SoftwarePot : An Encapsulated Transferable File System for Secure Software Circulation

Kazuhiko Kato Yoshihiro Oyama

Technical Report ISE-TR-02-185
Institute of Information Sciences and Electronics
University of Tsukuba

January 20, 2002

SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation

Kazuhiko Kato[†‡] Yoshihiro Oyama[‡]

† Institute of Information Sciences and Electronics
University of Tsukuba
Tennoudai 1-1-1, Tsukuba, Ibaraki 305-8573, Japan
‡ Japan Science and Technology Corporation
Email: {kato,yosh}@osss.is.tsukuba.ac.jp
http://www.osss.is.tsukuba.ac.jp/~kato/ and /~yosh/

January 20, 2002

Abstract

We have developed a general approach to enable secure circulation of software in an open network environment such as the Internet. By software circulation, we mean a generalized conventional software distribution concept in which software can be transferred even in an iterative manner such as through redistribution or using mobile agents. To clarify the problem that arises when software is circulated in an open network environment, we first considered a simple model for unsecure software circulation and then developed a model for secure software circulation (SSC). In the SSC model, we extended the sandbox concept to include its own file system and to have the ability to be transferred via a network. In this sense, our approach is characterized by an encapsulated, transferable file system. We describe how the SoftwarePot system was designed to implement the SSC model, and discuss the implications of experimental results that we obtained during the implementation.

Keywords: Open network environment, Internet, file system, software distribution, mobile code, mobile agents, secure computing.

1 Introduction

One of the most notable features of the Internet is that it is a truly open environment. Not only is it being constantly extended worldwide, no one can fully control or determine who the users are, or what software and content are exchanged through it. These features are in stark contrast to traditional, closed computing environments such as batch, TSS, LAN, or personal systems. Throughout the history of computing systems, the computing environment has almost always been closed, so designers of system software have implicitly assumed a closed environment was the norm. The worldwide spread of the Internet occurred within a relatively short portion of the history of computing system development, so there was little time for designers of system software to

fully anticipate the issues that would arise when closed environments became open. Though the environment has drastically opened up, we still use system software whose basic design is based on an assumption of a closed environment. Thus, current computer systems can often be characterized as *putting new wine in old bottles.*

If a network environment is closed and users cannot access an open network environment such as the Internet, malicious users and the files created by such users are far less likely to exist. Unfortunately, in the Internet environment, we *cannot* assume this. Thus, obtaining files, particularly software that includes executable or interpreted code, from the Internet can be a risky activity. One approach now being used to lessen the risk is to use a code-signing technique, such as Microsoft's Authenticode. Being able to obtain information about a distributor's identity is useful, but users still face a non-technical problem as to whether they can trust the distributor. A promising technical approach to solve this problem is to create a "sandbox" and encapsulate risky effects in a limited, controllable environment separate from the users' ordinary environment. This approach has been adopted in many systems, such as Java, SFI [21], Janus [8], MAPbox [2], and SubDomain [7].

In this paper, we discuss a general approach that we have developed to allow secure software circulation in the Internet environment. By software circulation, we mean conventional software distribution, even where software is transferred in an iterative manner such as through redistribution or by mobile agents. To clearly define the problem that arises when software is circulated in an open network environment, we first consider a simple model for *unsecure* software circulation, then propose a model for *secure* software circulation. In the secure software circulation model, we extend the sandbox concept to include its own file system and network transferability. In this sense, our approach is characterized by an encapsulated, transferable file system. We designed the system named *SoftwarePot* based on the approach. In the SoftwarePot system, all circulated files are encapsulated into a transferable file system called *pot*. We say that a pot is closed if it is perfectly self-contained and does not need to input or output files other than those contained in the pot. The execution of codes included in a closed pot is guaranteed to not affect the user's file system outside of the pot. Sometimes, though, it is necessary to affect the user's file system in a securely controlled way. For this purpose, SoftwarePot provides a mechanism to map between the files in a pot and those in the user's file system. By associating the mapping requirement of a pot, the user of the pot can know which files need to be accessed before execution. This helps to develop security policies for the executing sites. Furthermore, SoftwarePot incorporates a lazy file transfer mechanism, by which we mean that only references to files are stored in a pot, instead of the file entities themselves, and each file entity is automatically transferred only when requested during execution. These mechanisms allow use of the SoftwarePot system as a general tool to circulate software in the Internet environment.

The rest of the paper is organized as follows. Section 2 describes the models for *unsecure* and *secure* software circulation. It also shows that many useful software transfer patterns can be naturally modeled by combining the basic operations of the models. Section 3 explains how the SoftwarePot system is designed to implement the secure software circulation model. Section 4 shows some of our experimental results obtained during the system implementation. Section 5 discusses related work. Finally, Section 6 concludes the paper.

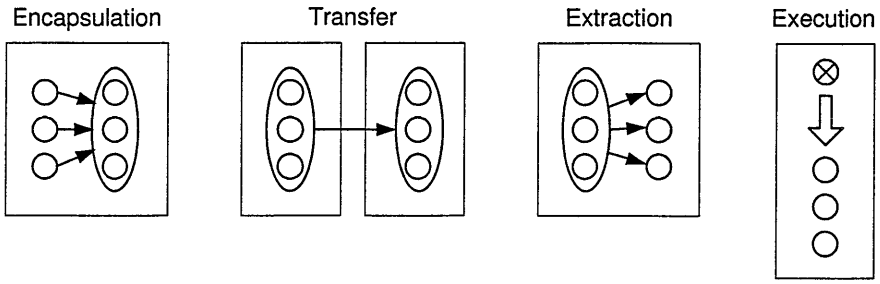


Figure 1: The four basic operations of the unsecure software circulation model.

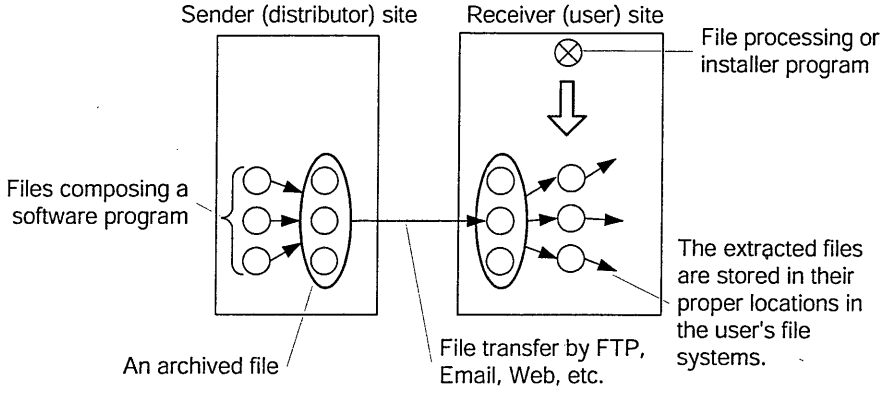


Figure 2: Software distribution.

2 Software Circulation Models

To clarify the concept of secure software circulation, we first present a model for *unsecure* software circulation, which is a generalization of software distribution as conventionally performed. We then present a model for *secure* software circulation and explain how it can be used.

2.1 A Model for Unsecure Software Circulation

Software circulation is the term we use to refer to a generalization of the "software distribution" concept. Software distribution usually means unidirectional, one-time, one-to-many distribution of a software package. By relaxing these properties, that is, by making software distribution multidirectional, multi-time, many-to-many, we obtain the concept of software circulation. Software circulation is composed of four basic operations: encapsulation, transfer, extraction, and execution. For each operation, we use a graphic representation as shown in Fig. 1.

- The encapsulation operation is applied to one or more files and creates an archive file that encapsulate those files.
- The transfer operation moves an encapsulated file from a source site to a destination site.
- The extraction operation is the inverse of the encapsulation operation: one or more files are extracted from an archive file. Extracted files are stored somewhere in the file system of the site where the operation is performed. File allocation, naming, and access-control setting are performed at that time.

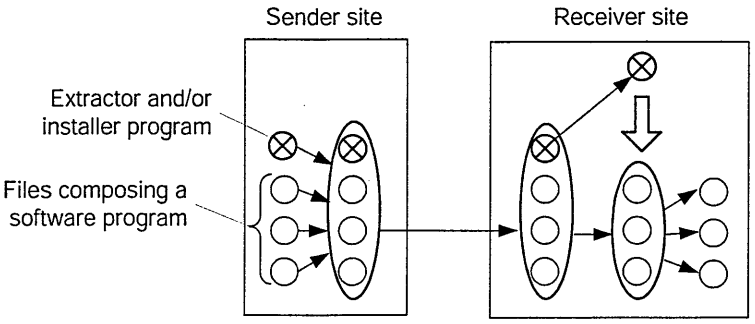


Figure 3: Transfer of a self-extraction file.

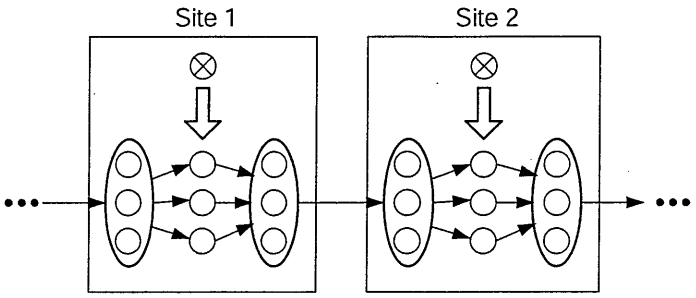


Figure 4: Workflow.

- In the execution operation, an executable program (shown as the circle containing a cross in Fig. 1) is executed. During the execution, files (the circles without a cross) may be inputted, outputted, or modified.

By combining these operations, we can represent typical software circulation scenarios. Figure 2 illustrates software distribution. At the sender site (e.g., a distributor of a software package), the files of a package are encapsulated into a archive file by using an archiving program. The archived file is transferred from the distributor's site to a receiver (user) site. At the user's site, the archive file is extracted and stored in the user's file system. Figure 3 shows a variation; an extraction and/or installation program are included within the archive file. This is flexible and convenient, since the distributor can do anything during the installation by describing it in the installation program, even if the user does not have an extraction or installation program. Figures 4 and 5 show iterative styles of software circulation. In the former, a program is applied to circulated files and this models a workflow system. In the latter, a program included in the circulated file is executed and this models a mobile agent system.

The presented model is unsecure in the following aspects:

- In the encapsulation operation, a malicious person may lay traps in the archive file; for example, by including files that perform malicious things when executed or that overwrite existing files for malicious purposes when extracted.
- During the transfer operation, files may be maliciously interrupted, intercepted, modified, or fabricated [18].
- During the extraction operation, extracted files are stored in the user's file system, and their file storage is allocated and named and their access control is set at that time. These operations

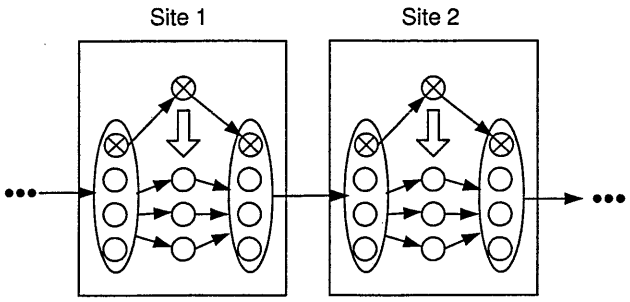


Figure 5: Mobile agent.

are critical for the management of the user's file system, and are quite dangerous if they are not done according to the user's management policy.

- The execution operation, needless to say, involves a potential lack of security. All effects of the execution on the computer resources should be controlled in accordance with the user's security policy.

For these reasons, we call this model the *unsecure software circulation* (USC) model.

2.2 A Model for Secure Software Circulation

Now, we consider a model for *secure software circulation* (the SSC model). To start with, we should explain what we mean here by "secure." First, it is generally accepted that an absolutely "secure" system is virtually impossible to design. This model and our implemented system based on this model (described in the next section) is simply more secure than the USC model regarding security-related issues that are more easily handled in a systematic way. Second, the unsecure aspects of the *transfer* operation are not limited to software circulation; they are also issues of concern in general information transfer. However, we will not discuss these issues any further in this paper.

The central idea of the SSC model is that we distinguish the *inner* and the *outer* parts of circulated software during both the execution and the encapsulation. To enable this, we introduced two concepts into the model. First, we do not extract files from archives; instead, we extend the concept of using archive files as a file system. Second, we enable program execution *within* a distinct virtual address space associated with the archive space. The SSC model integrates the two spaces—the virtual address space and the file system—into one to create a sandbox called a *pot*. Each pot has its own, distinct view for both a virtual address space and a virtual file space.

We define the basic operation of the SSC model as follows (see Fig. 6 for a graphical representation).

- The encapsulation operation is applied to one or more files and stores them in a pot. During storing, each file is given a unique pathname compatible with the Unix file system, and the attribute-information, such as time of last modification and executability, is recorded.
- The transfer operation moves an encapsulated file from a source site to a destination site.

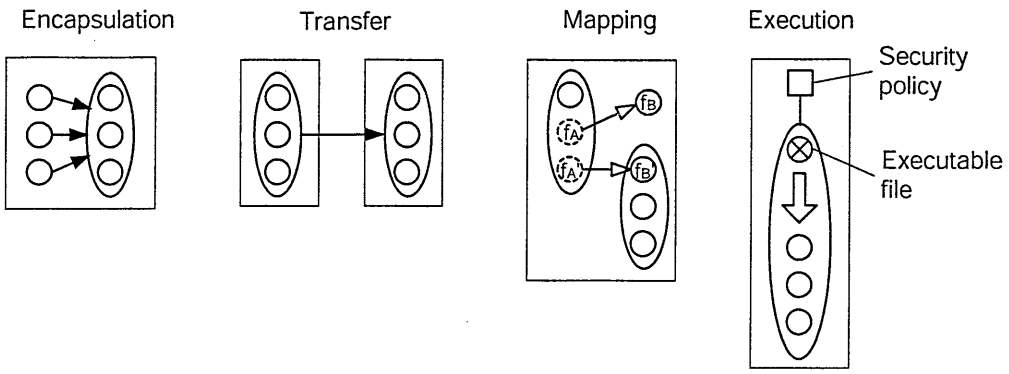


Figure 6: The four basic operations of the secure software circulation model.

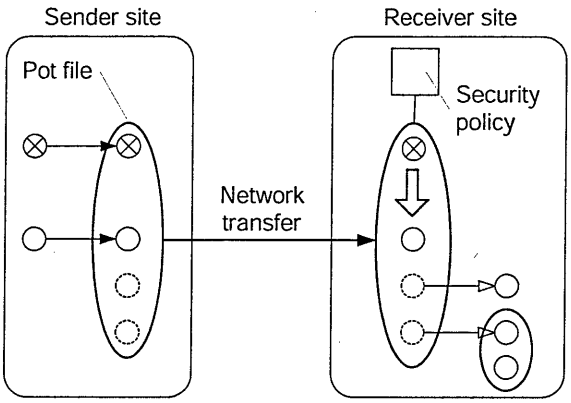


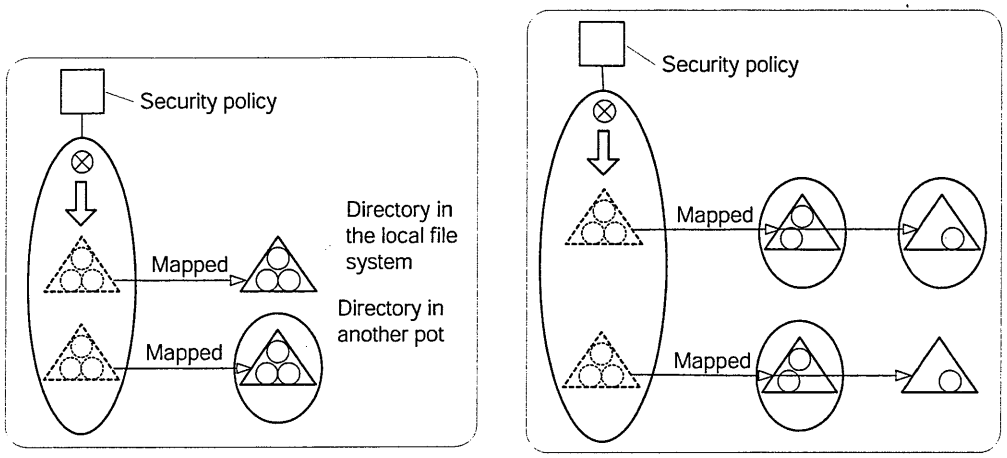
Figure 7: Typical use of the secure software circulation model.

- The mapping operation in the SSC model is a substitute for the extraction operation of the USC model. In Fig. 6, the mapping operations are represented by arrows with unshaded heads. The mapping from file f_A in a pot to file f_B in a pot or local file means that every access to f_A during execution within the pot is redirected to f_B.

- In the execution operation, a program is not executed in a virtual address space that shares a file system of the executing site. A pot is instead associated with a virtual process space just like its own file system, and a program in the pot is executed within that environment; no files other than those appearing in the pot can be seen. When execution is initiated, a security policy specification is associated with the execution. It specifies a mapping scheme and the resources allowed to be used.

As described above, a pot has two states depending on whether a process is associated with the pot. To distinguish between the two states, we use the terminology *pot-file* when a process is not associated with the pot, and *pot-process* when it is associated.

Figure 7 illustrates a typical use of the SSC model combining the four basic operations. In the sender site, one executable file (a circle containing a cross) and one data file (a solid circle with no cross) processed by the executable file are encapsulated. The remaining two files (broken circles) do not exist physically in the pot at the sender site and their file entities are mapped in the receiver site at the execution time. The archived file is transferred from the sender site to the receiver site via any transfer method such as FTP, WWW, or e-mail. At the receiver site, the user prepares



(a) Directory mapping (b) Cascaded mapping

Figure 8: Mapping directories in SoftwarePot.

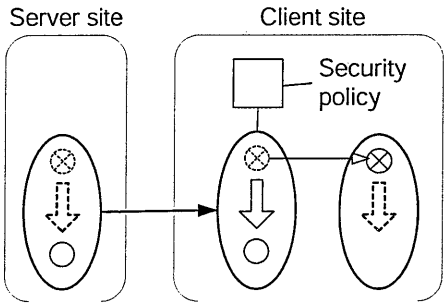


Figure 9: Secure interpretation of downloaded code.

a security policy description that specifies the mapping schemes between the file in the pot and a local file and between the file in the pot and a file in another pot. During the execution, no other files in the local file system of the receiver site can be seen. The security policy file can also specify system calls of the OS kernel allowed to be issued in the execution. Thus, the pot constitutes a sandbox distinct from the ordinary execution environment in user processes.

Last, we want to point out two slight extensions to the mapping operation. We permit the mapping of a directory in a pot to a directory in another pot or in a local ordinary file system as shown in Fig. 8(a). Furthermore, we permit cascading of the mapping as shown in Fig. 8(b). The informal operational semantics of the cascaded mapping is as follows. The cascaded mapping operation results in a totally ordered relationship between multiple directories in pots and/or an ordinary file system. When the pathname of a file is retrieved, the totally ordered directories are retrieved in the total order, and the first one found is the result. These extensions are useful in that they facilitate and simplify the description of security policies, and also make the computing of transferred pots more flexible.

2.3 Utilization of the SSC Model

Now we show how we can apply the SSC model to represent typical computation requiring secure software transfer in an open network environment.

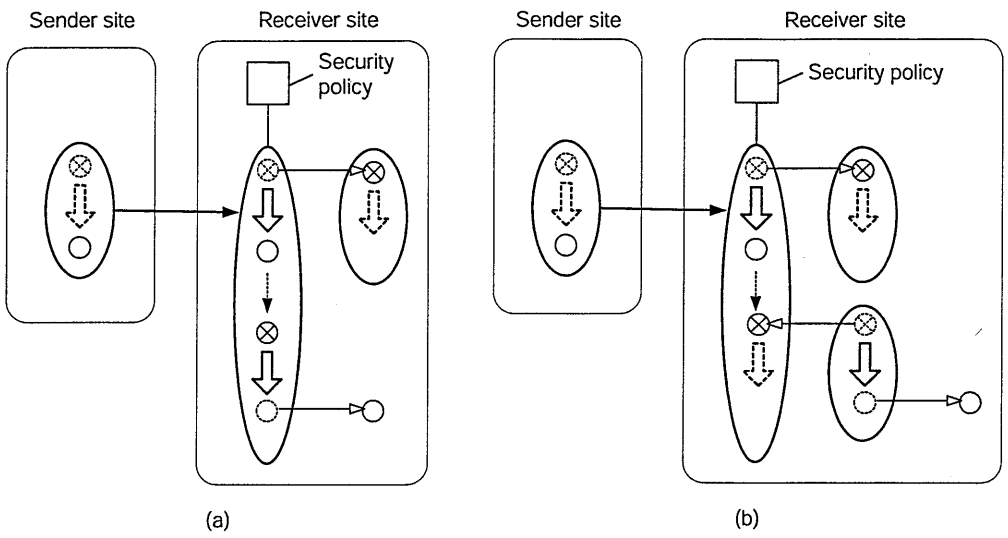


Figure 10: Secure execution of dynamically generated native code. The code generation and the execution of the generated code are done in a single pot in Fig. 10(a), and in different pots in Fig. 10(b)

Secure interpretation of downloaded code. A typical use of SSC is the secure interpretation of downloaded code. This form of computation has been made popular by the widespread use of Java and its applet system. SSC can be used to provide a general framework that enables secure interpretation of downloaded code (Fig. 9). The server site stores a pot that includes code which will be interpreted by an interpreter stored at a client site. In the figure, the interpreter is also assumed to be stored in a pot. Through the mapping operation, the interpreter entity appears in the pot sent from the sender and interprets the code under the control of the associated security policy. Note that this framework for the secure interpretation of downloaded code does not limit which programming language system or virtual machine instruction set can be used.

Secure execution of dynamically generated native code. Next, we will describe a more advanced form of downloaded code execution (Fig. 10). In this, the code downloaded from server to client is compiled into native code for the client's CPU architecture then the generated native code is executed at the client site. This form of computation is usually called execution by *just-in-time compilation.* As shown in the figure, such computation is represented by simply combining the basic operations. Thus, we expect a system implementing the SSC model to be able to realize such relatively complicated computation in a systematic way.

Secure execution of mobile agents. Mobile agent systems represent one of the most sophisticated forms of distributed computation. Interestingly, this form of computation can be represented in a straightforward way, in principle, by using the SSC model (Fig. 11). At the receiver site, after receiving a pot, the program stored in the pot is executed within the pot environment under the control of the specified security policy. This figure illustrates only the principle of mobile agent computation. More complicated settings will be required in real situations. For instance, files of a local file system at the receiver site might be mapped to the pot, the executed code might be interpreted as shown in Fig. 9, or the code might be executed using a JIT-compiler as shown in

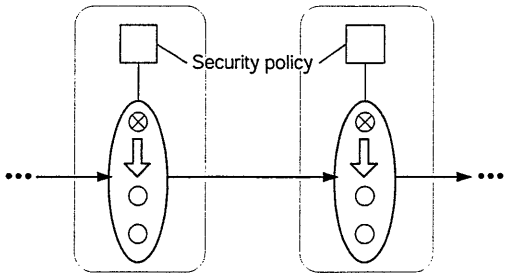
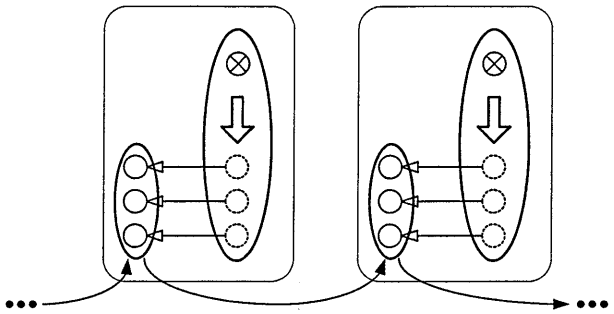
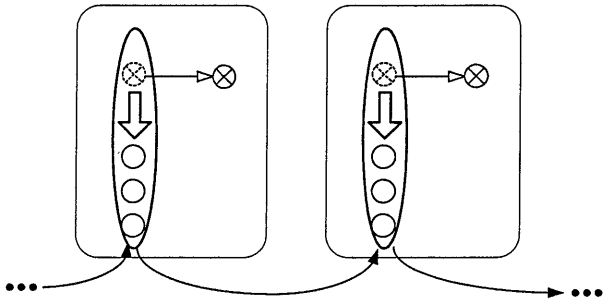


Figure 11: Secure execution of mobile agents.



(a) The executed programs are also in SoftwarePot



(b) The executed programs are in ordinary file system

Figure 12: Secure workflow computing.

Fig. 10.

Secure workflow computing. A representation for secure workflow computation can be obtained by slightly modifying that for the secure mobile agent computation. The difference is that the program code is included in the transferred pot in mobile agent computation, while each executed program code is usually locally prepared at each computing site in workflow computation. Typical settings are shown in Fig. 12.

3 SoftwarePot: An Implementation of the SSC model

The SSC model described in Section 2 can be implemented in several ways. This section describes a portable approach that does not modify the OS kernel. Instead, it uses functionalities extensively

```
static:
  /data/pic1.jpg  /home/user1/picture/picA.jpg
  /data/pic2.jpg  /home/user1/picture/picB.jpg
  /mybin/viewer   /usr/local/bin/viewer
  /mybin/runner   /home/user1/bin/runner
dynamic:
  /mybin/viewer_plugin1  http://www.foo.com/viewer_plugin1
  /mybin/viewer_plugin2  /home/user1/bin/viewer_plugin2
required:
  /var
  /extern_world
saved:
  /log
entry:
  /mybin/runner
```

Figure 13: Example specification for encapsulation.

to intercept and manipulate issued system calls. Such functionalities are provided in many modern operating systems such as Solaris, Linux, and FreeBSD. The system designed based on this approach is called *SoftwarePot*. It supports the four basic operations of the SSC model described in Section 2.2. One of these operations, the transfer operation, can be implemented by any common network transfer method, such as FTP, Web, or e-mail. In the rest of this section, we will explain our implementation scheme for the three remaining operations.

3.1 Encapsulation Operation

The encapsulation operation is implemented by collecting files specified as encapsulated and storing these files in a pot-file. Figure 13 shows an example of encapsulation specification. The lines preceded by the keyword `static:` specify that the file of the local file system specified in the second column is stored in the pot-file with the pathname specified in the first column. For example, the second line in the figure specifies that the local file `/home/user1/picture/picA.jpg` is stored in the pot-file with the pathname `/data/pic1.jpg`.

In addition to the basic function to store a file in a pot-file statically, the SoftwarePot system has a function used to store only the original location of the files to be stored, and the contents of the files are dynamically transferred when required (Fig. 14). This is specified in the `dynamic:` section in an encapsulation specification. In the example in Fig. 13, the file of pathname `/mybin/viewer_plugin1` should be obtained from the location specified by the URL `http://www.foo.com/viewer_plugin1`, and the file `/mybin/viewer_plugin2` should be obtained from the location `/home/user1/bin/viewer_plugin2` at the site where the pot-file is created. The static method is useful for storing absolutely necessary files, while the dynamic method is useful for storing optionally necessary files and for reducing the size of a pot-file.

The section preceded by `required:` specifies which files or directories must be mapped at the execution time. In Fig. 13, the example specification specifies that directories `/var` and `/extern_world` must be mapped at the execution time. The `saved:` section specifies that the

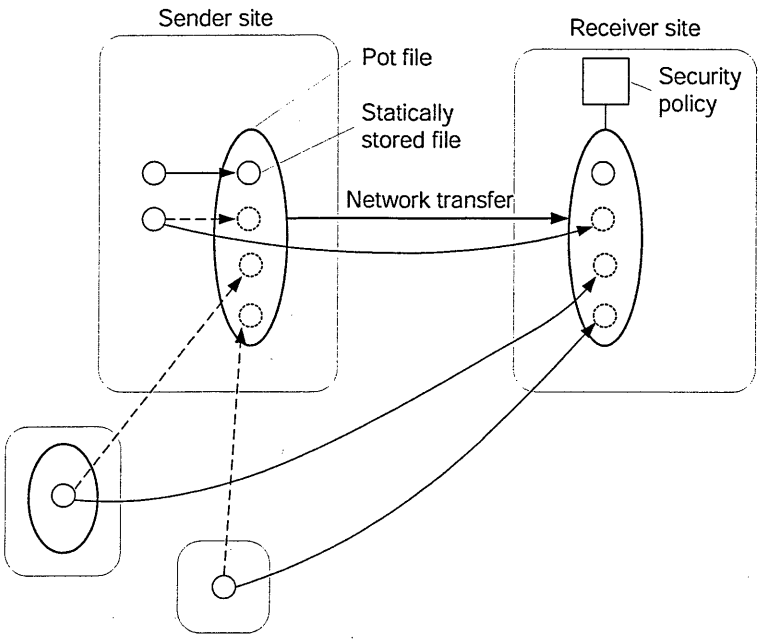


Figure 14: Dynamic file transfer.

modification of files in the listed directories should be permanently reflected in the pot-file system; the modification of other files is only temporarily reflected within the execution session and is thrown away after the session. The `entry:` section specifies the default program file that is first to be executed.

Figure 15 shows the internal structure of a pot-file in the current implementation. The entities of the files are stored in the area named *file store*. We adopted a modular structure to store files in the file store; currently the *tar*, *tar+gzip*, or *zip* formats can be used. The selection of storing format is completely transparent to the programs executed in a pot-process owing to the mechanism described in the next subsection.

3.2 Mapping and Execution Operations

The implementation of the mapping and execution operations is integrated in the SoftwarePot system, thus we will describe them together.

The essence of both operations is name translation; that is, every primitive operation to manipulate files (such as open, read, write, lseek, close system-calls) is redirected to the mapped destination file. Figure 16 illustrates this name translation scheme. All the file accesses in the pot—whether to mapped files or to unmapped files—can be treated uniformly through the mapping since we extract the files stored statically in a pot-file into the local file system by the time they are accessed. For the extraction, we provide two modes: the *eager* one and the *lazy* one. In the eager extraction mode, all the files statically stored in a pot-file are extracted at once when the execution operation is initiated. In the lazy extraction mode, on the other hand, each file is extracted only when it needs to be accessed.

In the middle of the initial setting of a pot-process, the name translation scheme is prepared according to a *mapping and security policy description*. The file-system space viewed from a pot-

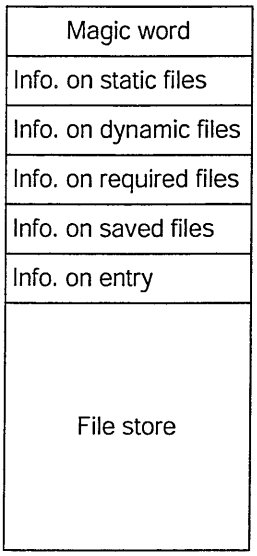


Figure 15: File format.

process is a virtual one created by the name translation specification, and a pot-process can never directly see the "real" file system of the executing operating system. We will explain how we can describe a specification for name mapping and security policy in SoftwarePot using the example shown in Fig. 17.

- The `map:` section specifies that each file or directory in the pot-file specified in the first column is mapped to the existing files or directories specified in the second column. In the example, the line `/etc/termcap /lib/tools.pot:/etc/termcap` specifies that the `/etc/termcap` file accessed in the pot-process is redirected to the file having the same name in the pot-file named `/lib/tools.pot`. The line `/alpha /beta,/gamma,/delta` shows a way to specify cascaded mapping. For instance, accesses to the file `/alpha/x` will be redirected to either file `/beta/x`, `/gamma/x` or `/delta/x`, and in this file selection `/beta/x` has the highest priority and `/delta/x` has the lowest.

- The line `indirect:` specifies a special mapping function called "process mapping"; that is, file I/O in a pot-process is redirected to interprocess communication. In the example, `/etc/passwd /home/user2/bin/passwd_filter` specifies that accesses to the file `/etc/passwd` in the pot-file is redirected to interprocess communication with the process initiated by the command `/home/user2/bin/passwd_filter`.

The remaining sections are used for specifying the security policy.

- In the `network:` section, the example specifies that all communication-related system calls are prohibited, with the exception that connection to TCP-port 80 of IP-address `130.158.80.*`, connection to TCP-port 21 of `130.158.85.97`, and bindings to TCP-port 22 of `*.tsukuba.ac.jp` are allowed.

- The `syscall:` section specifies the permission given for system calls other than network-related ones. In the example, all system calls other than network-related ones and those listed in the last two lines (preceded by the keyword `deny`) are allowed.

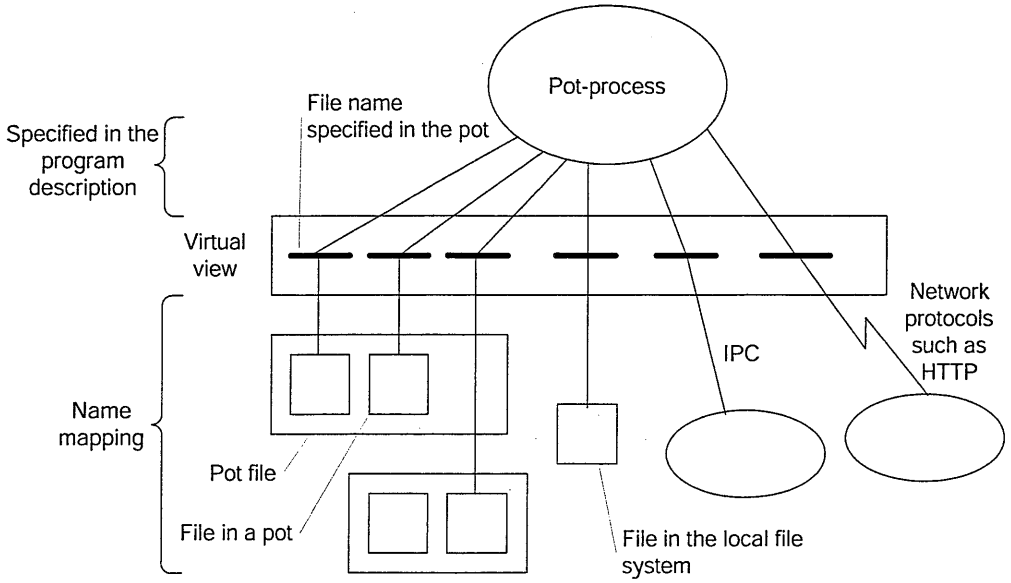


Figure 16: Implementation scheme.

Now we will explain how we implement the name translation. The trick is to rewrite the argument of the open system call as:

```
open("/foo.txt", ...)
        ↓
open("/tmp/XXX/foo.txt", ...)
```

at the execution time. Most modern operating systems support functions to intercept system calls just before and after the execution of the calls in the OS kernel, and for user-level code to be executed upon each interception. Also, modern OSs allow a user process to examine and modify the contents of another user process in a controlled manner. By combining these functions, we implemented the required name mapping mechanism transparently[1] without modifying the OS kernels. In the following explanation, each number in parentheses corresponds to that in Fig. 18. As the initial setting, a *monitor process* (monitor for short) is initiated and associated to a pot-process to monitor the execution. In the initial setting, the monitor process reads a mapping and security-policy description and records the information in its internal tables. The setting for name mapping is recorded in the *pathname translation table* in Fig. 18.

(1) When a pot-process issues an open system call, the kernel catches the call and calls up the monitor.

(2) The monitor looks into the contents of the pot-process and obtains the pathname of the file being opened. By referring to the pathname translation table, the monitor decides to which actual file the open system call should be applied. If the lazy file extraction mode is specified, the file is extracted from the file-store of the pot-file; if cascaded mapping is specified, file-retrieving is done according to the specified totally-ordered directory list; if

[1]The programs executed in a pot-process cannot detect such name translation and cannot modify or avoid the name translation.

```
map:
  /var            /tmp/var
  /extern_world   /home/user2/tmp
  /etc/termcap    /lib/tools.pot:/etc/termcap
  /alpha          /beta,/gamma,/delta
indirect:
  /etc/passwd     /home/user2/bin/passwd_filter
network:
  deny all
  allow connect tcp 130.158.80.*     80 # HTTP
  allow connect tcp 130.158.85.97    21 # FTP
  allow bind    tcp *.tsukuba.ac.jp  22 # ssh
syscall:
  allow all
  deny fork, fork1, vfork,
  deny unlink, rmdir, rename
```

Figure 17: Example specification for mapping and execution.

dynamic file transfer is specified in the mapping specification, the appropriate file is obtained via the network; if the `indirect:` option is specified, the specified process is initiated and interprocess communication between the pot-process and the initiated process is established.

(3) The monitor process writes the translated pathname to the unused area in the pot-process, and rewrites the register storing pointer to the pathname so as to reference the translated filename. Just before rewriting the register, the monitor process saves its content to be restored later.

(4) The monitor then tells the kernel to perform the actual processing of open.

(5) The kernel performs the processing for open.

(6) The kernel again calls up the monitor.

(7) The monitor restores the content of the register and erases the translated pathname written in (3). Finally, the monitor tells the kernel that the execution of the pot-process should be resumed.

During the execution, the pot-process may issue `fork` system calls and spawn other pot-processes. At that time, provided that a `fork` system call is allowed by the security policy description, the monitor will also fork and the child monitor process will monitor the child pot-process (Fig. 19).

All other system calls, apart from those explicitly specified as allowed in the security policy description, are intercepted by the monitor with the help of the kernel. Such calls are then examined and judged according to the security policy. When a violation is detected, the monitor takes a predetermined action, such as aborting the pot-process.

We have summarized the current implementation of SoftwarePot in Table 1.

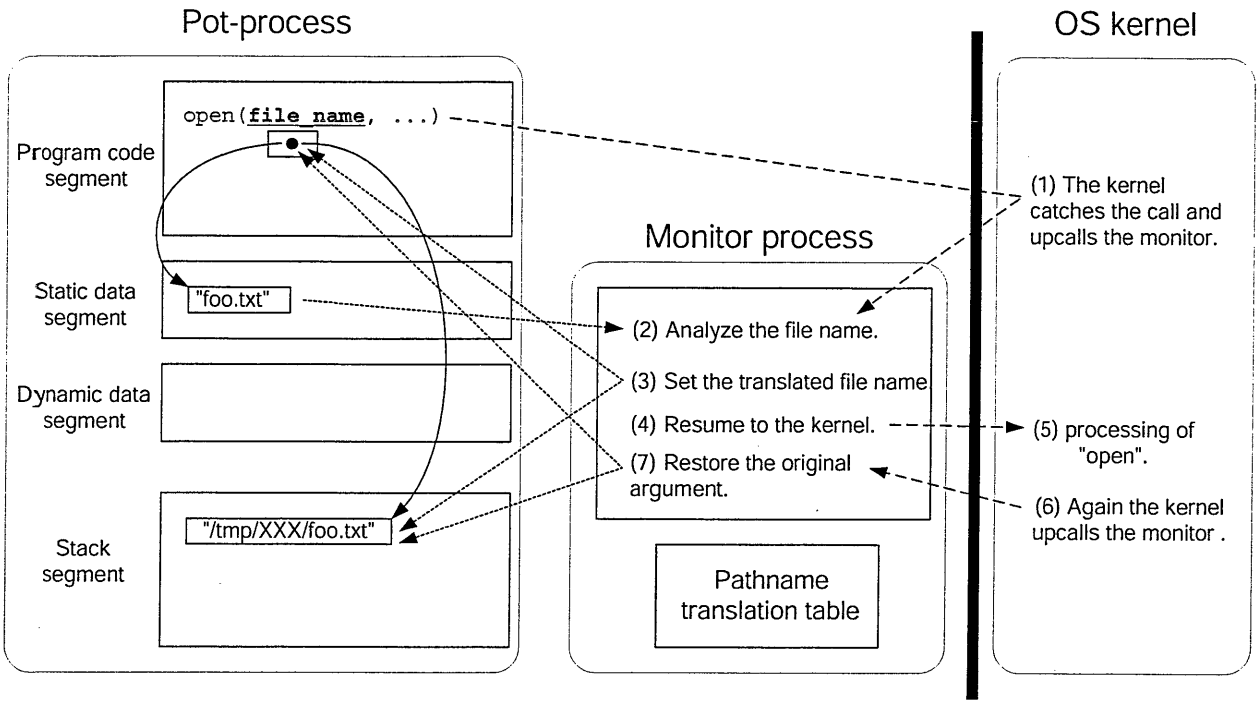


Figure 18: Implementation trick for name translation.

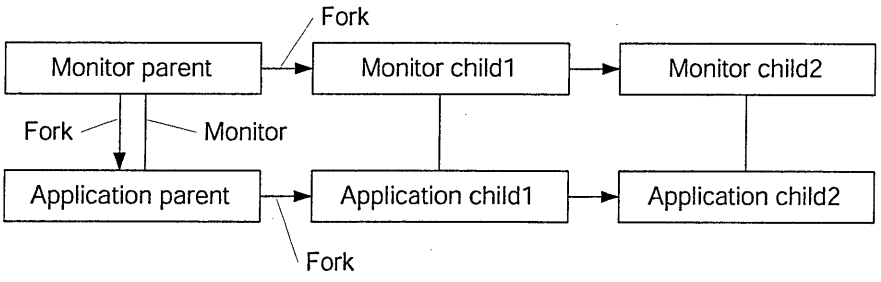


Figure 19: Implementation of fork.

4 Experiments

We implemented the SoftwarePot system on top of the Solaris 5.6 operating system based on the implementation scheme described in Section 3. About 5,000 lines of C program code were required to implement. This section describes the experimental results we obtained from the implementation. The hardware platform was a Sun Ultra 60 workstation (CPU: UltraSPARC II, 4MB cache memory, 512 MB main memory, 36GB hard disk).

4.1 Microbenchmark

4.1.1 Overhead imposed on system calls

We wrote three microbenchmark programs and identified the overhead imposed on system calls by comparing the execution time of these programs with and without SoftwarePot. The benchmark programs were as follows:

Table 1: Summary of the implementation scheme.

Operation	Implementation method	Current implementation
Encapsulation	Data archiving/storing format	Modular
Transfer	Data transfer method	Static (transferred before execution)
		Dynamic (transferred within execution time)
Map	Name mapping method	System-call interception
	File extraction method	Eager (extracted before execution)
		Lazy (extracted within execution)

Table 2: Execution times (msec) of the microbenchmark programs.

	Fib	getpid-caller	open-caller
Without SoftwarePot	13,415	10,620	192
With SoftwarePot	13,477	14,739	8,438

Fib This program calculates a Fibonacci number. It invokes no system calls while calculating.

getpid-caller This program calls the `getpid` system call repeatedly in its main loop (10 million times), but does not invoke any other system call.

open-caller This program repeats an operation that opens the same file and immediately closes it. It invokes pathname system calls many times: it executes 10,000 pairs of the `open` system call and the `close` system call.

We used the eager extraction mode in this experiment. The file extraction times in the above programs were extremely small and are not included in the results shown.

Experimental results are given in Table 2. The performance of Fib was little influenced by whether it ran with SoftwarePot. This was quite natural because the monitoring processes in SoftwarePot intercept nothing but system calls. Getpid-caller was slower when it ran with SoftwarePot. This was unexpected because `getpid` is not a pathname system call and so calls to it are never intercepted. We suppose that when some system calls are configured for interception, an overhead is added to all system calls. The benchmark open-caller ran an order of magnitude slower with SoftwarePot (the difference in overall execution time was 8,246 milliseconds). Since open-caller invoked `open` 10,000 times, the penalty paid for each `open` was about 0.8 milliseconds. Since the amount of computation SoftwarePot performs when intercepting one pathname system call does not depend on the kind (i.e., the number) of system call, the same penalty will be paid for all pathname system calls.

4.1.2 Overhead due to basic mechanisms

We measured the overhead of two basic mechanisms in SoftwarePot: the extraction of files lazily from archive files, and the acquisition of files dynamically through the network.

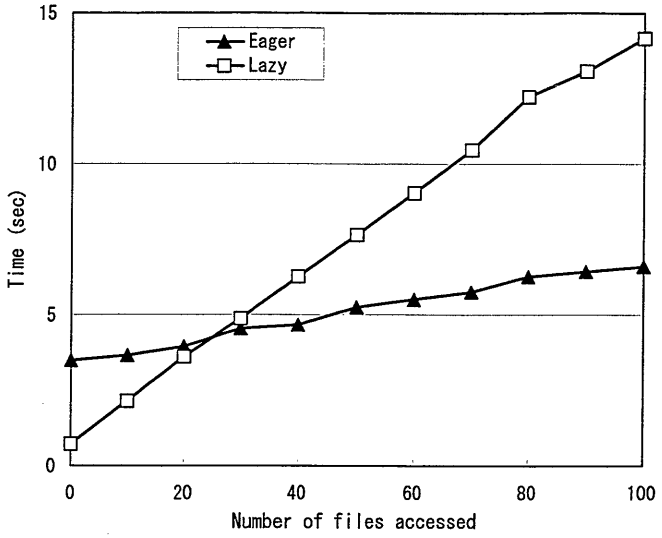


Figure 20: Execution times of the program written to compare the eager and the lazy extraction modes.

Eager extraction vs. lazy extraction. We compared the performance in the eager extraction mode with that in the lazy extraction mode using a pot-file containing 100 data files and a shell script file. The script opened some or all of the data files and concatenated their contents into a newly-created file. All the data files were 10 kilobytes long and their contents identical. We measured the execution times of the script while varying the number of concatenated files.

The results are shown in Figure 20 where the execution time includes the time needed for file extraction. The lazy mode was faster when fewer than 20 data files were accessed. When the script accessed more than 30 data files, the eager mode was faster. These results justify the use of SoftwarePot to support both extraction modes.

Performance impact of runtime file transfer. We also examined how much the SoftwarePot performance was affected by the number of files obtained dynamically through the network. In this experiment, we again used a pot-file that contained 100 data files and a shell script file. The script opened some or all of the data files and concatenated their contents into a newly-created file. All the data files had identical content and were 4,457 bytes long. Some of the data files were included in the pot-file, whereas the others were obtained at runtime through the network. In this experiment, SoftwarePot ran at the University of Tsukuba and obtained dynamic files from the University of Tokyo. We measured the execution times of the script while varying the ratio of files transferred at runtime. We used the eager file extraction mode throughout this experiment. The measured file extraction time was insignificant and is not included in the results shown.

Figure 21 shows our results. Read20files indicates that the script concatenated 20 files and Read100files indicates that the script concatenated 100 files. We found that the overhead of dynamic file transfer was significant (note that the concatenation of 20 dynamic files was more costly than the concatenation of 100 static files). Therefore, from the aspect of runtime performance, putting all necessary files into a pot-file and avoiding the obtaining of files from the network is generally reasonable. However, we should not forget that as more and more files are included in a pot-file,

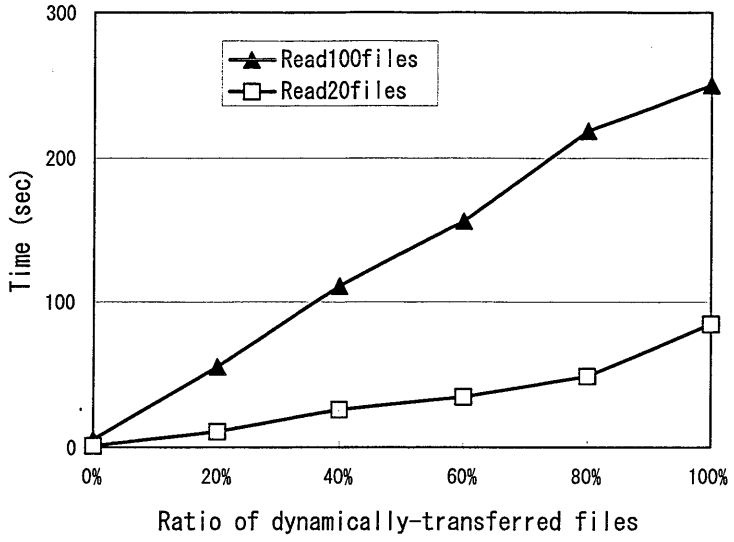


Figure 21: Execution times of the program for identifying the cost of dynamic file transfer.

the file becomes larger and takes longer to transfer. For example, the pot-file size used in this experiment was roughly proportional to the number of data files it contained. If we want to take the pot-file sizes into account, there's no straightforward way to decide which files should be put into a pot-file.

4.2 Macrobenchmark

We also executed larger applications to determine the overhead of SoftwarePot. These applications are described below:

Emacs Lisp We used a pot-file that contained a shell script and 216 Emacs Lisp files (.el files). The script invoked `emacs` in the batch mode. The invoked program then byte-compiled all the .el files into .elc files. After the compilation, the script copied the .elc files to the directory specified by the user.

LaTeX We used a pot-file that contained a shell script, a LaTeXsource file, three LaTeXstyle files, and 17 encapsulated PostScript files. The script first compiled the LaTeXsource file into a DVI file by invoking `latex`. Then it created a PostScript (PS) file from the DVI file by invoking `dvi2ps`. Finally, it copied the PS file to the directory specified by the user. The size of the PS file was 1.6 megabytes, which was equivalent to a PS file containing a 15-page paper written in English.

make & gcc We used a pot-file that contained a shell script and a source tree of the grep utility. The script first compiled source files by invoking `make` and `gcc`. Then the script verified the behavior of the newly-made grep binary with the test suite contained in the source tree. Finally, it installed the binary to the directory specified by the user.

catman We used a pot-file that contained a shell script and 189 nroff input files (which made up the

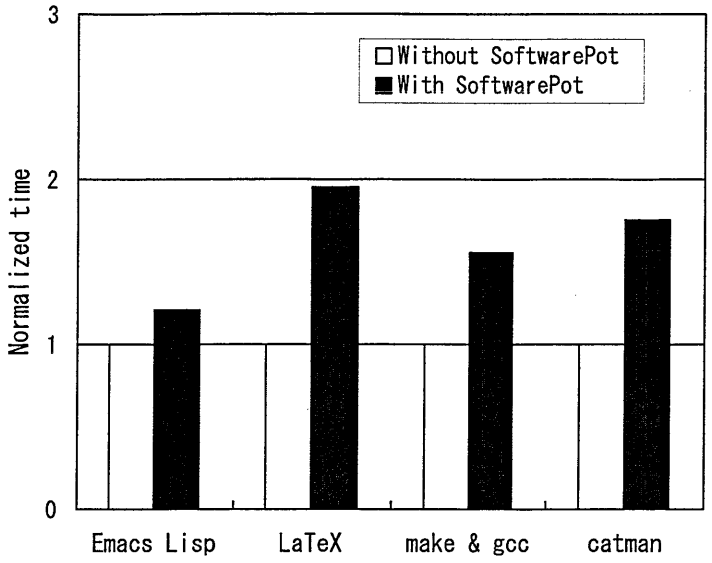


Figure 22: The macrobenchmark results.

second section of an online manual). The script invoked `catman` and created preformatted versions of the manual and the windex database from the input files. Then it copied the windex database to the directory specified by the user.

The pot-files used in the applications did not necessarily contain all the files required by the program binaries (e.g., dynamic libraries); the required files in a real file system are mapped to their counterparts in the virtual file system. We used the eager extraction mode throughout this experiment.

Figure 22 shows the normalized execution time of the applications with and without SoftwarePot, including the time needed for file extraction. The performance degradation of Emacs Lisp caused by SoftwarePot was fairly small (a 21% slowdown), probably because of the long computation time in the Emacs Lisp interpreter. The interpreter is so slow that overheads added to file accesses become proportionately less conspicuous. SoftwarePot will have little performance impact on this kind of CPU-intensive applications. On the other hand, the slowdown in the execution of LaTeX, make, and catman exceeded 50%. We think the overheads in these three applications, which are not negligible, were due to the frequent file accesses required to complete the applications. Since this preliminary implementation of SoftwarePot significantly increased overheads, we are now working on reducing these overheads by optimizing the SoftwarePot code.

4.3 Application Benchmark

We have implemented a tiny mobile agent system in Perl with SoftwarePot based on the scheme described in Section 2.2. Using this system, we have mobilized a Web search robot that collects HTML files based on the method described in Reference [14]. A mobile robot program moves from a base site to a robot server where it collects all HTML files placed in a Web server near the robot server. It obtains the root document file from the Web server then obtains other files from the Web server by recursively following hyperlinks in the files already obtained. After it collects all files

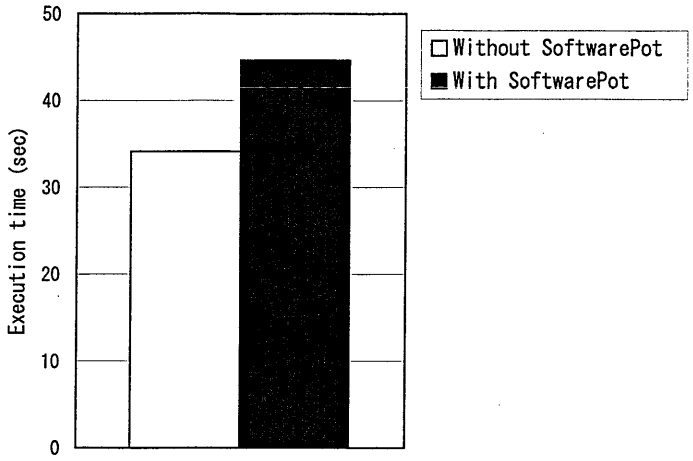


Figure 23: Result of the application benchmark.

obtainable from that Web server via hyperlinks, it packs the files into an archive file and moves itself to another robot server.

In this experiment, we executed the robot program in two ways: one that used SoftwarePot and another that did not. In the first, we created a pot-file that contained the program and an empty directory into which obtained documents were stored. The pot-file was sent to a robot server when the server invoked SoftwarePot and started the program contained in the pot-file. The program ran within the pot-process and stored obtained document files in a directory in itself. When it collected all document files in the Web server, it created a new pot-file that represented the updated file system, sent the pot-file to another robot server, and terminated the execution.

In the second, the robot program ran "directly" on the robot server and stored document files into a directory in the real file system of the robot server. When it had collected all documents from the Web server, it archived the documents and the program itself into a zip file. Then it sent the zip file to another robot server and terminated the execution.

We measured the elapsed time from when the program started to collect documents in a robot server to when it finished creating a pot or archive file that contained all obtainable documents. In this measurement, the robot server ran on the same machine as the Web server from which it obtained document files. The number of obtained documents was 236 and the size of the zip file was 492 kilobytes (Fig. 23). SoftwarePot added 31% overhead to the execution time. This overhead will become proportionately smaller, though, if the overhead due to network latency becomes larger (e.g., if the robot server accesses a Web server running on another machine).

5 Related Work

5.1 Chroot-based Systems

Many systems use the chroot facility to run untrusted programs in a confined file tree [13, 17, 19]. These systems put application programs and the files they will access (e.g., data files, shared

libraries, and configuration files below `/etc`) into a directory and then chroot it. The systems can thus prevent the application programs from accessing files outside the chrooted directory. Sbox [19] is a system for Web servers which confines CGI scripts to a chrooted directory. The File Monitoring and Access Control (FMAC) tool [17] provides extensive support for setting up a chrooted directory. Some Linux distributions use the chroot facility aggressively to enhance their security. For example, Kaladix Linux [13] confines nearly all daemons including Apache and BIND to a chrooted environment.

Those who use these systems must themselves set up a chrooted file tree. It is the users' responsibility to identify the set of files required to execute their software. SoftwarePot users do not have to do that because circulated pot-files contain their private file tree. Another advantage of SoftwarePot is its flexibility. Specifically, it can map "external" files to "internal" ones using the granularity of the files. The chroot-based systems either forbid access to external files, or else allow only mappings realized through special mechanisms such as NFS or samba. Note that the granularity of the mappings is a directory and that the control of NFS demands the root privilege.

5.2 Installation Tools

RPM [9] is a tool for managing software packages. Packages created by RPM (RPM packages) contain file archives and scripts needed for their installation. Unlike SoftwarePot, RPM pays little attention to security concerns: RPM executes installation scripts with no security check. Scripts in RPM packages run at the user's privilege and the user is sometimes `root`. Hence malicious RPM packages can seriously damage a system (e.g., can maliciously modify `~/.cshrc` or `/etc/passwd`).

Executable installers, including self-extracting archives, have the same problem. Most installers do not take security into account; those who execute installers are faced with a threat of executing malicious code (e.g., Trojan horses) with their privilege.

SoftwarePot addresses this problem by running programs in a confined file tree. Malicious code can read and update the files inside the tree, but it cannot access files residing outside.

5.3 Virtual OS

VMware [20] lets one operating system (a guest OS) run on top of another (the host OS). VMware is useful for executing untrusted software because software running in the guest OS cannot directly access files in the host OS. However, VMware is not intended for use in software circulation.

The world OS server [11] also allows users to create an execution environment that has its own virtual file system. This server helps execute untrusted software securely: damage caused by the software is confined to the virtual file system. The latest version of the server is based on system call interception and runs at the user level. Unlike SoftwarePot, though, the server provides no support for circulating software.

5.4 Secure Mobile Code

FlexxGuard [12] is also a framework for secure software circulation. SoftwarePot shares many ideas with FlexxGuard. The differences between SoftwarePot and FlexxGuard include their target languages and the mechanisms for security enforcement. FlexxGuard is targeted towards Java, and the custom class loader controls the protection domain of downloaded applets. SoftwarePot is targeted towards general software, and enforces security by virtually constructing a private file tree.

SKETHIC [6] is a proposed framework for distributing a combination of a program and its resource access list. SKETHIC, like SoftwarePot, transfers the burden of generating security policies from users to program developers. However, only the theoretical foundation of the SKETHIC framework has been reported and no details of its implementation have been given.

Developers and/or distributors of Java software can attach digital signatures to their code, and users can decide whether to execute any given Java code by examining the signature attached to it. A similar approach is used in Microsoft's ActiveX. A large problem with the signed code approach is that the user can only choose whether to trust the *principal* who wrote the code, not whether to trust the code itself. Hence, there are only two levels of trust: complete trust and complete distrust. Users have no information except the identity of the principal to use in deciding whether to trust any given code. This approach works only for the code written by reputable companies or organizations. On the other hand, SoftwarePot enables users to safely execute code written by unknown principals.

5.5 Sandboxing Tools

Janus [8], MAPbox [2], and SubDomain [7] are confinement systems that intercept system calls and kills an application if the system call invoked is determined to be malicious. Safe-Tcl [16] achieves a similar effect in the context of scripting languages by intercepting library commands. Users of these systems describe a policy file that dictates a set of allowed operations. These systems can be used for secure software circulation in a way very similar to that of SoftwarePot: software distributors bundle policy files with their software and the users execute the software on top of the systems to which they give the policy files.

Software Fault Isolation (SFI) [21] is a technique that modifies memory access instructions in a given program to prevent the processes executing the program from accessing addresses outside their address space. SoftwarePot realizes a similar effect in the context of files: an invocation of a pathname-related system call that tries to access a file outside the virtual file system is transformed into another invocation that tries to access a file inside the virtual file system. The relationship between SoftwarePot and SFI is similar to the relationship between chroot and process-based memory protection. SoftwarePot and chroot protect files, whereas the others protect memory. SoftwarePot and SFI are user-level services, whereas chroot and process-based memory protection are kernel-level ones.

5.6 File System Extension

The cascaded mapping mechanism of SoftwarePot relates to the viewpath mechanism supported in some distributed file systems such as the translucent file system [10] or the 3D file system [15]. Ufo [3] is a file system that allows users to treat remote files exactly as if they were local. Ufo is built on top of the system call interception facility provided by the `/proc` file system. Like SoftwarePot, Ufo works completely at the user level. The low-level mechanism used in Ufo is almost the same as the one used in SoftwarePot. While SoftwarePot focuses on confining untrusted applications and circulating software, Ufo focuses on providing a global file system.

6 Conclusion

We have introduced the concept of software circulation. With this concept, a category of useful software-transfer usage, including software distribution, workflow, and mobile agent approaches, was modeled by combining four basic operations. In an open network environment, such as the Internet, we cannot assume that every user or software application is trustworthy. To enable secure software circulation in such an environment, we reevaluated and redesigned the four basic operations. We designed the SecurePot system to support the redesigned four basic operations. The system supports an encapsulated, transferable file system. The system also supports additional advanced functions such as cascade directory mapping, temporary file mapping, file-process mapping, and dynamic file transfer. Our portable implementation of SoftwarePot did not modify existing operating system kernels. Experiments to evaluate the implemented system showed that the overhead depended on how many issued system calls were watched and intercepted during the runtime of the SoftwarePot system. If there were many intercepted system calls, the overhead became a significant problem. However, for the ordinary practical setting of realistic applications, we believe the overhead is reasonable and within an acceptable range for practical use.

Several interesting items for future work remain. First, we should develop implementation techniques that lower the runtime overhead. One obvious way is to implement the system at the level of OS kernels. Fortunately, most recent operating systems provide *kernel modules* that allow dynamic loading into OS kernels without requiring reconfiguration or recompilation of the kernels. Thus, we are now developing a Linux kernel module to implement the SoftwarePot functions. Second, we plan to use the SoftwarePot system to implement practical software tools that contribute to secure software circulation. We are developing a tool for software distribution that, like RedHat's RPM system [1, 4], does not require "installation" into the user's local file system in the conventional sense. We are also developing a mobile agent system that can use existing scripting language systems such as Perl, Python, or Lisp. Third, we hope to develop a theoretical foundation for secure software circulation and the SoftwarePot system. A possible starting point will be to apply and extend the framework of *mobile ambients* proposed by Cardelli [5].

Acknowledgements

We thank Hirotake Abe for his helpful discussions concerning the design of the SoftwarePot system. We also thank Hiroshi Ohkubo for developing a Perl-based mobile agent system with SoftwarePot.

References

[1] RPM. http://www.rpm.org/.

[2] Anurag Acharya and Mandar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *Proceedings of the 9th USENIX Security Symposium*, Denver, August 2000.

[3] Albert Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. UFO: A Personal Global File System Based on User-Level Extensions to the Operating System. *ACM Transactions on Computer Systems*, 16(3):207–233, August 1998.

[4] E. Bailey. Maximum RPM. Available from http://www.rpm.org/.

[5] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 204/1, Jul. 2000. Special issue on Coordination.

[6] Eun-Sun Cho, Sunho Hong, Sechang Oh, Hong-Jin Yeh, Manpyo Hong, Cheol-Won Lee, Hyundong Park, and Chun-Sik Park. SKETHIC: Secure Kernel Extension against Trojan Horses with Information-Carrying Codes. In *Proceedings of the 6th Australasian Conference on Information and Privacy (ACISP 2001)*, volume 2119 of *Lecture Notes in Computer Science*, pages 177–189, Sydney, July 2001. Springer.

[7] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious Server Security. In *Proceedings of the 14th Systems Administration Conference (LISA 2000)*, New Orleans, December 2000.

[8] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, July 1996.

[9] Red Hat. RPM. `http://www.rpm.org/`.

[10] D. Hendricks. A filesystem for software development. In *USENIX-Summer'90*, pages 333–340, Jun. 1990.

[11] Kouei Ishii, Yasushi Shinjo, Katsumi Nishio, Jun Sun, and Kozo Itano. The Implementation of an Operating System Based on the Parallel World Model by Tracing System Calls. In *Proceedings of the IEICE/IPSJ CPSY OS '99*, Okinawa, May 1999. In Japanese.

[12] Nayeem Islam, Rangachari Anand, Trent Jaeger, and Josyula R. Rao. A Flexible Security System for Using Internet Content. *IEEE Software*, 14(5):52–59, 1997.

[13] Kaladix linux. `http://www.kaladix.org/`.

[14] Kazuhiko Kato, Yuuichi Someya, Katsuya Matsubara, Kunihiko Toumura, and Hirotake Abe. An approach to mobile software robots for the WWW. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):526–548, July/August 1999.

[15] David G. Korn and Eduardo Krell. A New Dimension for the Unix File System. *Software – Practice and Experience*, 20(S1):19–34, 1990.

[16] Jacob Y. Levy, Laurent Demailly, John K. Ousterhout, and Brent B. Welch. The Safe-Tcl Security Model. In *Proceedings of the USENIX Annual Technical Conference (NO 98)*, New Orleans, June 1998.

[17] Vessilis Prevelakis and Diomidis Spinellis. Sandboxing Applications. In *Proceedings of 2001 USENIX Annual Technical Conference, FREENIX Track*, Boston, June 2001.

[18] William Stallings. *Cryptography and Network Security—Principles and Practice.* Prentice-Hall, 2nd edition, 1999.

[19] Lincoln D. Stein. SBOX: Put CGI Scripts in a Box. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, June 1999.

[20] VMware. VMware. `http://www.vmware.com/`.

[21] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP '93)*, pages 203–216, Asheville, December 1993.