Efficient User-Space Information Flow Control

Ben Niu
Department of Computer Science and Engineering
Lehigh University
19 Memorial Drive West
Bethlehem, PA, 18015
ben210@lehigh.edu

Gang Tan
Department of Computer Science and Engineering
Lehigh University
19 Memorial Drive West
Bethlehem, PA, 18015
gtan@cse.lehigh.edu

ABSTRACT

The model of Decentralized Information Flow Control (DIFC [17]) is effective at improving application security and can support rich confidentiality and integrity policies. We describe the design and implementation of duPro, an efficient user-space information flow control framework. duPro adopts Software-based Fault Isolation (SFI [22]) to isolate protection domains within the same process. It controls the end-to-end information flow at the granularity of SFI domains. Being a user-space framework, duPro does not require any OS changes. Since SFI is more lightweight than hardware-based isolation (e.g., OS processes), the inter-domain communication and scheduling in duPro are more efficient than process-level DIFC systems. Finally, duPro supports a novel checkpointing-restoration mechanism for efficiently reusing protection domains. Experiments demonstrate applications can be ported to duPro with negligible overhead, enhanced security, and with tight control over information flow.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*

General Terms

Security

Keywords

Secure information flow, software-based fault isolation, checkpointing

1. INTRODUCTION

Decentralized information flow control (DIFC [17]), which is evolved from classic information flow control models ([7, 9]), is an effective way of improving application security. In DIFC, components of an application are put into separate protection domains; information flow between domains is restricted by a trusted reference monitor (e.g., the OS) to enforce application-wide data confidentiality and integrity. For example, information in system password files should not flow to untrusted domains without explicit declassification. As another example, a web browser might forbid network data from contaminating user profiles.

There has been substantial research progress to apply DIFC at the OS-process level to improve application security, including Asbestos [10], HiStar [26], and Flume [14]. In this line of work, each protection domain is realized by an OS process and a reference monitor tracks inter-process information flow. Information flow between processes is forbidden by the reference monitor if that flow would violate the policy implied by the security labels of the sending and receiving processes. These systems' practicality in terms of security benefits has been demonstrated on real-world applications.

On the other hand, existing process-level DIFC systems have some drawbacks that hinder their adoption in practice. Since many process-level DIFC systems such as Asbestos and HiStar are implemented as new OSes, porting them to other OSes is nontrivial; they also do not benefit from OS kernel updates or new device driver support. To address the challenge of OS portability, Flume [14] is proposed as a user-space DIFC implementation (on Linux and OpenBSD). However, it still requires the root privilege to inject an OS kernel module that performs system-call interposition. Furthermore, performance overhead of process-level DIFC is significant when there is heavy inter-process communication (IPC) in an application. IPC involves heavy process-level context switches (including the cost of switching the page table). As a result, Flume imposes over 30% performance overhead on real applications.

We propose duPro, an efficient framework that allows applications to control information flow between components to enhance their security. One key starting point of duPro is that protection domains are put into the same process, but are isolated through software-based fault isolation (SFI [22]). Because of this design, duPro provides the following advantages over previous DIFC systems.

- duPro requires no OS changes. Separation between domains is achieved via a pure user-space implementation. Therefore, applications running in duPro can benefit from new OS kernel updates and driver support. In addition, duPro can be deployed as a user utility program; non-root users can use it without the root privilege.

- duPro is efficient. duPro domains reside in the same address space, enabling efficient context switches between domains. The state of an SFI-enforced domain is much smaller than a process state and there is no need to switch page tables when performing context switches. This makes SFI especially suitable for situations where domain communications are frequent. Moreover, duPro applies SFI to intercept system calls. It incurs less overhead compared to a kernel-level

ASIA CCS'13, May 8–10, 2013, Hangzhou, China.

system-call interposition mechanism such as Flume's kernel module or a `ptrace`-based solution.

A user-space DIFC framework is not without its challenges. One challenge we have encountered during experiments is to provide an efficient user-space *domain-reuse* mechanism. The need can be seen through the example of a web server. To isolate between client connections, the server could allocate a new domain to handle each connection. When connection requests are frequent, this would be inefficient in duPro because SFI domain creation is relatively expensive. A more efficient design is to reuse a domain for future connections after a previous connection has been handled. However, the challenge is that the domain has been tainted by data of previous connections and reusing it without caution would allow information flows between connections, violating the connection-isolation policy. To address the challenge, duPro implements a novel *checkpointing-restoration* mechanism based on signals. An untrusted domain can checkpoint its state before performing actions that can taint the domain. At a later point, the domain can be restored to an early state for reuse by rolling back all the changes to memory and DIFC labels between checkpointing and restoration. This mechanism provides a safe and efficient way of reusing a domain.

Another challenge is how to minimize changes to source code when porting applications to duPro. Similar to process-level DIFC systems, duPro requires changes to source code to adapt to its programming model. Since processes are not supported in duPro, duPro provides a set of APIs with similar semantics and interfaces as the process-related abstractions (e.g., pipes, socketpairs) in order to reduce the effort spent on porting applications to duPro. Our evaluation demonstrates that complex applications such as Apache can be ported to duPro without many changes to the source code.

At a high level, duPro's technical contributions are mostly two-fold. First, it provides an efficient, pure user-space DIFC enforcement tool at the granularity of SFI protection domains. This includes adapting an existing DIFC model (the Flume model) to a new context and building the infrastructure. Second, it provides an effective and efficient domain-reuse mechanism based on domain checkpointing-restoration.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the background of two pieces of work that duPro builds upon. Section 4 describes the design and implementation of duPro. Evaluation of duPro is presented in section 5. We conclude in section 6.

2. RELATED WORK

duPro adopts SFI to isolate protection domains. SFI is a special kind of Inlined Reference Monitors (IRM), which statically weave security checks into the application code to enforce a security policy. Specifically, an SFI security policy specifies valid code and data regions in which at runtime the program's control flow and memory accesses are restricted. According to the policy, the branch and memory access instructions are instrumented with checks to assure the destinations of such instructions fall within the valid regions. IRMs have had a long tradition in improving software security (e.g., [22, 21, 15, 12, 23, 5, 11, 6, 8]).

Mandatory access control (MAC) [20] enforces mandatory instead of discretionary data access by applications. Example MAC systems include SELinux [4], AppArmor [3] and `systrace` [19]. SELinux and AppArmor are Linux kernel extensions, which enforce MAC based on application-specific configuration files. In both SELinux and AppArmor, untrusted application components are forbidden to define or update the security policy. This design impacts flexibility and performance if there exist many such untrusted components. The system `systrace` intercepts Linux system calls and enforces MAC by checking the system calls of an application.

Classic information-flow-control models [7, 9] are special MAC models. In such models, the security levels (with a partial order) are predefined and managed by a centralized security officer. All the protection domains in the system are granted with certain security levels and the information is only allowed to flow from domains with low security levels to domains with higher levels. Example systems include TightLip [24], IX [16] and LOMAC [13]. These systems are also inflexible because the security levels are defined and maintained by a security administrator, who acts as the only data owner on the system and whose policy may be unsuitable for individual applications.

DIFC relaxes the restriction of centralized information flow control to allow an application to define its own custom policy. In such systems, data owners can create and maintain new security labels and grant capabilities to other principals discretionarily. Therefore, a data owner is capable of declassifying data. DIFC can be enforced at the language level or the protection-domain level. Example language-level DIFC enforcement tools (e.g., [17, 25, 27]) add DIFC related annotations in the source code and analyze the information flow mostly statically. Language-level DIFC enforcement provides flexibility, portability and fine granularity (at byte level). However, porting legacy applications to these systems is time consuming. Protection-domain level DIFC systems include Asbestos [10], HiStar [26], and Flume [14], all enforcing end-to-end information flow at the granularity of processes. As discussed before, duPro adopts DIFC on SFI-enforced protection domains, which provides better inter-domain communication performance and requires no OS changes.

Asbestos provides an abstraction called event processes, which has the same motivation as duPro's checkpointing-restoration mechanism: to reuse protection domains. In Asbestos, a base process can spawn multiple event processes that share most of the memory pages with the base process. As a result, an event process is lightweight. In this way, resources are reused across multiple event processes. To provide efficient sharing between an event process and the base process, Asbestos's OS kernel manipulates page tables. Being a user-space mechanism, duPro cannot manipulate page tables directly and instead relies on a mechanism that allows one domain to return to its previous memory and security state. We believe duPro's domain-reuse mechanism is conceptually simpler and can handle most of the practical situations when a domain needs to be reused.

3. BACKGROUND

duPro builds upon two pieces of previous work: (1) uPro, a user-space framework for isolating domains using SFI; and (2) Flume's DIFC model.

3.1 uPro

uPro [18] is a capability-based framework for enforcing privilege separation on applications. In uPro, applications are separated into modules that are loaded into separate protection domains. Each domain is given minimal privileges (e.g., allowed communications) specified by a configuration file. Communications between domains and system-call support are provided by uPro's runtime. We next discuss the major design choices in uPro.

First, it adopts SFI to isolate domains in a single process. The uPro toolchain compiles an application's source code and emits SFI-compliant object code. The toolchain inserts checks into the

object code to enforce SFI domains. Thanks to SFI, context switches between domains are more efficient than the case when domains are implemented as OS processes. Moreover, SFI provides an efficient mechanism for intercepting system calls. In particular, each SFI-enforced domain is provided with a set of safe exits, called *trampolines*, which are the only ways that code in a protection domain can escape the protection domain to uPro's runtime. Trampolines allow a domain to invoke OS system calls (after security checks succeed) or invoke uPro's API functions for inter-domain communication.

Second, uPro provides a configuration language that allows explicit modeling of an application's security architecture. An application's configuration file declares protection domains, privileges of domains (e.g., allowed inter-domain calls), execution units, the initial configuration, and other information.

Third, uPro separates protection domains from execution units. A protection domain is a *protection unit*, whose functionality is isolation. In contrast, an *execution unit* is for control flow and is realized by an OS thread. An OS process combines protection and execution in one abstraction. In uPro, multiple execution units can execute in one protection domain and one execution unit can cross multiple protection domains during its lifetime.

Finally, two kinds of communications are provided: cross-domain calls (CDCs) and communicators. CDCs have similar semantics to remote procedure calls and allow one domain to call functions provided by another domain. Allowed CDCs are statically declared in the configuration file of an application. CDCs involve lightweight context switches between domains within the same process; their performance is much better than inter-process communication mechanisms. Communicators are either unidirectional or bidirectional and have similar semantics to OS pipes and socketpairs. When a communicator is dynamically created, two handles representing the two ends are returned to the two domains involved in the communication; each domain has one handle. Handles are passed to uPro's `read` and `write` API functions to receive and send data.

3.2 Flume's DIFC Model

A DIFC model assigns security labels to protection domains and includes rules for allowing or forbidding inter-domain information flow and rules for updating security labels when flow is allowed. Many DIFC models have been formulated in the literature and they are equivalent in terms of supported security policies. duPro adopts Flume's DIFC model for its succinct and intuitive formulation. We next give a brief introduction to the model and leave example uses of the model to the Flume paper [14].

Each domain is associated with a *security label*: (S, I, Cap), in which S is the *secrecy label*, I the *integrity label* and Cap the *capability set*. A secrecy/integrity label is a set of tags, which are drawn from an opaque token set. A tag is associated with some category of secrecy or integrity. As one example, t_1 may be associated with data in a secret file and a domain having t_1 in its secrecy label implies that the domain contains data in the file. As another example, t_2 might label data that is endorsed by a trusted entity and a domain having t_2 in its integrity label implies that the domain contains only data endorsed by the trusted entity.

The model allows decentralized control of tags. Any domain can create new tags and can dynamically adjust its secrecy label S and integrity label I if it has a corresponding capability. The capability set, Cap, of a domain is the set of tag operations that the domain can discretionarily perform to change S or I. If $t^+ \in Cap$, then the domain can dynamically add tag t to either S or I. If $t^- \in Cap$, the domain can dynamically drop t from S or I. Adding a tag t to S allows itself to receive secret t data and dropping t from S declassifies secret t data in the domain. In an opposite way, adding an integrity tag t to I endorses its data as having high-t-integrity and removing t from I allows it to receive low-t-integrity data.

As notational convenience, when a domain's capability set is empty, we omit Cap and write just (S, I) for its security label. We write C for the set of global capabilities owned by all domains. When C is clear from the context, we omit those capabilities from Cap when writing (S, I, Cap). For a domain d, we write D_d for the set $\{t \mid \{t^+, t^-\} \subseteq Cap_d\}$, where Cap_d is the domain's capability set. Those tags in D_d are owned by domain d.

DEFINITION 1 (SAFE MESSAGES). *Domain p is allowed to send to domain q messages iff the following two conditions are satisfied:*

- *Secrecy:* $S_p - D_p \subseteq S_q \cup D_q$
- *Integrity:* $I_q - D_q \subseteq I_p \cup D_p$

When D_p and D_q are empty sets, the rule says that information can flow from domains with less secret data and higher integrity to domains with more secret data and lower integrity. With non-empty D_p and D_q, the sending and receiving domains can first adjust their secrecy/integrity labels using capabilities in D_p and D_q, send the message, and adjust the labels back to their original values.

External objects such as files, network connections, or the user's terminal also have security labels. However, they always have empty capability sets and their labels are immutable once set. Before a domain communicates with an external object, an *endpoint* must be created; the endpoint serves as an intermediary between the domain and the external object. An endpoint e also has a security label. Information can flow between an endpoint and its associated external object if their labels obey rules in Definition 1. By default, an endpoint e has the same label as its creator domain p, but its label can be adjusted dynamically by p to a new security label (S_e, I_e, Cap_e) as long as one of the following conditions is met.

DEFINITION 2 (SAFE ENDPOINTS).

- *e is a safe readable endpoint iff* $(S_e - S_p) \cup (I_p - I_e) \subseteq D_p$
- *e is a safe writeable endpoint iff* $(S_p - S_e) \cup (I_e - I_p) \subseteq D_p$
- *e is a safe readable and writable endpoint iff the above two conditions are met.*

Two endpoints, e and f, may be two ends of a communication channel (e.g., the two ends of a pipe), and the information flow from e to f is allowed iff the following rule is obeyed.

DEFINITION 3 (SAFE ENDPOINT COMMUNICATION). *e is writeable, f is readable,* $S_e \subseteq S_f \wedge I_f \subseteq I_e$

4. DUPRO

As a first step, duPro combines Flume's DIFC model with uPro's user-space protection domains. Each domain is associated with a security label (S, I, Cap) and information flow between domains is regulated by the DIFC model. External objects such as files, network, and terminal are also labeled. Domains communicate with external objects through endpoints. Furthermore, the configuration language of uPro is extended to allow specification of initial security labels of domains and external objects. Labels of domains may change dynamically because of new data received, explicit declassification, or newly created categories of tags. The duPro runtime monitors inter-domain information flow and prevents forbidden information flow.

The combination of DIFC and SFI domains yields immediate advantages over uPro and Flume. One drawback of uPro is that privileges of domains are assigned statically and it does not support dynamic creation/revocation of privileges and transferring of

```
1:  Trusted_PD_Type CM {
2:    export = {next_submission};
3:  };
4:  PD_Type GM {
5:    import = CM:{next_submission};
6:  };
7:  Labels {
8:    GM = ({}, {}, {});
9:  };
```

Figure 1: Configuration of the duPro-protected grading application.

privileges from one domain to another; furthermore, there is no notion that treats data owned by a protection domain at different security levels. These features are useful in many practical situations, but the challenge is how to prevent privileges from being propagated arbitrarily. This question is elegantly addressed by the DIFC model. Compared to Flume, duPro maintains the strength of user-space protection domains: low context-switch overhead and requiring no OS changes.

Next we discuss a few unique design points in duPro when combining DIFC with user-space protection domains. We first introduce a running example, which will be used to illustrate concepts in duPro.

4.1 A Running Example

We consider a grading application that runs on a college server shared by users including professors and students. Each student submits his/her homework to the professor of a course. After the submission deadline, the professor launches a grading application to grade submissions. The grading application consists of two modules: the control module (CM) and the grading module (GM). The CM module launches GM and iterates a list of students. For each student, it sends the file name of the submission to GM, waits for the completion of grading before proceeding to the next student. GM receives the file name of a student submission, grades the submission, and stores the grading result in a file.

For this application, we assume CM is simple and trusted, while GM is untrusted and may contain bugs that are exploitable by malicious student submissions. We desire a security policy that the grading application should not leak any student's submission or grade to other students even if the GM module has been taken over by an attacker. When both CM and GM are put into one protection domain, the policy is not enforced as an attacker can hijack the application using a malicious submission and leak other students' submissions and files.

We show how this policy can be enforced within duPro. As discussed before, each duPro-protected application comes with a configuration file, which declares the number of domains, their communication channels, and initial security labels. Figure 1 presents (part of) the configuration of the duPro-protected grading application. CM runs in a trusted domain, declared in a `Trusted_PD_Type` block; GM is in an untrusted domain, declared in a `PD_Type` block. CM exports a CDC function `next_submission`, which is invoked by the GM domain. The `Labels` block declares initial labels for domains and external objects. The GM domain has an initial label $(\{\},\{\},\{\})$.

Being a trusted domain, the CM domain is blessed with the ability to dynamically adjust security labels of other domains and external objects. A duPro application often needs a small trusted domain for controlling the setting of security labels. Figure 2 visualizes the grading application and its label settings at the moment when the application is grading the submission of student s_1. For simplicity,

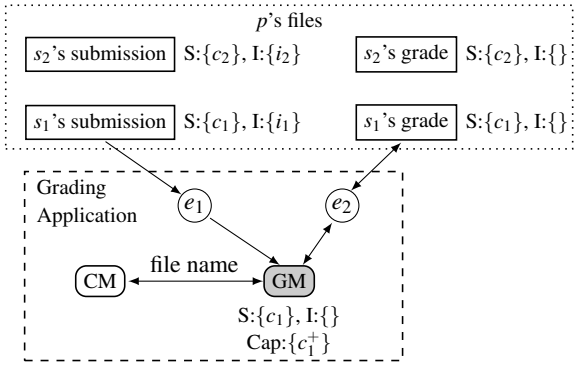

Figure 2: duPro-protected grading application.

only two students are included in the figure: student s_1 and s_2. In this graph and others, we use circles for endpoints, square boxes for external objects such as files, and rounded boxes for domains. An endpoint has the same security label as the domain it is attached to when there is no label for the endpoint in a graph.

The CM domain assigns a submission-specific label to each submission. The submission of student s_1 has label $(\{c_1\},\{i_1\})$ and the submission of student s_2 has label $(\{c_2\},\{i_2\})$. Intuitively, tag c_k stands for s_k's secret submission and tag i_k stands for data endorsed by s_k. Before the CM domain sends the GM domain the submission file name of s_1, it creates one grade report file with label $(\{c_1\},\{\})$ and grants capability c_1^+ to the GM domain. When grading the submission of s_1, the GM domain adds c_1 to its secrecy label so that it can read the submission file; it then performs the grading and outputs the grade to the grade report. When grading s_1's submission, the security label of the GM domain prevents it from touching s_2's submission. Therefore, even if the GM domain is hijacked, the attacker cannot read or write other students' files.

There is one issue that the above setup has not addressed. After grading s_1's submission, the GM module is supposed to grade s_2's submission. However, the GM module has already been tainted with data from s_1. Simply giving the GM module the capability of c_2^+ would allow a malicious submission from s_2 to possibly learn information from the submission of s_1. One solution is to allocate a new GM domain from scratch for grading s_2's submission. This ensures separation between submissions, but performance would be a concern as multiple GM domains have to be launched when many students' submissions need to be graded. Another solution is to ask the CM domain to perform erasure of sensitive information from GM, downgrade GM's security label, and then grant capability c_2^+ to GM for grading s_2's submission. This would work and reuse the GM domain. However, its drawback is that what information to erase is highly application-specific and the CM domain may forget erase part of the sensitive information. duPro's solution is to introduce a checkpointing-restoration mechanism, which is a principled way of performing erasure and downgrading.

4.2 Checkpointing and Restoration

Two API functions are implemented in duPro: one for checkpointing and one for restoration.

(1) `ST_HANDLE checkpoint();`
(2) `void restore(ST_HANDLE s);`

The `checkpoint` function remembers the current state of the calling domain. The domain state includes the state of its memory pages, its security label, context, and its control data. It returns a

```
1:  init();
2:  ch = checkpoint();      // ({}, {}, {})
3:  f = next_submission();  // ({}, {}, {c_k^+})
4:  cap = get_ownership();
5:  change_label(Secrecy, cap.plus[0]);
6:  grading(f);             // ({c_k}, {}, {c_k^+})
7:  restore(ch);
```

Figure 3: Pseudocode of the reusable GM domain. For understanding, we add security labels as comments at certain program states.

handle, which is later used by the `restore` function to restore the domain to its last checkpointed state.

It is important to note that these two functions are not privileged and can be used even by an untrusted domain. The reason is that checkpointing not only remembers the memory pages, but also the security label of the domain. If the domain is in a *consistent state* during checkpointing, meaning its security label accurately reflects the taint levels of data in those memory pages, then it is safe to go back to this state after the domain has evolved into another consistent state. It is safe even if there are side effects such as writing to external files between checkpointing and restoration because the interaction between the domain and external files is already governed by the DIFC model.

With the checkpointing-restoration mechanism, the GM domain in the grading application can be reused safely. The pseudocode of the GM domain is presented in Figure 3.

After initialization (line 1), the code calls `checkpoint` (line 2) to take a snapshot of its state, including the security label, memory pages and others. Since no grading work has been performed at this point, it is in a clean state with label $(\{\},\{\},\{\})$. Then the GM domain calls `next_submission` (line 3) exported by CM to retrieve next student's submission file name from the CM domain. The CM domain also changes GM's capability set to add c_k^+, assuming c_k is next student's secrecy tag. After GM receives the file name, it queries its own capability set (line 4), adds c_k to its secrecy label (line 5), and then performs grading (line 6). After grading, it calls `restore` (line 7) to revert to the last checkpointed state. Consequently, the memory pages tainted when grading the student's submission are rolled back to the clean state. Note that the `restore` operation also restores the program counter to one that follows the `checkpoint` operation and therefore GM's control continues at line 3.

The attacker can hijack the GM domain when grading (line 6), but the GM domain can neither read nor write other students' files. The CM domain checks whether the security label of the GM domain is restored before granting next student's secrecy tag, and hence if the GM domain does not restore itself after grading, the CM domain detects this and rejects the capability granting for the next student.

We next discuss how checkpointing is implemented in duPro. One way is of course to back up the entire domain state, including all memory pages. However, it performs unnecessary copying when the domain modifies only a subset of the memory pages between checkpointing and restoration. duPro adopts lazy copying. When checkpointing a domain, all domain state (the security label, context, and control data) except for memory pages is copied. Memory pages of the domain are set to read only through page protection, but are not copied at the moment of checkpointing. The domain continues execution and a `SIGSEGV` signal is generated when it writes data to a memory page. The signal is handled by duPro's runtime to copy the memory page and set the page to be writable, and then the domain code resumes performing the write to the original page. Restoration is implemented by restoring those copied memory pages. Note that only one signal is generated for an entire page when the page is first written. Therefore, if the domain writes a small number of pages between checkpointing and restoration, only those written pages are copied and restored.

The checkpointing-restoration mechanism does not forbid concurrency in a domain. Note that only the control of the thread that calls the `restore` is reverted and therefore the thread synchronization in a domain is left to the application.

4.3 Application-Controlled Label Setting

Process-level DIFC systems such as Flume use system-wide persistent labels for external objects such as files. Objects have security labels shared by all applications. The advantage is that the security policy can be enforced even after the system is shut down or crashes and later recovered. This implies that (1) the system provides persistent labels so that it is fail-safe; (2) the system also provides a mechanism to help rebooted processes to regain their security labels.

On the other hand, system-wide persistent labels have disadvantages. First, persistent labels cost extra storage. Second, system-wide persistent labels are inflexible and complicates the label setting of applications. We believe each application should be responsible for how it sets labels of objects to best suit its security goal. A persistent label setting is unlikely to work for all applications.

Instead of persistent labels, duPro allows applications to set their own labels of domains and external objects such as files and the network. Those labels are stored in duPro's memory. They are temporal and only durable during the execution of an application. Two applications running concurrently have two instances of duPro and can therefore have different labels for the same object. As a result, one duPro instance may forbid information flow between two files while another may allow such flow. This design is justified by duPro's goal, which is to regulate information flow during the run of an application according to an application-controlled policy; it is not to regulate information flow of the whole system. Though duPro does not maintain persistent labels, an application can still achieve application-level persistent labels. Its trusted domain can store in-memory labels to non-volatile storage before the application exits, and restores the labels when the application reloads.

All objects, including domains, files and the network have a default label. By default, a file f has label $(\{f_S\},\{f_I\})$, where f_S and f_I are tags unique to the file. Similarly, a domain d has label $(\{d_S\},\{d_I\})$ as well as an empty capability set, where d_S and d_I are unique tags. The network and terminal are with the default label $(\{\},\{\})$. The default labels can be overridden in multiple ways. The configuration file can declare initial labels for domains and other objects. A trusted domain can also set labels of other domains.

4.4 duPro API

duPro provides API functions to domains to interact with other domains and the duPro runtime. Table 1 presents a subset of the API functions. There are two groups: OS system-call wrappers and duPro-specific API functions. For untrusted domains, some API functions such as those performing I/O are related to information flow and their invocation has to be *checked* according to the DIFC rules by the duPro runtime. Some APIs are *unchecked* because they are unrelated to information flow; for example, `gettimeofday` queries the current time and it is unchecked. Other APIs such as `fork` are *forbidden* since processes are not allowed. Instead, duPro provides `pd_dup`, which allows the creation of a child domain that duplicates the parent domain. It also provides

Category	API functions	
	System-call wrappers	duPro-specific
Unchecked	gettimeofday,dup,dup2, exit,fstat,ftruncate, fcntl,flock,fsync,getuid,setsockopt,lseek, bind,close,select, mmap,socket,...	eu_create,com_create, create_tag*,checkpoint,restore,...
Checked	open,read,write,recv, send,stat,accept,connect,readlink,access, link,mkdir,rmdir, getcwd,chdir,chmod,...	get_label*,com_open, get_ownership*, change_label*,pd_dup, reduce_ownership*, get_fd_label*, change_fd_label*, pd_create,...
Forbidden	fork,execve,...	trusted_change_label, setgid,seteuid,setuid, setpgid,setgroups,...

Table 1: duPro API functions. Those marked with * provide the same semantics as their Flume counterparts.

`pd_create` for creating a new protection domain and loading code from an object file; in essence, `pd_create` is similar to `fork` followed by `execve` except that an SFI domain is created instead of a process.

All API functions are available to a trusted domain and no check is performed. In addition, a trusted domain can change the security label of an object by invoking `trusted_change_label`. We explain part of duPro-specific API functions next.

- `int trusted_change_label(TYPE t, OBJECT_ID obj_id, {`(S,I,Cap)`})`
  Change the security label of an object of type `t`, identified by `obj_id`, with the specified security label. This API function can be used to change labels of protection domains, files, directories, and other objects.

- `PD_HANDLE pd_create(PD_Type pt, char* exec, char* argv[], char *env[], COM_HANDLE ch,` (S_c,I_c,Cap_c)`)`
  Create a domain c of type `pt` with the object code file `exec`, the initial arguments, the environment variables, the created communicator, and optionally the specified security label. Suppose the parent p's security label is (S_p,I_p,Cap_p), it is required that $S_c \subseteq S_p \cup D_p$, $I_c \subseteq I_p \cup D_p$ and $Cap_c \subseteq Cap_p$. If c's security label is not specified, the default one (defined either by the application or the duPro runtime) is granted.

- `PD_HANDLE pd_dup()`
  Create a child domain with the same type and state as the parent. The context, memory pages, security label and file descriptors are copied to the child. The program counter of both the parent and the child is set to the instruction right after this call.

- `COM_HANDLE com_create(FLAG f)`
  Create a communicator and return a handle to it. The argument `f` indicates whether the communicator is *unidirectional* or *bidirectional*.

- `int com_open(COM_HANDLE ch, END e)`
  Return the logical read or write end of a communicator, depending on the second argument `e`, which is either *read* or *write*. The first argument `ch` is a created communicator handle.

4.5 Covert Channel Control

Besides explicit information flow through communication channels, information can also implicitly flow between domains through covert channels. Complete elimination of covert channels is difficult (if not impossible) because any resource sharing between domains might form covert channels. duPro limits covert channels in similar ways as previous process-level DIFC systems. For completeness, we list how certain covert channels are controlled and also list those uncontrolled covert channels.

Inter-domain communication. In duPro, communication can be either synchronous or asynchronous. Cross-domain calls (CDC) provide synchronous communication while communicators provide asynchronous unidirectional or bidirectional communication. A CDC can go through only when the security labels of the caller and callee domains allow bidirectional information flow, even if the callee returns no result. It is because the callee can control the response time—a timing channel. When using communicators, the operation of writing data to a communicator returns immediately with a success return code. However, communicators are unreliable in that data may fail to reach the reader domain and there is no failure notification; a notification would allow the reader domain to transmit one bit to the writer domain. Finally, if domains involved in inter-domain communication change their security labels during runtime, the change may result in the shutdown of communication channels as the new labels may no longer allow communication according to the DIFC model.

File descriptors. duPro protection domains reside in the same process. All domains share a single process-level file descriptor table. Without control, file descriptors could be used as covert channels between domains. File descriptors are allocated sequentially in OSes such as Linux. In addition, Linux sets a small limit on the size of the file descriptor table (e.g., 1024) and as a result the table entries can be exhausted in a short period of time. Both the predictability of file descriptors and possible exhaustion of the file descriptor table can allow one domain to observe the behavior of other domains. To limit such covert channels, duPro adds a layer of indirection. It assigns to each protection domain a domain-specific, logical file descriptor table, which records the correspondence between logical file descriptors and real file descriptors. Logical file descriptors are provided to a domain as results of API functions such as `open`. The duPro runtime performs translation from logical to real file descriptors before invoking OS system calls that require file descriptors. Logical file descriptors of a domain are allocated sequentially but they are not observable to other domains.

Tags/Handles. Tags and handles are created for entities such as protection domains and communication channels. As the tag/handle space is shared by all domains, predicable allocation of tags and handles would allow one domain to observe the number and the rate of tag/handle creation by a different domain. In duPro, tags and handles are 64-bit integers and are generated by using a block cipher (AES specifically) to encrypt a counter.

Memory/CPU/Storage/Network. duPro's protection domains do not share memory and one domain cannot observe other domains' memory. However, one covert channel may arise when an attacker domain tries to exhaust memory of duPro's runtime. In this scenario, one domain might dynamically create many domains, which requires memory allocation from duPro's runtime. It can then use creation and destruction of protection domains to exhaust and re-

lease duPro runtime memory to transmit one bit at a time to another domain. This channel is uncontrolled in duPro.

Covert channels may also be possible because of shared resources such as CPU, storage, and network. duPro is a user-space implementation so that it has no control over CPU scheduling or cache usage, for which the OS kernel is responsible. In addition, the CPU cache is shared by all duPro domains and it can be used as timing channels. duPro also does not control covert timing channels that are possible because of the shared network and the shared storage (e.g., the disk). Since duPro intercepts system calls, it would be possible to impose quotas on network and storage usage of a domain to limit covert channels.

4.6 Implementation

duPro is implemented on top of uPro. Its current implementation targets 32-bit Linux and is implemented in C++. Security and integrity labels are represented by C++ STL sets. The capability set is represented by two sets of tags: the plus set contains the set of tags that can be mounted and the minus set contains those tags that can be dropped. Each domain's control structure has a field representing its security label. External objects such as a logical file descriptor are also associated with security labels when they are created.

The implementation has around 1,500 lines of code for the DIFC model and around 500 lines of code for implementing checkpointing and restoration. The total runtime of duPro is about 26,000 lines of C code, together with around 6,500 lines of wrappers of duPro trampolines, which facilitates porting application programs.

5. EVALUATION

In evaluating duPro we consider whether duPro improves application security and how much performance penalty is imposed on ported applications. For evaluation we ported wget and Apache to run them in duPro.

To evaluate performance, we first carried out a set of micro-benchmarks to test the latency for a domain to invoke OS system calls and inter-domain communication. We also measured the performance of the duPro-protected wget and Apache and compared with their native counterparts.

To evaluate security benefits, we searched known vulnerabilities for wget and Apache and analyzed whether attackers could still succeed in exploiting such vulnerabilities in duPro-protected wget and Apache. Successful exploits mean that given the same threat model and security policy an attacker can perform the same attacks and cause the same damage on duPro-protected applications as on native applications. If the same attacks fail on duPro-protected applications, the vulnerability is contained.

All evaluation was performed on a system running 32-bit Ubuntu 12.04 with the Linux kernel version 3.2.0, the Ext4 file system, an Intel Core i7 870 CPU, and 4GB of memory. When testing Apache's performance, a client machine was set up with 64-bit CentOS based on the Linux kernel version 2.6.32, an Intel Xeon X5550 CPU, and 12GB of memory. The average latency between the server and the client is 0.23 ms.

5.1 Micro-benchmarking

As discussed before, duPro runs application code inside SFI-enforced domains. Application code can invoke allowed OS system calls through trampolines. Trampolines responsible for system calls are mapped to system-call wrappers, which perform DIFC checks and invoke OS system calls if checks succeed. Both the trampoline calls and the wrapper functions incur overhead.

Operations	duPro multiplier	Flume multiplier
`open`		
`-create`	1.4	16
`-exists`	2.1	12.5
`-non-exists`	5.5	9.5
`close`	2.7	1.3
`stat`	2.6	13.7
`unlink`	1.2	7.2
`readlink`	2.4	33.0
`mkdir`	1.0	4.3
`rmdir`	1.1	7.7
`IDC latency`	1.2	8.2

Table 2: Normalized operation latency in uPro and Flume.

We used a set of micro-benchmarks to measure duPro system-call overhead. In the experiment, we also compared duPro's system-call overhead with Flume's system-call overhead. Table 2 presents the results. The column of duPro multiplier presents the latency of performing a system call in duPro, normalized by allowing code to invoke Linux system calls directly (that is, without going through SFI and DIFC checks, which is unsafe). The `open` system call was tested in three modes: in the `create` mode, a new file is created; in the `exists` mode, an existing file is opened; in the `non-exists` mode, it tries to open a non-existent file and fails. All the system calls were performed 10K times and the average is presented.

Also presented in the table is Flume's multipliers for those system calls, extracted from the Flume paper. In general, duPro imposes much less system-call overhead than Flume, due to the implementation differences. Flume's protection domains are based on processes. It injects a kernel module to intercept system calls and redirects certain system call requests to a reference-monitor process. Handling a system call in Flume involves multiple process-level context switches: for instance, from the user process to the reference-monitor (RM) process, from the RM process to its file-server process, from the file-server process back to the RM process, and from the RM process back to the user process. The procedure involves multiple heavyweight process context switches. By contrast, duPro runs each module in a user-level sandbox and only two lightweight domain-level switches are needed to invoke a system call. Therefore, duPro's support of protection domains is lightweight and its handling of system calls is more efficient.

We also measured the inter-domain communication latency using two unidirectional communicators by transferring one byte back and forth between two domains one million times. The average is then calculated and normalized by the case of sending one byte using pipes between processes. The result and its comparison with Flume is presented in the last row of the table.[1] duPro performs better than Flume because the Flume RM process mediates inter-process communication. In Flume, a sending process sends data to the RM process and the RM then sends the data to the receiving process. This involves two process context switches. Furthermore, the RM process needs to queue the data-sending requests, which further increases the latency.

[1] duPro communicators are implemented on top of OS pipes and socketpairs, which is why the communicator performance is slower than pipe performance. An alternative implementation, which implements communicators in the duPro runtime, would be more efficient than pipes as it would not go through the OS for communication. The uPro paper [18] shows that cross-domain calls, which is implemented by the uPro runtime without going through the OS, is an order of magnitude faster than remote procedure calls.

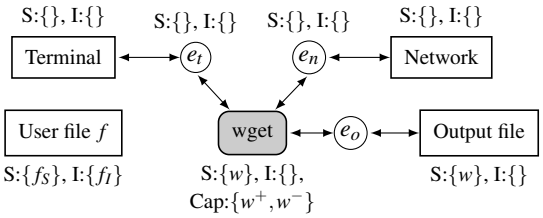

Figure 4: wget's label setting at runtime.

5.2 wget

wget is a command-line tool and commonly used to download files stored on a remote HTTP or FTP server. It parses the input URL of a remote file, connects to the server, creates a local file, retrieves the file contents, and writes the contents to the local file. We assume a threat model in which the remote server may hijack wget by sending malicious files. The desired security policy is that wget should neither taint existing user files nor upload any user data to the Internet, even if wget's implementation contains bugs. Obviously, the native wget fails to meet the security policy.

We ported wget version 1.14 to run in duPro and enforced the above security policy by restricting information flow. A process in the native wget was mapped to a duPro protection domain and an execution unit; this required an appropriate configuration file. wget's build scripts were also changed so that the default building toolchain was replaced by the duPro toolchain. We call this version duPro-wget.

No trusted domain is needed to enforce the desired security policy. Figure 4 shows the label setting of duPro-wget at runtime. The terminal, network, and other user files are associated with their default labels: the terminal and network are associated with label $(\{\},\{\})$, and each user file f is associated with $(\{f_S\},\{f_I\})$. In the configuration file, the initial label of the wget domain is $(\{\},\{\})$, and so is the directory `dir` that holds the output file. The execution steps of duPro-wget are as follows:

1. The wget domain is given the initial label ({}, {}). The domain then dynamically creates a tag w, which is associated with data downloaded from the remote server. Since wget owns the tag, duPro adds $\{w^+, w^-\}$ to wget's capability set. Then wget adds w to its secrecy label and creates an output file with the same security label as its label in directory `dir`.

2. wget connects to the network, retrieves the remote file, and writes to the output file. After finishing the writing, wget reports success to the terminal and exits.

During the execution, the wget domain cannot read or write other user files. A user file f has label $(\{f_S\},\{f_I\})$. Its secrecy label prevents wget from reading the user file and its integrity label prevents wget from writing to the user file. Therefore, even if wget is hijacked, the security policy is not violated.

To evaluate the security benefits of duPro-wget, we searched for past vulnerabilities of wget and investigated whether these vulnerabilities would be mitigated. We found 9 vulnerabilities [1] in Linux wget. Our analysis concluded that 6 out of 9 vulnerabilities would be contained. For example, vulnerability CVE-2002-1565 allows an attacker to execute arbitrary code and read/write arbitrary files with respect to wget. With duPro-wget, it would be contained and could not touch arbitrary files. The vulnerabilities that would not be contained are irrelevant to information flow control. For instance, vulnerability CVE-2006-6719 allows a remote attacker to perform denial-of-service attacks by sending a response message of large size to crash wget. Exploits of this vulnerability does not violate information flow control and are not contained.

File Size	50MB	100MB	150MB	200MB
Native wget	0.56s	1.13s	1.74s	2.29s
duPro-wget	0.58s	1.15s	1.77s	2.32s
Slowdown	4.4%	1.7%	1.5%	1.4%

Table 3: wget performance on files of different sizes.

To evaluate the performance overhead of duPro-wget, we compared the remote-file retrieval time between the native wget and duPro-wget. Results are shown in Table 3. Data files of various sizes were stored on a remote HTTP server. For each file, five tests were performed and the average time is presented. The results show that duPro imposes negligible overhead on wget.

5.3 Apache

Apache is an HTTP server running as a privileged program. On Linux, Apache can be configured with various Multi-Processing Modules (MPM) managed by a running process called Apache kernel at runtime. In the Prefork MPM, the Apache kernel forks a set of worker processes and each of them handles one connection at a time. When a worker handles a request, it logs the connection information (e.g., the IP address), sends web pages to the client, and processes the next one. If an attacker hijacks a worker, information of previous or subsequent connections handled by the same worker is leaked. The Prefork MPM can be configured so that a worker handles exactly one connection (called the Fork MPM). However, the Fork-MPM mode incurs frequent worker process creation and destruction, which impacts the performance.

We ported Apache (version 2.2.21) with the Prefork MPM to duPro. We call this version duPro-Apache. In duPro-Apache, both workers and the Apache kernel are untrusted. Instead, there is a trusted wrapper (around 90 lines), which is responsible for setting and adjusting labels of the workers and the kernel. In native Apache, a worker is created in the Apache kernel by invoking the system call `fork`. As `fork` is forbidden in duPro, it was replaced by the API function `pd_dup`. Besides, code for communication between a worker and the trusted wrapper via CDCs was added (sending log entries and retrieving labels for the next connection). In total, the porting required us to add around 140 lines of code and adjust the build scripts. The following security policy is enforced in duPro-Apache.

1. Client connections are isolated. This implies: (1) information should not flow between different workers; (2) if a worker handles multiple connections, the information of one connection should not flow to another connection.

2. Workers are only allowed to read web pages. Low-integrity data from the network should not flow to the web pages.

The policy is not enforced by the native Apache. duPro-Apache can enforce this policy with appropriate label settings. Figure 5 shows the label setting of duPro-Apache when handling requests. The initial labels of web page files, logs, Apache kernel and worker domains are specified in a configuration file. The trusted wrapper domain performs dynamic label changes. All the web pages have label $(\{p_S\},\{p_I\})$. The logs have label $(\{l_S\},\{l_I\})$. Each domain has label $(\{p_S\},\{\})$ when created. Each connection is modeled as an individual object. A secrecy tag is allocated for data received from a connection. For instance, connection C_x's secrecy tag is x and the connection's label is $(\{p_S,x\},\{\})$.

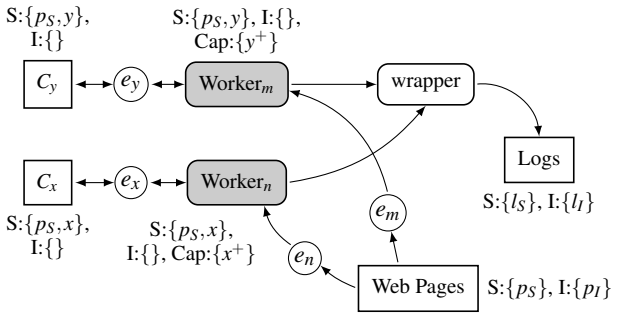

Figure 5: Label setting for Apache at runtime. The Apache kernel is not shown because, after worker domain creation, it enters into an idle state and does not handle connections.

```
1:  init();
2:  s = checkpoint();          // ({p_S}, {})
3:  next_connection();         // ({p_S, x}, {x+})
4:  d = accept();
5:  handle_conn(d);
6:  do_log();
7:  restore(s);
```

Figure 6: The pseudocode showing how an duPro-Apache worker domain handles connections.

In the Prefork MPM mode, each worker domain handles multiple connections. To ensure isolation between connections, duPro-Apache's worker code was modified to use duPro's checkpointing-restoration mechanism. Figure 6 shows the pseudocode for demonstrating how a duPro-Apache worker domain handles connections. The following steps happen during a worker's execution:

1. A worker domain is created with labels $(\{p_S\}, \{\})$. The `init` function (line 1) initializes the worker. The worker then calls `checkpoint` (line 2) to checkpoint itself.

2. The worker calls `next_connection` (line 3) to send a capability change request to the wrapper domain and the wrapper domain grants x^+ to the worker's capability set. Afterwards, the worker adds x to its secrecy label.

3. The worker accepts connection C_x by calling `accept` (line 4) and processes it by calling `handle_conn` (line 5).

4. The worker finishes processing and calls `restore` (line 7) to revert to the last checkpointed state; it then starts to handle the next connection.

In this setup, a worker cannot communicate information of its connections to another worker that has handled other connections. As Figure 5 shows, once two different workers have learned information from their connections, they are tainted with labels of different secrecy tags. This prevents communication between workers.

If two connections are consecutively processed by the same domain, shown in Figure 7, no information should flow between the two connections. In the figure, the worker handles C_{x_1} after handling C_x. The worker is supposed to call `restore` to restore its state so that C_x-related data are purged by the duPro's restoration mechanism and cannot be mixed with data from C_{x_1}. If the worker has been hijacked by C_x and it calls `next_connection` without first invoking `restore`, its secrecy label remains $\{p_S, x\}$. In this case, the trusted wrapper checks the worker's security label and refuses to grant the capability x_1^+ to the worker.

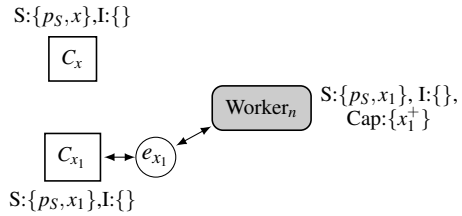

Figure 7: Label setting for an Apache domain handling consecutive connections. The worker can communicate with the current connection C_{x_1}. However, it cannot communicate with the previous connection C_x.

Native Apache's worker processes write logs to two files (for error logging and connection logging). The sharing of the log files forms covert channels between workers and between connections based on log-written speed and size. A possible approach to limiting such covert channels is that each worker domain writes a separate log file for each connection. However, this is impractical because a large number of small log files would be generated. This would waste the disk space and burden the file system. We moved the Apache logging code (only the part performing the I/O) to the wrapper domain (by adding around 60 lines) and exported CDCs for worker domains to invoke. The function `do_log` at line 6 sends a log entry to the wrapper domain, which truncates the log entry to at most 256 bytes and writes it to the log files. This limits the covert channels based on disk space exhaustion.

To evaluate duPro-Apache's security benefits, we inspected 36 publicly known Apache vulnerabilities posted on the Apache website [2]. All vulnerabilities are with respect to the Prefork-MPM Apache on Linux. After analyzing each vulnerability, we concluded duPro-Apache can contain the damages of 10 out of 36 vulnerabilities. Most of the vulnerabilities that cannot be contained are Denial-of-Service (DoS), which are out of the capability of duPro.

We measured and compared the performance of native Apache and duPro-Apache, in terms of the throughput and per-request latency. Native Apache was configured with the Prefork model and the Fork model. We configured all servers so that the number of workers at runtime was fixed. For duPro-Apache, the number of workers is the number of SFI domains created during initialization. For native-Apache-Prefork, the number of workers is the number of processes forked during initialization of Apache. for native-Apache-Fork, the number of workers refers to the maximum number of live worker processes at a given time. Since the Fork MPM creates one separate process to handle a connection, processes get created and destroyed dynamically.

Given a fixed number of workers, we tested the performance of the three versions by requesting a static 10KB page 10,000 times with different numbers of concurrent requests and took the geometric mean. Figure 8 shows the performance results of three Apache servers, including throughput and per request latency. The performance of duPro-Apache increases as the number of worker domains grow and maintains a relatively stable throughput. Its performance is comparable to native-Apache-Prefork. It even outperforms native-Apache-Prefork when the number of workers is large. The performance of native-Apache-Prefork has a slight decrease when the number of processes grow; we believe that is because of the process scheduling. Native-Apache-Fork has the same trend as duPro-Apache, but due to the expensive process creation, it performs much worse than the other two versions.

To evaluate the performance benefits of reusing workers in duPro-

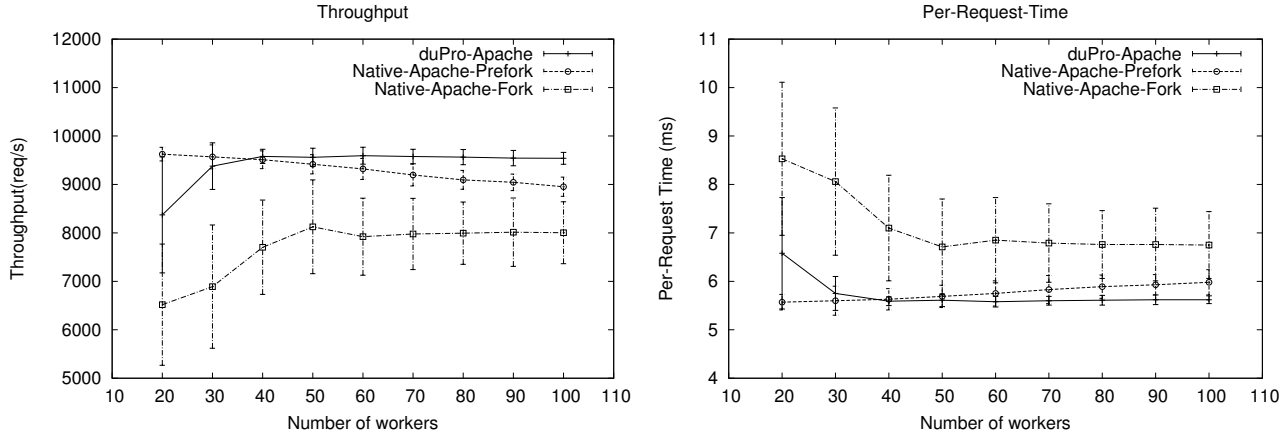

Figure 8: The performance comparison between duPro-Apache and native Apache. The left figure shows the throughput information and the right figure shows the per-request latency.

Apache, we also tested a version that recreates a worker SFI domain for each connection. We found the throughput of this version is three orders of magnitude worse than duPro-Apache because in the non-reuse version duPro spends almost all the time on creating and collecting protection domains. This demonstrates that reusing domains is critical to acceptable performance in duPro-Apache.

5.4 Summary of experiments

The experimental results of duPro are encouraging. The results demonstrate the efficiency of protecting user programs using duPro. In both applications, the performance of duPro-protected applications is close to its native counterpart, and with the enhanced security. Though duPro requires porting of application code, experiments demonstrate porting is straightforward to perform in practice.

6. CONCLUSIONS

duPro is an efficient user-level DIFC enforcement framework. duPro applies SFI to isolate protection domains in the user space. Each protection domain is associated with a DIFC label in order to track information flow. duPro neither requires OS changes nor root privileges when used, and thus benefits from the OS updates and new driver support. Thanks to SFI, duPro provides high performance IDC mechanisms. duPro also provides a checkpointing-restoration mechanism for efficiently reusing protection domains. The evaluation shows that the effort spent on porting applications to duPro is minimum and the performance overhead imposed by duPro on ported applications is negligible. At the same time, duPro-protected applications can use DIFC to provide end-to-end security guarantee on information flow.

7. ACKNOWLEDGMENTS

This research is supported by US NSF grants CCF-0915157, CCF-1149211, CCF-1217710, and two research awards from Google.

8. REFERENCES

[1] `http://web.nvd.nist.gov/view/vuln/search`. [Online; accessed on 21-October-2012].

[2] `http://httpd.apache.org/security/vulnerabilities_22.html`. [Online; accessed on 22-October-2012].

[3] AppArmor. `http://wiki.apparmor.net`.

[4] SELinux. `http://selinuxproject.org`.

[5] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-flow integrity. In *12th ACM Conference on Computer and Communications Security (CCS)* (2005), pp. 340–353.

[6] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with wit. In *IEEE Symposium on Security and Privacy (S&P)* (2008), pp. 263–277.

[7] BELL, D., AND LAPADULA, L. Secure computer system: Unified exposition and multics interpretation. Tech. Rep. ESD-TR-75-306, MITRE Corp., Mar. 1976.

[8] CASTRO, M., COSTA, M., MARTIN, J.-P., PEINADO, M., AKRITIDIS, P., DONNELLY, A., BARHAM, P., AND BLACK, R. Fast byte-granularity software fault isolation. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2009), pp. 45–58.

[9] DENNING, D. E. A lattice model of secure information flow. *Communications of The ACM 19* (1976), 236–243.

[10] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, M. F., AND MORRIS, R. Labels and event processes in the Asbestos operating system. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2005), pp. 17–30.

[11] ERLINGSSON, Ú., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 75–88.

[12] FORD, B., AND COX, R. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference* (2008), pp. 293–306.

[13] FRASER, T. LOMAC: Low water-mark integrity protection for COTS environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy* (2000), pp. 230–245.

[14] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard OS abstractions. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2007), pp. 321–334.

[15] McCamant, S., and Morrisett, G. Evaluating SFI for a CISC architecture. In *15th Usenix Security Symposium* (2006).

[16] Mcilroy, M. D., and Reeds, J. A. Multilevel security in the UNIX tradition. *Software Practice and Experience 22* (1992), 673–694.

[17] Myers, A., and Liskov, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering Methodology 9* (Oct. 2000), 410–442.

[18] Niu, B., and Tan, G. Enforcing user-space privilege separation with declarative architectures. In *Proceedings of the seventh ACM workshop on Scalable trusted computing* (New York, NY, USA, 2012), STC '12, ACM, pp. 9–20.

[19] Provos, N. Improving host security with system call policies. In *12th Usenix Security Symposium* (2003), pp. 257–272.

[20] Saltzer, J., and Schroeder, M. The protection of information in computer systems. *Proceedings of The IEEE 63*, 9 (Sept. 1975), 1278–1308.

[21] Small, C. A tool for constructing safe extensible C++ systems. In *COOTS'97: Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)* (1997), pp. 174–184.

[22] Wahbe, R., Lucco, S., Anderson, T., and Graham, S. Efficient software-based fault isolation. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (New York, 1993), ACM Press, pp. 203–216.

[23] Yee, B., Sehr, D., Dardyk, G., Chen, B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (S&P)* (May 2009).

[24] Yumerefendi, A. R., Mickle, B., and Cox, L. P. Tightlip: keeping applications from spilling the beans. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association, pp. 12–12.

[25] Zdancewic, S., Zheng, L., Nystrom, N., and Myers, A. Secure program partitioning. *ACM Transactions on Compututer Systems (TOCS) 20*, 3 (2002), 283–328.

[26] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making information flow explicit in HiStar. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 263–278.

[27] Zheng, L., Chong, S., Myers, A., and Zdancewic, S. Using replication and partitioning to build secure distributed systems. In *IEEE Symposium on Security and Privacy (S&P)* (2003), pp. 236–250.