set. Figure 11a compares the distribution with the synthetic lognormal distribution (the dashed line) with parameters μ= 8.97 and σ = 0.36. Figure 11b is the corresponding cumulative frequency plot. We can see that the overall trend follows a normal distribution.

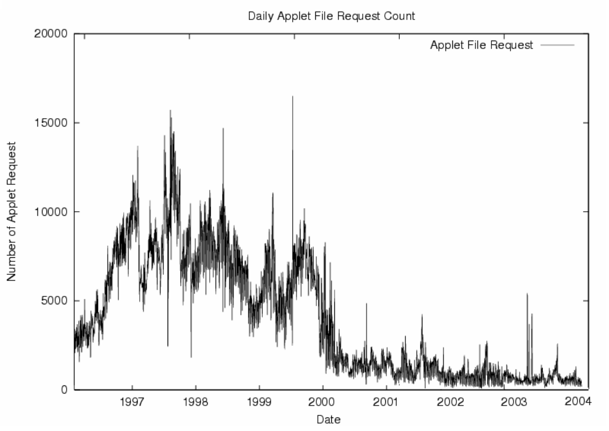


(a) Frequency (b) Cumulative frequency

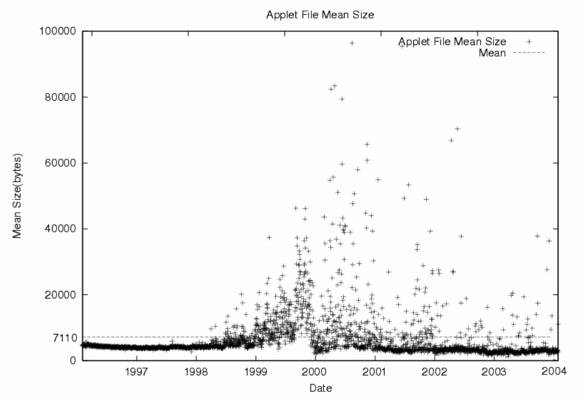
Figure 11. Unique JAR file size distribution

Based on the data availability, we also analyzed the IRCache daily summary report of applet file (which has the file extension .class) requests in an extended period from 1997 to 2004. Figure 12 shows the daily applet file request ratio (12a), daily number of applet requests (12b), and the daily mean size (12c) respectively. The dashed line is the mean value. From these figures we can see that the applet file requests rate started from 0.6% in the year of 1997, dropped after the year of 2002 and tends to be stabilized at 0.08%. We believe the reason behind this is that, at the first several years after Java applets is introduced, Java applets is a very popular media for many purposes such as internet transaction, user interaction, playing animation, and etc. As new more convenient and easy-to-learn software tools are invented, parts of the functionalities are replaced. For example, more and more people tend to use Macromedia Flash for graphical interactions. Instead, Applets are becoming more stabilized and specialized.





(a) Daily class file request ratio **(b) daily number of class file requests**



(c) Mean size of daily class file requests
Figure12. Class file requests daily stat (1997~2004)

In summary, according to our analysis, 0.08% of Web requests are Java class file requests, 40.4% of which are repeated requests. Its file size distribution conforms to lognormal distribution. JAR file requests take about 0.04% of all Web requests, 75.1% of which are repeated requests. Its file size conforms to the same distribution. Table 5 shows the Java binary traffic model we used in our experiment. We took 0.08% and 0.04% from the 'other' type of Polymix-4 workload as the Java jar file and class request percentage respectively. Because we force the proxy cache to cache all rewritten Java binaries, we use 100% for its cachability. We built up the Java binary file generation database on the server side with real java binary files downloaded randomly from the Internet and with the file sizes according to the unique file size distribution. At the client side, we configure it to generate 40.4% and 75.1% of repeated Java binary requests. For all the other parameters such as object life cycle, etc, we use the same as the 'other' type.

Table 5. Java binary traffic model

Type	Percentage	Reply Size Distribution	Cachability	Extension
Class	0.08%	Lognormal(7.61, 0.66)	100%	.class
JAR	0.04%	Lognormal(8.89, 0.37)	100%	.jar

6.3 Experiment Setup

We use one pair of client and server in our experiments. The hardware configuration is as following: the client runs on a PC with 756Hz AMD CPU and 256MB. The server and Apache proxy run on PCs with 2GHz Pentium4 CPU and 640 MB memory, respectively. All the three machines are connected through a 100M network. In addition, we installed a name service and a network time(ntp) service on the proxy server, which are required by the Web Polygraph benchmark. The proxy server software that we used is Apache version 2.0. At the time of our experiments, the garbage collection functionality has not been implemented on Apache 2.0 but the same functionality works properly on Apache 1.3. To confine the disk cache used, we rewrote the garbage collection part in version 1.3 and ported it to Apache 2.0.

Web Polygraph uses synthetic clients and servers. Given a specified Peak Request Rate (briefly PRR), a number of clients will be created to generate the specified PRR and an according number of servers will also be created to response to the requests. Therefore, by varying the PRR, we are actually changing the number of clients. In our proxy configuration, we set the max client number that the Apache can sustain concurrently to be 250, which is approximately according to 30xacts/sec PRR. All other Apache proxy parameters are by default.

6.4 Experiment Results

In our experiments, we first repeated the experiment in Figure 4 in order to compare the performance before and after. Then, under real traffic model, we examined the overhead added to Java binary file requests, how PB-JARS affects the proxy's overall performance, and how the proxy cache size affects the overhead.

6.4.1 The Performance after Applying PB-JARS. We randomly downloaded Java binaries from the Internet and rewrote them with the security policy *limitMem*, which has the worst performance overhead shown in Figure 4. Figure 13 shows the performance result as compared with each binary file's size. The number above each bar refers to hit times, which is the number of events that PoET needs to add monitoring code to. As we can see that after applying PB-JARS, in the worst case, the rewriting overhead is about 0.5 seconds. That is because in PB-JARS we modified PoET to allow it to work in the background as a service daemon with security policies preloaded. Therefore, the initialization overhead is eliminated. Figure 13 also shows that the performance overhead increases gradually as the file size and hit times increase. As shown in our previous work [10,11], the rewriting time is also related to the binary file size, complexity of the security policy and hit times.

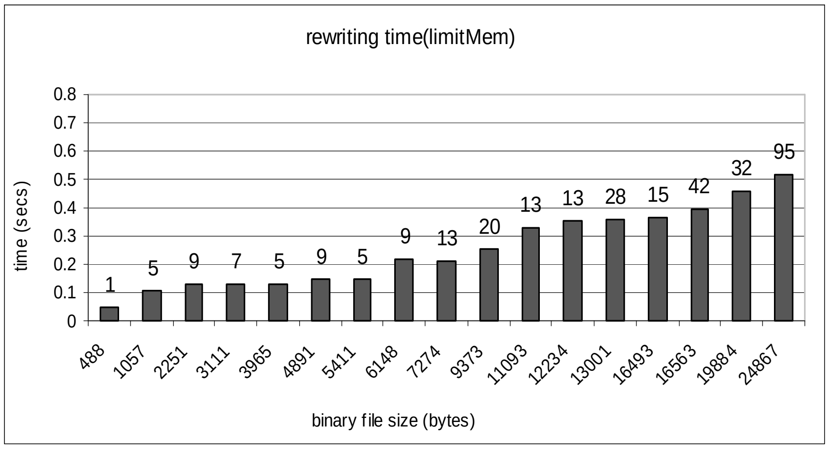
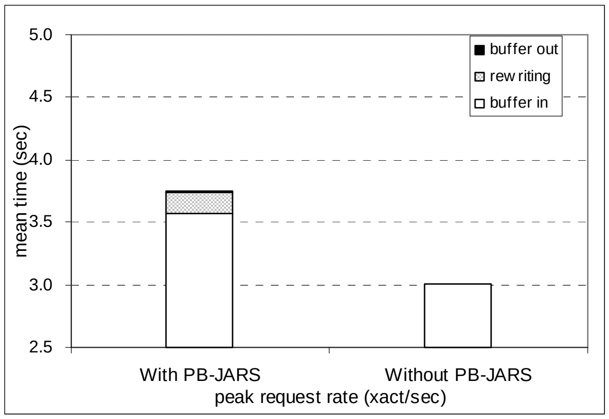
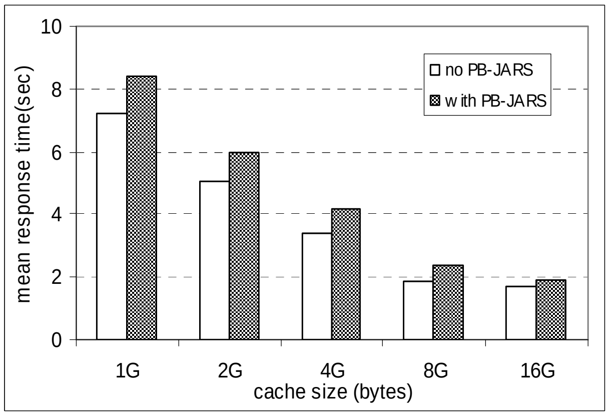


Figure 13. Performance evaluation after applying PB-JARS

6.4.2 The Overhead of PB-JARS on Java binary requests. To measure the overhead of PB-JARS on Java binary requests, we kept a log on the proxy server to record the Java binary cache miss time, in which case the requested Java binary file is not cached and the proxy forwards the client's requests to the remote server until it gets the response data back and sends it to the client. With PB-JARS, this time can be further divided into: *buffer in*, which is the time from the point proxy gets the client's request for a Java binary file until the receiving buffer buffers the entire file (for jar files, it also includes the time to extract the jar file to a class file); *rewriting*, which is the time that the binary rewriter takes to rewrite the file and *buffer out*, which is the time in that the rewritten file in the output buffer is cached and sent to the client (for jar files, it also includes the time to recompress multiple class files into one single jar file). Figure 14a shows the comparison result. We use the PRR 30 xacts/sec for the experiment, under which the proxy is fully loaded. As we can see, the overhead added to the Java binary request is mainly caused by *buffer in* time. This is because the binary rewriter requires the entire file to be cached before it can rewrite it, while, instead, without PB-JARS the file will be cached and then sent to the client in the unit of chunks of the file. The rewriting time adds relatively little to the overhead. In our experiments, we used the security policy *limitMem*. The buffer out time is relatively very small that can barely be seen from the figure.

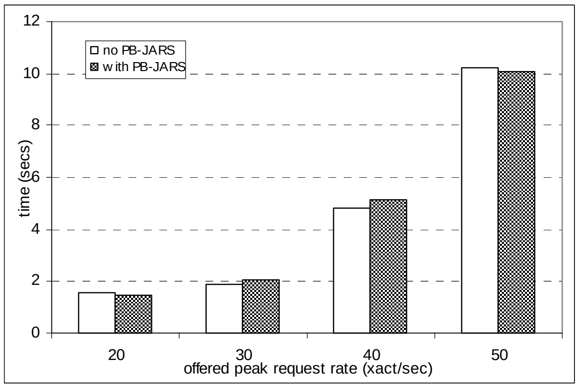
To see how varying the cache size affects the overhead added by PB-JARS on the Java binaries, we logged each transaction for a Java binary request on the client side. The time measured thus is the mean value of both cache miss and cache hit. Figure 14b compares the result with and without PB-JARS under different cache sizes. The PRR used is 30 xacts/sec. We can see that as the cache size increases and so the number of cached rewritten Java binary files and cache hits increases, the mean response time difference between with and without PB-JARS is decreasing, which means the overhead caused by PB-JARS as shown in Figure 14a is amortized by the increased cache hits.



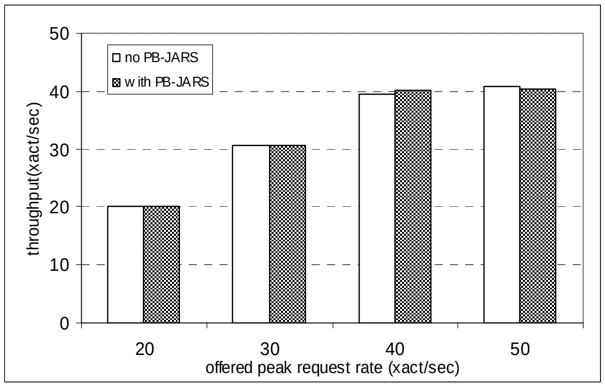


(a) Time measured at the proxy (with 16G cache) (b) Mean response time under different cache size
Figure 14. Overhead added by PB-JARS on Java binary request (PRR 30 xacts/sec)

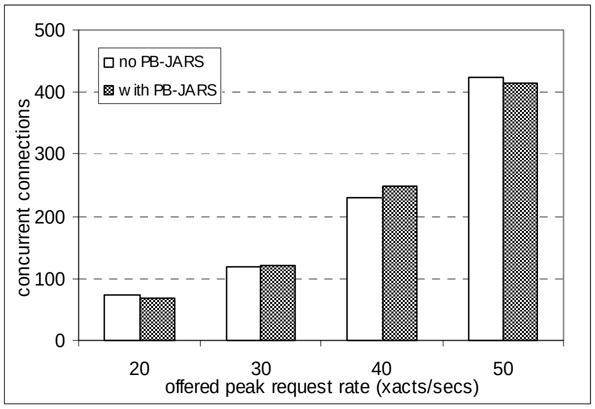
6.4.3 The Overall performance. To find out how PB-JARS affects the proxy's overall performance, we compared the proxy's performance with and without PB-JARS under different PRRs that the proxy server becomes from lightly loaded to badly overloaded. The result is shown in Figure 15. As the PRR increases, the mean response time and throughput increases. After the 30 xacts/sec, the proxy becomes overloaded: the response time increases dramatically; The concurrence level increases while the throughput tends to be stabilized at 40 xacts/sec, which means client requests keep being queued at the proxy; the error rate rises up to be over 1% and over 90% of the errors are caused by connection time out; The CPU usage tends to be stable at 12%. We can see that except for a little overhead on the server's CPU usage (which is less than 1%), the performance difference between with and without PB-JARS is very small and within the measurement error range.



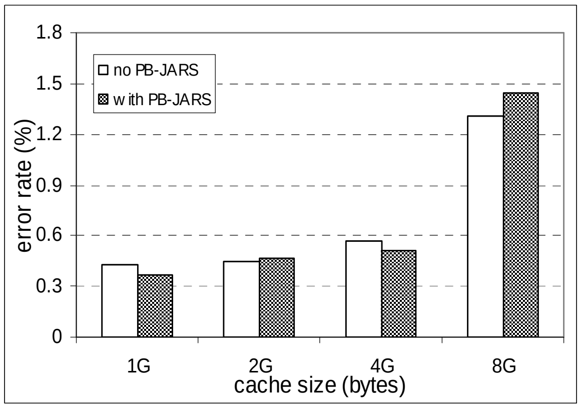
(a) Mean response time



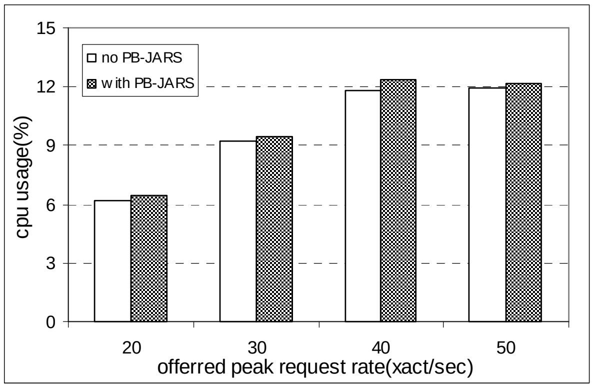
(b) Throughput



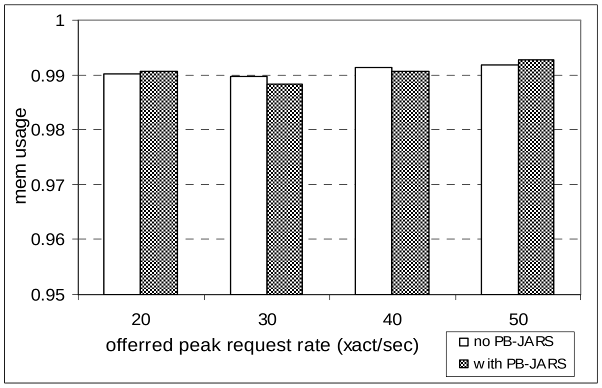
(c) Concurrence level



(d) Error ratio



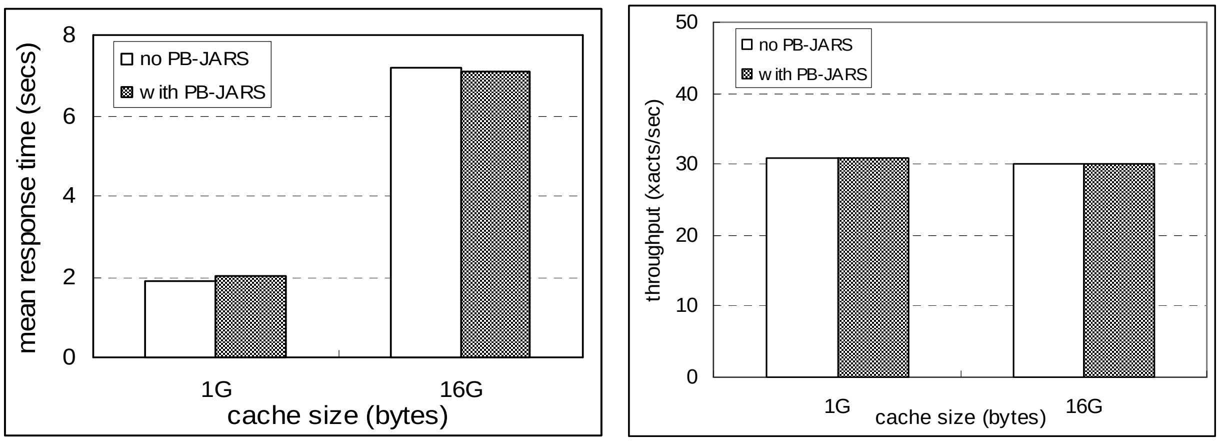
(e) CPU usage



(f) Memory usage

Figure 15. Apache proxy performance under different request rates (16G Cache Size)

In the above experiment we set the cache size to 16G bytes. The reason is that under heavy load (e.g. 50 xacts/sec PRR) and at the same time with a small cache (e.g., 1G), the Apache proxy's behavior becomes very poor and nondeterministic. This is consistent with Cao's result [24]. However, to verify that our above result is also consistent with a small cache, we give the overall performance under 30 xact/sec PRR with 1G cache size as shown in Figure 16. As we can see that with different size of cache, the overall performance with and without PB-JARS is still almost the same.



(a) Mean response time **(b) Throughput**

Figure 16. Overall performance under 30 xacts/sec PRR

Therefore, to summarize it, PB-JARS almost has no impact on the proxy's overall performance under different workloads and cache sizes. The overhead added specially on Java binary requests can be amortized by increasing the cache size.

7. Related Work

Distributed Virtual Machine (DVM) [1] is the most related work to our work. Its idea of locating JVM security services at a centralized server, and improving the manageability and security requirements of large, heterogeneous clusters of networked computers is similar to our work. The difference between DVM and our work is that DVM thinks of an enterprise network as a whole with uniform security, and therefore deploying JVM security at each client side will cause the problems of discretionary security and redundant management. Thus DVM proposes to move JVM security functionality from the client side to a centralized network access point. Our work, however, admits that each client has different security requirements even within the same enterprise networks, but the current JVM security implementation is not trustable and has its own inherited limitation such as a rigid access control policy. Therefore, we leave the current JVM security at each client side and build up a complementary second-line defense at the centralized network server. In addition, DVM re-implemented the JVM security model at server side, which is still unable to break away from JVM's rigid security policy model. However, our work builds up a frame to leverage existing binary rewriting techniques that support more flexible security models. The newest binary rewriters can be easily plugged in and applied quickly.

Applet Trap [33] is a commercial anti-virus software. It wraps applets in security monitoring code before the gateway server passes the applets on to the requesting client computer. Different from many other content filtering or blocking software, the applets run their original code along with a monitoring wrapper which looks ahead into the applets' behavior to determine if the action, applets will take, matches any behavior defined in the administrators' policy as malicious (such as reformatting the hard disk). The execution of the monitoring code doesn't need the host system or external software support. There is little material available about how AppletTrap rewrites the Java applet code, we suspect that AppletTrap can only add checking code to block the predefined malicious instructions while PB-JARS enforces more comprehensive security policies.

Many other solutions to Java binary security use behavior-block methods, such as Fanjan's anti-virus software [34]. In this method, a monitoring system is employed to intercept the system calls made by a Java binary. The system determines whether its resource requests are granted or rejected access. However, with this method, benign and useful functionalities are also blocked in many cases.

8. Conclusion

In this paper, we began by introducing binary rewriting techniques to security and discussed issues when using a binary rewriter as a security tool. We then presented our system called PB-JARS, a security system that utilizes binary rewriting. Experiments with PB-JARS demonstrate that binary rewriters combined with Web proxy caching are a good way to improve end host JVM security. We show that Java binaries are a small fraction of Internet requests and have high repeated request rates. Consequently, adding binary rewriting to web caching systems can be very efficient in improving end host security with minimal patching inconvenience on end-users.

We conclude this paper with a focus on the research we have conducted and where we might expect the research to progress in the future. We selected PoET because of its intrinsic simplicity and its availability.

We believe that as new binary rewriters become available for more complex binaries, the research premise of combining web proxies with binary rewriters will grow in impact. The promise of strengthening our network perimeter with computer security tools will become even more pressing as threats continue in the future. We believe the ideas presented in this paper will resonate in the computer security community as a tremendously potent and easy way to improve end host security in distributed systems.

References

[1] Emin Gün Sirer, etc, Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *Proceedings of the Seventeenth Symposium on Operating Systems Principles,* pages 202-216, Kiawah Island, South Carolina, December 1999.

[2] M. Prasad and T. Chiueh, A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks, *Proceedings of Usenix Annual Technical Conference,* San Antonio, TX, June 2003

[3] Greg Andrews, Link-Time Optimization of Parallel Scientific Programs, University of Arizona Seminar Abstract, Nov, 2002.

[4] R. Hastings and B. Joyce, Purify: A tool for detecting memory leaks and access errors in C and C++ programs, *Proceedings of the Winter 1992 USENIX Conference*, Berkeley, CA, January 1992, pp 125–138.

[5] F. B. Schneider, etc, A Language-Based Approach to Security, *Informatics*, 2001, pp. 86-101.

[6] U. Erlingsson, F. B. Schneider, IRM Enforcement of Java Stack Inspection, *IEEE Symposium on Security and Privacy,* Oakland, California, May 2000.

[7] Li Gong, Inside Java 2 platform Security architecture, API design, and Implementation, *Addison-Wesley Longman Publishing Co., Inc.*, 1999.

[8] D. Evans and A. Twyman, Flexible Policy-Directed Code Safety, *1999 IEEE Symposium on Security and Privacy*, Oakland, California, May 9-12, 1999.

[9] Ú. Erlingsson and F. B. Schneider, SASI enforcement of security policies: A retrospective, *Proceedings of the New Security Paradigms Workshop,* Ontario, Canada, September 1999, pp87-95.

[10] Y. Song and B. D. Fleisch, Rico: A Security Proxy for Mobile Code, *Journal of Computers and Security Elsevier Advanced Technology*, Elsevier Press, Volume 23, Issue 4, 2004, pp. 338-351

[11] Y. Song and B. D. Fleisch, Sandboxing Mobile Code from Outside the OS, *in the 19th ACM Symposium on Operating Systems Principles*, Work in Progress Session, Bolton Landing, New York, 2003.

[12] Libsafe library: http://www.bell-labs.com/org/11356/ libsafe.html.

[13] R. Wahbe, et al., Efficient Software-Based Fault Isolation, *Proceedings of the Symposium on Operation System Principles*, 1993.

[14] D. C. DuVarney, V.N. Venkatakrishnan and S. Bhatkar, SELF: a Transparent Security Extension for ELF Binaries, *New Security Paradigms Workshop,* Ascona, Switzerland, August 2003.

[15] T. Chiueh and F. Hsu. Rad: A compile-time solution to buffer overflow attacks, *21st International Conference on Distributed Computing*, Phoenix, Arizona, April 2001, pp 409.

[16] C. Cowan et al., Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998, pp 63-78.

[17] S. Debray, W. Evans, R. Muth, B. D. Sutter, Compiler Techniques for Code Compaction, *ACM Transactions on Programming Languages and Systems. ACM Press*. Vol. 22 (2). 2000. pp. 378-415

[18] C. Cifuentes, M. Emmerik, UQBT: Adaptable Binary Translation at Low Cost, *Computer*, Vol 33, No 3, March 2000, IEEE Computer Society Press, pp 60-66

[19] T. Romer, et al., Instrumentation and Optimization of Win32/Intel Executables Using Etch. *Proceedings of the First USENIX Windows NT Workshop*, Seattle, WA, August 1997.

[20] Byte Code Engineering Library (BCEL) http://jakarta.Apache.org/bcel/manual.html

[21] A. Srivastava and A. Eustace, ATOM: A system for building customized program analysis tools, *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, May 1994.

[22] L. R. James and E. Schnarr, EEL: Machine independent executable editing, *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[23] B. Buck and J. K. Hllingsworth, An API for runtime code patching, *the International Journal of High Performance Computing Applications*, vol. 14, no. 4, 2000, pp. 317-329.

[24] J. Almeida and P. Cao. Measuring Proxy Performance with the Wisconsin Proxy Benchmark, Technical Report 1373, Computer Sciences Dept, Univ. of Wisconsin-Madison, April 1998.

[25] CERT security report #237777, http://www.kb.cert.org/vuls/id/237777.

[26] CERT security report #447569, http://www.kb.cert.org/vuls/id/447569.

[27] Sun Java documentation: http://java.sun.com/

[28]The Mesurement Factory Document about server workload file size. http://www. measurement-factory.com/docs/FAQ/pmix4-reply-size-distr/

[29] Web Polygraph: http://www.web-Polygraph.org

[30] Apache Document. http://www.apache.org/

[31] The Fourth TMF Cache-Off: http://cacheoff.measurement -factory.com/

[32] IRCache Project: http://www.ircache.net/

[33] AppletTrap: www.trendmicro.com.au/product/isap/

[34] Finjan Software Documentation: http://www.finjan.com

[35]Access Control documentation: http://groups.northwestern.edu/exec/htmldocs/NT_Security. html

Yougang Song is a Ph.D. student at the University of California, Riverside. He received the M.S. degree in computer science from the University of Texas at Dallas in 2002 and the M.E. degree in computer engineering at Chinese Academy of Sciences in 2000. His current research interest includes distributed systems, security and file systems.

Brett D. Fleisch joined the National Science Foundations Directorate for Computer and Information Science and Engineering (CISE) / Computer and Network Systems (CNS) as Program Director in April 2004. Dr. Fleisch currently is an IPA from the University of California, Riverside where he has been since 1992. At the University of California, Riverside, Fleisch serves as Associate Professor of Computer Science and Engineering at the University of California, Riverside. He acquired the Ph.D. in computer science from UCLA in July 1989. He received the B.A. degree in computer science at the University of Rochester, and the M.S. degree in computer science at Columbia University in 1981 and 1983, respectively. He has served as a consultant and employee at Xerox Corporation's Webster Research Center, IBM Corporation's Thomas J. Watson Research Center, the Educational Testing Service in Princeton, New Jersey, the College Board (West Coast offices) and the State of California, Department of Motor Vehicles. He is a member of the ACM, IEEE Computer Society, and USENIX.