# Enforcing User-Space Privilege Separation with Declarative Architectures

Ben Niu
Lehigh University
19 Memorial Drive West
Bethlehem, PA, USA
ben210@lehigh.edu

Gang Tan
Lehigh University
19 Memorial Drive West
Bethlehem, PA, USA
gtan@cse.lehigh.edu

## ABSTRACT

Applying privilege separation in software development is an effective strategy for limiting the damage of an attack on a software system. In this approach, a software system is separated into a set of communicating protection domains of least privilege. In a privilege-separated system, even if one protection domain is hijacked by an attacker, the rest of the system may still function.

uPro is a tool that provides efficient and flexible enforcement of privilege separation. It adopts software-based fault isolation to implement protection domains in the user-space so that inter-domain communication is efficient. It provides a declarative language to describe an application's security architecture, facilitating developers to identify different architecture alternatives. The evaluation shows that real applications can be ported to uPro with enhanced security, acceptable performance, and declarative architectures.

## Categories and Subject Descriptors

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

## General Terms

Security

## Keywords

Privilege separation, declarative security architecture, SFI

## 1. INTRODUCTION

Software security becomes increasingly important as the complexity and size of software systems grow. Applying the principle of least privilege [39] in software development is an effective way to improve software security. In this design, modules of a software application are put into separate protection domains so that the compromise of one domain does not directly lead to the compromise of other domains. Each domain is given only necessary privileges to complete its task. Privileges are added as needed and discarded after use. In this way, if one domain is hijacked, the attacker is restricted by the privileges of that specific domain.


*STC'12*, October 15, 2012, Raleigh, North Carolina, USA.


This software design principle, called *privilege separation* (or compartmentalization), has been adopted by many security-critical applications. As one example, Provos restructured OpenSSH into an architecture with one privileged server process and many unprivileged monitor processes for handling user connections [35]. Similarly, the Chromium browser puts the browser kernel into one process and separates each tab's complex rendering engine into a sandboxed process ([11, 36]).

Traditionally, a common approach to privilege separation is to use OS processes. However, as pointed out by Krohn *et al.* [25], the security abstractions of commodity OSes often get in the way of realizing privilege separation. A number of research systems aimed to address the insufficiency of commodity-OS isolation primitives by implementing new OSes or augmenting OS kernels ([16, 49, 26, 12, 45]). These systems' practicality has been demonstrated on real-world applications.

Despite the success of past research, applying privilege separation to modern software is still a daunting task. A couple of factors contribute to the difficulty.

- *Performance overhead*. Many privilege-separated systems map protection domains to OS processes. Since the performance overhead of inter-process communication (IPC) is significant, this approach is unsuitable for many situations. For example, software applications such as web browsers often allow third-party plug-ins to run in the same address space as the main application. Isolating these plug-ins in separate processes would be better for security, but the use of IPC for communication would be too costly.
- *Declarative security architectures*. An application's security architecture refers to how it is structured into protection domains and what privileges each domain owns (e.g., allowed communications with other domains). The security architecture is usually not static and often needs adjustments to balance between security and performance and to respond to new security threats. Unfortunately, there is no systematic tool that developers can use to describe an application's security architecture in a high-level, impeding their description.

In this paper, we describe the design and implementation of of a software tool titled uPro (for user-space Protection), which makes it easier for the adoption of privilege separation by addressing the above limitations. Specifically, uPro's design is based on two simple ideas. The first idea is to adopt a notion of *user-space protection domains* that are isolated in a single process via software-based fault isolation (SFI [43]). The second idea is to provide *a declarative language* for explicitly modeling an application's security architecture. Compared to previous tools for enforcing privilege separation, uPro has the following advantages:

- **Lightweight inter-domain communication.** uPro's user-space protection domains reside in one address space so that the context switches between domains do not need the whole address space to be switched. It has better cross-domain communication performance than IPC.
- **Flexible configuration.** uPro's Security Architecture Language (SAL) allows developers to describe their application's security architecture at a high-level via a declarative model. Such a declarative model helps declare the security architecture and identify it. uPro also provides a tool for automatically generating glue code between protection domains based on the declarative model, implementing policy-based inter-domain communication.
- **No OS-kernel change.** uPro has been implemented on top of Linux. The implementation does not insert an OS kernel module or modify the OS kernel code. It is completely implemented in the user space.

Before proceeding, we stress that uPro is about *enforcement* of privilege separation, rather than about *partitioning* applications into least-privileged modules. uPro assumes an application has already been partitioned into modules that are ready for being loaded into protection domains. Given modules and a security architecture, uPro enforces the architecture on the modules. How to split the application into modules is out of the scope of this paper.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the design and implementation of uPro. Evaluation of uPro is presented in section 4. We describe future work in section 5 and conclude in section 6.

## 2. RELATED WORK

uPro adopts SFI for isolation within the same process. Isolation can be achieved in many other ways. We already mentioned the OS-level approach that relies on process-level isolation. Example systems in this category include Asbestos [16], HiStar [49], Flume [26], Wedge [12], and Capsicum [45]. These systems' management of privileges is flexible, allowing dynamic creation and transfer of privileges, while uPro's privileges are statically assigned to modules. On the other hand, these systems modify the OS-kernel for enforcing process-level privilege separation. In contrast, uPro is a user-space framework and is portable across OSes. Isolation can also be achieved through *language-based isolation* (e.g., [31, 30, 29, 24, 32, 48, 50]), which is fine-grained, portable, and flexible. uPro's isolation through SFI works at the binary-code level and can sandbox components developed in different programming languages. Another way is through *virtual-machine isolation* (e.g., [15]). But this approach is heavyweight in terms of time, space, and communication costs. Finally, *hardware-level protection domains* within a single address space have also been explored ([46, 33]). This approach is efficient, but is incompatible with commodity hardware on which most user applications are running.

Software-based fault isolation (SFI) is a special form of inlined reference monitors (IRM), which performs static rewriting to weave security checks into the target application. IRMs have had a long tradition ([43, 42, 27, 19, 47, 41, 10, 18, 7, 17, 8, 13, 9]). Recent production-quality implementations of SFI from Google have made it efficient [47], compatible with multiple hardware architectures [41], and compatible with dynamic code generation [10]. Instead of static rewriting, program shepherding [23] (as well as systems in [40, 34]) relies on dynamic binary rewriting to enforce security. One downside of dynamic rewriting is that the whole dynamic optimization and monitoring framework is in the trusted computing base (TCB), whereas in static rewriting a separate verifier can be constructed to verify the result of static rewriting, thus removing the rewriter out of the TCB.

uPro's security architecture language, SAL, bears many similarities to software Architecture Description Languages (ADL) studied in the software-engineering community (e.g., [28, 21]). The main difference is that most ADLs are designed for concerns such as performance and conformance to standards, not system security. Besides describing the architecture of the application, SAL is capable of implementing Mandatory Access Control (MAC) since the declaration of the policy-based inter-module invocation checks is also embedded. Compared to other MAC implementations on Linux (e.g., SELinux [6] and AppArmor [3]), uPro runs completely at the user-space, sacrificing little performance while gaining potential OS portability.

XDR [4] and MiG [5] both define data types with annotated size information to facilitate the RPC client and server stub code generation. SAL also applies annotated data types when generating the cross-domain call (a kind of uPro inter-domain invocation, described fully in section 3.4) stub code.

## 3. UPRO

Before discussing uPro's design in detail, we give a quick overview of uPro and introduce a running example.

**Overview of uPro.** Figure 1 visualizes the phases and steps involved to apply uPro to an application with a security architecture. The first phase for a developer is to assemble a package from the application's source code. To assemble a package, the developer first decides a security policy and performs manual partitioning of the application so that the granularity of modules is sufficiently fine to enforce the policy. The effort for partitioning may be small or significant depending on the policy and the application. uPro treats each file[1] as a separate module. Therefore, the first step may involve splitting a file into multiple smaller files. Some additional code may also be necessary; for example, code for validating messages received from other modules may need to be added. After this step, the application becomes a collection of modules.

In the second step, the developer models the application's security architecture through uPro's SAL language. This step involves explicit declaration of protection domains, privileges of domains, execution units, the initial configuration, and other information. The result of this step is a configuration file, which is a formal documentation of the application's security architecture.

Next, the developer can use a uPro tool `cdcgen` to generate stub code for cross-domain calls. The stub code marshals function arguments sent by a sender domain and performs unmarshalling in the receiver domain.

The next step is to build object files from modules. Modules that are loaded into the same protection domain, together with the stub code generated by `cdcgen`, are compiled into one object file. uPro relies on Native Client's toolchain [47] for compiling source-code modules to object files. Minor changes to the application's build script may be necessary. The resulting object files are SFI-compliant, meaning their code obeys the SFI sandbox policy.

The above steps are about assembling an application package, consisting of a set of object files and a configuration file. Given a package, uPro's loader loads it into an address space. The loader performs initialization according to the configuration file, such as constructing the initial protection domains, loading object files into

[1]Multiple source files can be grouped into a single file in practice when demanding.

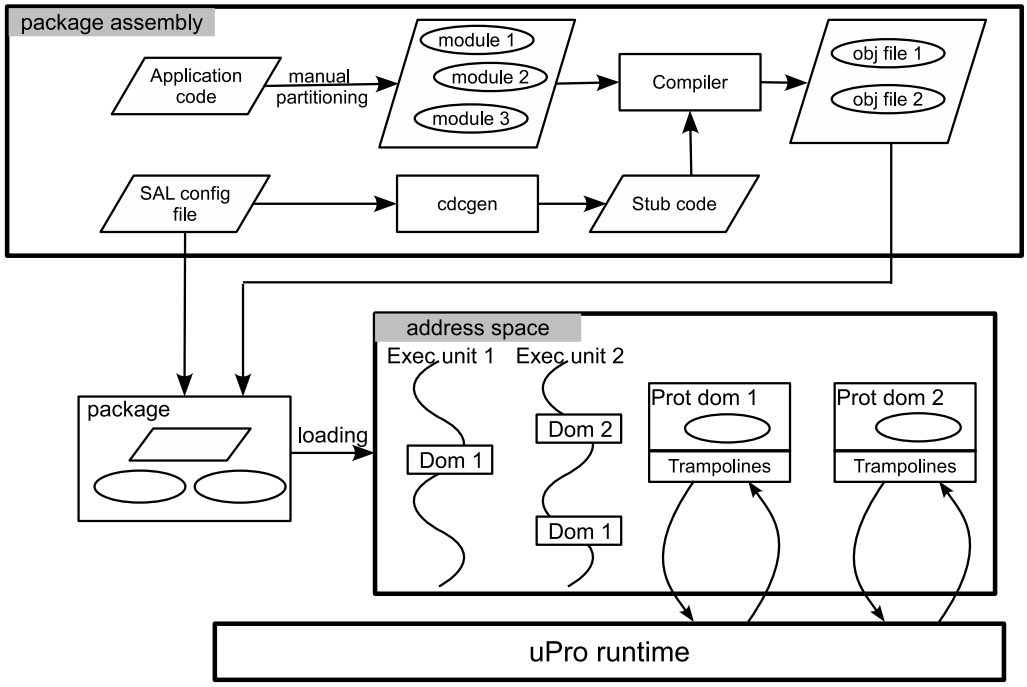


**Figure 1: An overview of uPro.**

the domains, setting up safe exits for domains, and constructing initial execution units. The safe exits, called *trampolines*, are the only ways that code in a protection domain can escape the protection domain to uPro's runtime for requests such as inter-domain communication.

**A running example.** We will use a simplified version of Linux `chfn` (change finger) program to illustrate uPro's components. It is a command-line program that is used to change a user's finger information stored in `/etc/passwd`. Figure 2 shows the program's main files and the relationship between these files. The file `control.c` takes user input and makes coordination. After the user inputs the password, `control.c` invokes the `authenticate` function in `auth.c` for password authentication. The `authenticate` function computes the hash of the input password and compares the result with the one stored in `/etc/shadow`. Note for security Linux stores hashed passwords in a shadow password file, not in `/etc/passwd`. If the authentication succeeds, `control.c` asks for more user input about the finger information. It then invokes the `write_entry` function in `chinfo.c` to modify the corresponding entry in `/etc/passwd`.

`chfn` is a privileged program as it accesses system password files. If all modules are put into one protection domain, a malicious input might trigger a buffer overflow in `control.c` and allow an attacker to access sensitive files. Therefore, one security goal is to contain damages even when the code in `control.c` has been hijacked by the attacker. We can achieve this goal by putting the three modules into separate protection domains and restricting the privileges of `control.c` so that it can only accept user input and cannot directly perform privileged accesses.

## 3.1 Protection domains vs. execution units

An OS process combines a *protection domain* with an *execution unit*. It is a protection domain in that each process has its

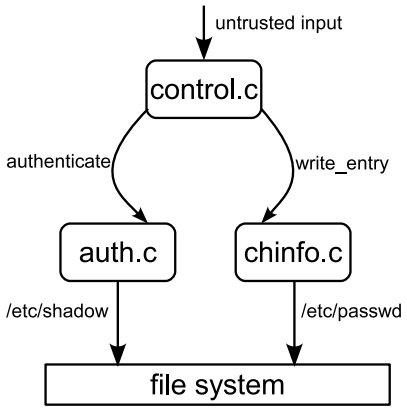


**Figure 2: `chfn`'s file structure.**

own address space and privileges are associated with process IDs. A process is also an execution unit in that each process runs as a separate entity. However, bundling protection domains with execution units is unnecessary in situations where an application desires only a new protection domain for isolating sensitive code; creating an extra process would force the application to create an execution unit as a by-product. Take `chfn` as an example. Putting each module into a process would create three processes, but it is sufficiently enough to have just one execution unit whose control can flow through three protection domains. Creating extra processes would also necessitate some form of synchronization between execution units.

In uPro, protection domains and execution units are orthogonal concepts. Protection domains can be created without creating new

execution units and vice versa. An execution unit is realized by an OS thread, which is lightweight in both creation and scheduling. On the other hand, there is no isolation among traditional threads; all threads in one process share most of the resources (e.g., code, data, and file descriptors). In uPro, isolation between threads is provided by protection domains. In particular, protection domains are realized through Software-based Fault Isolation (SFI), which allows isolation in the same address space.

Separating these two notions makes it possible to combine protection domains and execution units in flexible ways. One such combination allows a single-threaded program to execute in multiple protection domains. For instance, the `chfn` program can be single-threaded but have three protection domains for the three modules. The single thread starts in the domain for the `control` module, and makes calls to other privileged domains. Another way of combining protection domains and execution units is to allow multiple threads to run inside one protection domain. In uPro, a protection domain is a memory region of code and data and it accommodates multi-threading. Of course, the problem of synchronization between threads remains, as in the case without uPro.

DEFINITION 1. *An **execution unit** is a schedulable unit in OS, which can independently run.*

DEFINITION 2. *A **protection domain** is a memory region that holds code and data and a set of privileges for the code. The code portion and the data portion are disjoint. The code portion is readable and executable, but not writable. The data portion is readable and writable, but not executable.*

Through the SFI enforcement, code in a protection domain can read/write memory contents only within the data portion. It can only transfer control to addresses within the code portion, except for a set of well-defined exit points (i.e., trampolines). At runtime, the code region in each protection domain is executable-only and the data region is non-executable, both enforced by the hardware, thus resisting code-injection attacks (e.g., stack smashing). Other types of attacks such as Jump-Oriented Programming [14] and Non-control Data Attacks (e.g., by overrun the data controlling branches) are not resisted by uPro.

Since they are orthogonal concepts, uPro manages the protection domains and the execution units at runtime via separate data structures. The control structure for a protection domain stores entries including its ID, the domain type (section 3.2), the sandbox size, an array of references to the control structures of execution units that can execute in the domain, and others. The control structure for an execution unit stores entries including the ID, its type (section 3.2), a cross-domain call (section 3.2 and 3.4) stack, and others.

## 3.2 Security architecture language

An application's security architecture often plays a critical role in achieving its security goal. However, designing an appropriate security architecture is non-trivial. Developers often need to balance multiple considerations, including security, performance, and ease of migration from the existing architecture. Furthermore, an application's architecture may need adjustments to respond to new security threats. An application may even need multiple architectures to meet varying requirements on performance and security. For instance, Chromium provides users with command-line switches to choose from several architectures, from everything in one process to fine-grained separation [22]. Therefore, what is desired is a mechanism that helps developers explore the architectural design space and easily describe the alternative architectures.

To support declarative security architectures, uPro provides a declarative Security Architecture Language (SAL). A SAL model describes how an application is structured into protection domains in terms of what modules should go into what domains, and what privileges a domain owns. SAL offers flexibility to developers. Assuming the application has been sufficiently partitioned, switching to an alternative architecture involves changes at the model level instead of the code level. Moreover, SAL provides a formal notation for encoding architectural graphs. A SAL model conveys the overall security architecture and is more concise and easier to understand than the implementation.

At a high level, a SAL configuration provides three types of information for an application: (1) types of protection domains and execution units used in the application; (2) the initial instances of protection domains and execution units of the application; (3) enhanced type signatures of interface functions. The enhanced type signatures of interface functions are used to generate stub code; examples will be given when we discuss the stub-code generation in section 3.4. We next use `chfn` to illustrate how the first two types of information can be specified in SAL.

Figure 3 presents two possible `chfn` configurations in the SAL syntax. The left column presents a three-domain configuration. For each domain, the configuration declares its set of privileges, including the set of OS system calls it can invoke, the set of allowed inter-domain communications, the set of system files it can access, and others. The three-domain configuration shows that a `Control` domain is allowed to perform OS reads and writes, which are used to interact with `stdin` and `stdout`. However, it is not allowed to touch any system files. The `Control` domain imports the `authenticate` function from `Auth` and `write_entry` from `ChInfo`. Other external functions cannot be invoked by `Control`. An `Auth` domain can read the `/etc/shadow` file, which stores users' hashed passwords. It exports the `authenticate` function. Similarly, a `ChInfo` domain is allowed to read and write the `/etc/passwd` file and exports the `write_entry` function.

The three-domain configuration also declares an execution unit `EUChfn`. An execution unit has one master domain and multiple worker domains. An execution unit starts execution from the `main` function of the master domain and may transfer the control in and out of worker domains because of inter-domain communication.

SAL distinguishes between *types* and *instances*. For instance, `Control` is a protection-domain type and the application may create several domain instances of the type. A configuration file includes an `init_config` section for specifying the domains and execution units created during initialization. The three-domain configuration of `chfn` specifies that a `control` domain of type `Control` is created. Argument "`control.o`" is the object file that should be loaded into the domain. It is the compilation result of `control.c` and is created when assembling the `chfn` package. The initial configuration also specifies that an `auth` domain, a `chinfo` domain, and an execution unit should be created. When loading the `chfn` application, uPro's loader reads the configuration file and creates the initial configuration according to `init_config`.

Although having the notion of types does not help much in `chfn`, it is useful when dynamic creation of domains and execution units is necessary. When there is dynamic creation, it is impossible to specify all instances statically. But since the number of types of domains (or execution units) is usually fixed, we can specify those types in the configuration file and allow dynamic creation of instances of those types. As an example, in software systems that support dynamic loading and unloading of plug-ins, in the simplest case only one type of protection domains is needed for all plug-ins; those plug-ins share the same set of privileges. Another example is the Apache web server, which will be discussed later.

SAL helps developers describe the alternative security architec-

(a) A three-domain configuration of `chfn`

```
PD_Type Control {
  import = OS:{read, write},
           Auth:{authenticate},
           ChInfo:{write_entry};
  sysfile = {};
}

PD_Type Auth {
  import = OS:{open, close, read};
  export = {authenticate};
  sysfile = {"/etc/shadow"};
}

PD_Type ChInfo {
  import = OS:{open, close, read, write};
  export = {write_entry};
  sysfile = {"/etc/passwd"};
}

EU_Type EUChfn {
  Master={Control};
  Workers={Auth,ChInfo};
}

init_config {
  create control : Control("control.o");
  create auth : Auth("auth.o");
  create chinfo : ChInfo("chinfo.o");
  create main : EUChfn(control, {auth, chinfo});
}
```

(b) A two-domain configuration of `chfn`

```
PD_Type ControlAndChInfo {
  import = OS:{open, close, read, write},
           Auth:{authenticate};
  sysfile = {"/etc/passwd"};
}

PD_Type Auth {
  import = OS:{open, close, read};
  export = {authenticate};
  sysfile = {"/etc/shadow"};
}

EU_Type EUChfn {
  Master={ControlAndChInfo};
  Workers={Auth};
}

init_config {
  create cac  : ControlAndChInfo("cac.o");
  create auth : Auth("auth.o");
  create main : EUChfn(cac, {auth});
}
```

**Figure 3: Two SAL configurations of `chfn` (the type signatures of interface functions are not included).**

tures, facilitating them to explore the design space of an application's security architectures. For `chfn`, the simplest one is to put all modules into one protection domain, but it does not offer much security. We have discussed a three-domain configuration for `chfn`. Another configuration is presented in the right column of Figure 3. In this two-domain configuration, `control.c` and `chinfo.c` are put into one domain, and `auth.c` is in a separate domain. This configuration results in better efficiency because there is no need to cross domains when `control.c` invokes the `write_entry` function. The downside is that it is less secure because a vulnerability in `control.c` may cause leak of information in `/etc/passwd`. Note in both configurations, an attacker can tamper with `/etc/passwd`. Even in the three-domain configuration, the attacker can issue `write_entry` requests after taking over the `control.c` domain. If that is a concern, developers can restructure `chfn` in the following way: `control.c` cannot invoke `write_entry` directly and has to send both the password and the new finger information to `auth.c`; the code in `auth.c` performs authentication and then requests `write_entry` upon a successful authentication. This involves more code changes but ensures the policy that only authenticated users can change `/etc/passwd`. Clearly, which configuration is the best choice depends on the tradeoff between security, performance, and other factors.

Two special domains are implicitly included in every configuration. The first one is the OS domain, which exports the system call interface. The second one is a per-application trusted domain. Code in the trusted domain is not sandboxed. Developers may choose to put performance-critical portions of applications into the trusted domain, at the risk of a larger TCB.

The full SAL syntax is presented in appendix A. For each domain, developers can declare the following privileges: the set of interface functions it can invoke; types of protection-domain, execution unit and communicator of which it is allowed to create instances; the set of accessible system files. The major limitation of uPro's system of privileges is that privileges are statically assigned to domain types and cannot be dynamically created or transferred.

## 3.3 Application loading

An application package is launched through uPro's loader. A package consists of a set of object files and a configuration file that describes the application's security architecture. The major steps in uPro's application launching are described as follows.

(1) Configuration file processing. The SAL configuration file is parsed and its information is converted into an internal representation based on hash maps and hash sets. The information is used by uPro's runtime for making runtime decisions. For example, the uPro API `pd_create` allows an application to dynamically create a protection domain of a type. That protection-domain type must have been declared in the configuration file so that the runtime knows how to create a domain instance.

(2) Signal handler registration. uPro takes over the handling of certain signals for better support of applications in the case of multiple execution units within a process. For instance, the Unix semantics is that the whole process is terminated if a

multi-threaded process raises a `SIGSEGV` signal. By contrast, uPro registers its own signal handler for `SIGSEGV` so that only the execution unit throwing that signal is terminated. However, uPro's approach of taking over signals has its downsides; more discussion about this is in Section 3.5.

(3) Loading the initial instances of protection domains and execution units. According to information that is declared in the `init_config` section of the configuration file, the uPro loader sets up the initial protection domains and execution units. For each protection domain, uPro loads the object file required for the domain after verifying that code in the file satisfies the SFI sandbox policy. Furthermore, trampolines that are used to invoke uPro's runtime services are installed in each domain.

(4) Launching the initial execution units. The loader launches the running of the initial execution units by transferring the control to their entry points.

## 3.4 Runtime services

After an application is loaded, code in a protection domain may request uPro's runtime services. A service is requested by transferring the control to the relevant trampoline entry installed during application loading time. Depending on a domain's privileges, certain trampolines may be blocked. For example, if a domain is not allowed to invoke the `write` syscall, then the corresponding trampoline entry is blocked; the entry is filled with `HLT` instructions.

After services are requested through trampolines, uPro saves the domain's state (e.g., register values) in the control block of the domain, copies service-call arguments out to memory owned by uPro's runtime, performs necessary security checks, and invokes the service. One example security check is performed before invoking the file-open syscall. Recall that uPro's configuration allows users to declare the set of system files accessible to a domain; that restriction is enforced before the OS file-open syscall is requested.

We next go through the major runtime services provided by uPro.

**Cross-domain calls.** A cross-domain call (CDC) is similar to a Remote Procedure Call (RPC), except it happens between two protection domains on a local machine. Cross-domain calls are common in privilege-separated applications. One domain may not have direct access to privileged resources and has to invoke functions exported by other domains for indirect access.

In a cross-domain call, the caller and the callee are in two different domains. To support cross-domain calls and returns, uPro's implementation provides a CDC stub code generator and a per-execution-unit CDC stack.

The CDC stub code generator, `cdcgen`, helps developers generate stub code that marshals and unmarshals data exchanged in a CDC. In a SAL declaration file, developers can provide enhanced type signatures of interface functions. For instance, the function `authenticate` exported by `auth.c` has the following declaration:

```
int authenticate([string] char* passwd);
```

The syntax is based on the C function declaration syntax, plus additional annotations in square brackets. The additional annotations help automatic generation of code for data marshaling and unmarshalling. For instance, the `string` annotation above tells that `passwd` is a null-terminated string, instead of a char array of unspecified length. As another example, the following declaration specifies that the first parameter is an integer array, whose length is the second parameter.

```
void foo([dim:n] int* a, int n);
```

Given the declaration of an interface function, `cdcgen` generates stub functions in domains that import and export the function. The stub function for importing domains marshals the arguments of the caller, invokes the uPro API function `cdc`, and unmarshals the return value when `cdc` returns. The `cdc` function copies the marshaled arguments to the callee domain and invokes the corresponding stub function in the exporting domain. The stub function for the exporting domain unmarshals the arguments, calls the actual function, marshals the return value, and invokes another uPro API function `cdc_return`. `cdc_return` copies the return value to the caller domain (if there is one), and transfers the control back.

To support a chain of CDC calls and recursive CDC calls, the implementation of `cdc` and `cdc_return` relies on a per-execution-unit call stack. `cdc` pushes a frame onto the stack; `cdc_return` pops a frame from the stack and uses the information in the next frame to restore the calling context. Each stack frame stores the program counter, general-purpose registers, segment registers, CDC arguments and return values, and others. The CDC call stack resides in an execution unit's control structure, stored outside of the memory of any protection domain. Note each domain still has a per-execution-unit local call stack inside the domain for supporting local function calls and returns.

The CDC call stack also serves as the basis for authorization. Recall that privileges are assigned to protection domains. In uPro, an execution unit has the privileges owned by the top domain in the execution unit's CDC stack. For instance, if an execution unit starts execution in `PD1` and invokes a function in `PD2`, then the execution unit owns the privileges of `PD2` during the function invocation.[2]

**Communicator.** A uPro communicator similar semantics to pipes and sockets in inter-process communication, provided as another inter-domain communication mechanism. Therefore, the communicator abstraction is helpful when we port a multi-process application to run in uPro; pipes and sockets can be converted to communicators. There are two kinds of communicators: unidirectional and bidirectional. Unidirectional communicators are similar to pipes and bidirectional communicators are similar to sockets. When the application invokes uPro API `com_create` and it has the privilege to create a communicator, uPro's runtime creates a uni- or a bi-directional communicator. After both protection domains call `com_open`, the communicator will be registered to them and begin functioning. `com_open` returns a descriptor, which is used to represent the communicator. The creator of the communicator may invoke `com_destroy` to close that communicator. The ability to create a communicator is a domain's privilege and has to be declared in the configuration file.

**Dynamic management of domains and execution units.** uPro provides dynamic protection-domain creation and removal. The API `pd_create` allows one domain to create another domain. There are two creation modes. In the duplication mode, the newly created domain duplicates the state (including code and data) of the creation domain; this is similar to the semantics of `fork`. The second mode is the non-duplication mode, in which one argument of `pd_create` must tell the new object file that is loaded into the new domain. The semantics of the non-duplication mode is similar to the semantics of a `fork` call followed by an `exec` call. The two modes help the process of porting applications to uPro; applications' use of `fork` and `exec` is changed to use of `pd_create` (possibly followed by execution-unit creations). Note that the ability for a domain to create domains of a certain type is also a privilege

[2]The drawback of this design is that it may suffer from confused deputy attacks. An improved design would adopt Java's stack inspection [44], which performs authorization based on the whole stack.

and should be declared in the configuration file. uPro implements a reference-counting mechanism to reclaim protection domains. The control structure of a protection domain remembers the number of execution units that can execute in the domain. When that reference count becomes zero, uPro's runtime releases the domain and its resources (e.g., memory, file descriptors, communicators, etc.).

Execution units can be dynamically created through `eu_create` and destroyed through `eu_destroy`. A set of uPro API functions is used to support synchronization between execution units. They include the support of mutexes, conditional variables and semaphores. For example, `eu_mutex_create` is used to create a mutex; `eu_mutex_lock` and `eu_mutex_unlock` acquire and release the mutex, respectively. In Linux, these synchronization APIs are implemented on top of the pthread library. uPro records a global counter to record the number of active execution units, if the counter hits zero, uPro runtime exits gracefully.

**OS system calls.** To facilitate the porting of existing applications, uPro provides a set of API functions that are similar to Linux system calls, including system calls that perform file and network IOs. Most of these uPro API functions are simply wrappers of the corresponding OS system calls (with code for performing security checks). The benefit of exposing the OS system calls as uPro API functions is that it helps the migration of existing applications. On the other hand, the system-call interface is not portable across OSes and prior research has documented various kinds of pitfalls when security relies on the system-call interface [20]. One future work for uPro is to design a portable, clean API interface for accessing OS resources.

## 3.5 Implementation And Limitations

We started the implementation of uPro's runtime by modifying NaCl's service runtime, but gradually moved away from that. NaCl was designed to isolate browser plug-ins in the Chromium browser and is unsuitable for general applications. In particular, its service runtime allows only one protection domain for each plug-in. It is impossible for one execution unit to run across multiple domains. Furthermore, NaCl does not provide cross-domain calls within the plugin as there is only one protection domain. It also does not support privilege configuration, and the security checks are hard coded into the implementation; consequently it is inconvenient to change privileges of a protection domain.

uPro has been implemented in x86-32 Linux. uPro relies on Google's Native Client (NaCl) SDK to build SFI-compliant object code. The implementation of uPro has three main components: a configuration file parser, `cdcgen`, and its runtime. The configuration file parser converts a SAL configuration file into uPro's internal representation, used by uPro's runtime. The configuration file parser and `cdcgen` are implemented in 3,600 lines of Flex, Bison, and C++ code. The implementation of uPro's runtime includes around 20,000 lines of C code.

**Limitations.** First, the implementation does not support an application to register its own signal handler. Applications that rely on their own signal handlers cannot be ported to uPro. To address this issue, uPro's runtime needs to be changed so that it can deliver signals to domains.

Second, our implementation of `cdcgen` deals with most C types, including primitive types, structs, and arrays. However, its implementation does not deal with pointers with arbitrary aliasing. In the worst case, developers are still required to write their own stub functions.

Finally, the current implementation lacks support for debugging and profiling. Debugging sandboxed and multi-execution-unit programs is challenging.

# 4. EVALUATION

We have carried out a set of experiments to evaluate uPro. These experiments are designed to answer the following questions:

(1) How does uPro's pure software-based fault isolation compare with process-based isolation in terms of performance?
(2) How much developer effort is needed to port applications so that their security architectures can be described using SAL and they can run in uPro?

To answer these questions, we first performed experiments with a set of micro-benchmarks. They are designed to measure the performance of various aspects of uPro, such as context-switch time and cross-domain communication time. Afterwards, we compiled a set of real-world applications and ported them to uPro. The porting process gave us a direct understanding of the amount of effort needed to port applications to uPro. After the porting, we measured the performance and also experimented with different security configurations. We next report our experimental results. All evaluation was carried out on a system with 32-bit Ubuntu 12.04, an Intel Core i7 870 CPU, and 4GB of memory. The `thttpd` and `Apache` experiments required a second remote client machine. The machine is a system with 32-bit Ubuntu 12.04 on Intel Core i3 350. The client and the server were on the same Ethernet.

## 4.1 Micro-benchmarking

A set of micro-benchmarks were used to measure the performance of various aspects of uPro. One interesting question is how different the performance of uPro's software-based isolation is from that of process isolation. We start with a high-level comparison. Process isolation has high overhead in context switches between processes. Switching to a different address space involves expensive translation lookaside buffer (TLB) flushing and refilling[3]. Context switch in software-based isolation is more efficient. On the other hand, uPro brings a few extra categories of overhead than process isolation. First, there is the overhead for performing dynamic checks in the object code for enforcing SFI (e.g., a mask instruction is inserted before an indirect jump). Second, there is the overhead for performing SFI verification. When object code is loaded into a protection domain, uPro invokes an SFI verifier to check whether the code satisfies the SFI sandbox policy. Finally, there is the overhead for performing runtime security checks in uPro. For instance, when a cross-domain call is requested, it checks whether it is allowed by the configuration.

**Context switch time.** We used a `null` service call to measure the time to switch to uPro's runtime. When an application invokes this `null` service call, the control escapes the protection domain to the runtime and then immediately re-enters the domain. The `null` service call time is the minimum overhead for an application to invoke a service call. The following table shows the time for uPro's `null` service call and the time for Linux's `getpid`, both were the average of 100 million calls. The table shows that the time for a uPro context switch is slightly less than the time for switching into the Linux kernel.

| | |
|---|---|
| uPro `null` | 115 ns |
| Linux `getpid` | 146 ns |

[3]Tagged TLBs do not flush all the entries, but flushing part of the entries is still time-consuming.

| Buffer size (KB) | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1,024 | 2,048 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CDC ($\mu$s) | 0.59 | 0.66 | 0.74 | 0.85 | 1.29 | 2.18 | 3.70 | 8.15 | 21.84 | 62.40 | 126.55 | 254.34 |
| Communicator ($\mu$s) | 0.84 | 0.99 | 1.78 | 2.84 | 5.24 | 10.54 | 42.36 | 128.14 | 292.32 | 607.74 | 1187.87 | 2325.75 |
| IPC ($\mu$s) | 0.74 | 1.11 | 2.12 | 3.73 | 7.08 | 14.94 | 54.45 | 170.93 | 376.43 | 781.28 | 1563.60 | 3200.10 |

**Table 1: Performance comparison between CDC, communicator and IPC.**

**Inter-domain communication.** This experiment compared the performance of uPro's inter-domain communication with inter-process communication (IPC). In the IPC case, a pipe was established between two processes and the sender process sent a fixed-size buffer to the receiver process. uPro's inter-domain communication has two kinds: cross-domain calls (CDC) and communicators. For the CDC case, the caller domain invoked a function provided by the callee domain and the buffer was passed as a function argument. For the communicator case, a unidirectional communicator was attached to two protection domains. Two execution units A and B were also created. Execution unit A stayed in the sending protection domain and sent the buffer through the communicator to the receiving protection domain, in which execution unit B ran. In all three cases, the buffer was sent one million times. Furthermore, the receiver read the first byte of every memory page of the received buffer to simulate the use of the buffer.

Table 1 presents the average transmission time for buffers of varying sizes. The results show that when the buffer size increases, CDC performs an order of magnitude better than communicators and IPC, and communicators performs better than IPC. uPro's implementation of CDC is completely in the user space without switching into the OS kernel. Moreover, CDC does not switch address spaces; there is no expensive TLB flushing and refilling involved. uPro's communicators are built on top of OS-level pipes (or socket pairs). Two communicator ends share the same address space and the probability that the scheduling of them does not switch the address space is higher than the process case.[4] This experiment confirms that software-based isolation is better suited than process isolation when inter-domain communication is frequent.

**Execution-unit creation.** We measured the creation time for a process, a pthread, and a uPro execution unit. The following table lists the results. The pthread creation is lightweight compared to the process creation since it does not need to duplicate resource control structures. Although an execution unit in uPro is based on a pthread, its creation involves extra time for context switching to uPro's runtime, security checks and others. Hence, the creation of a uPro execution-unit consumes more time than that of a pthread but less than a process.

| | |
|---|---|
| Process fork | 234.5 $\mu$s |
| Pthread creation | 24.1 $\mu$s |
| uPro execution unit creation | 117.3 $\mu$s |

**Protection-domain creation.** Recall that protection-domain creation has two modes: the duplication mode duplicates the state of the parent domain; the non-duplication mode loads a new object file into the new domain. In both modes, the time spent can be divided into two parts: one is a fixed amount of time ($T$) spent on the control structure allocation and initialization, and trampoline installation. The other part depends on either the size of the accessed pages in the parent domain (in the duplication mode), or the size of the object file to be loaded (in the non-duplication mode). The second part is almost linear to the size and therefore we measure the time per unit size (denoted by $S$). The following table shows the protection domain creation times of both modes (excluding object-file verification time for the non-duplication mode), averaged by executing the relevant programs 10k times.

| | T ($\mu$s) | S ($\mu$s/MB) |
|---|---|---|
| Duplication mode | 129.1 | 340.1 |
| Non-duplication mode | 135.9 | 690.9 |

In the non-duplication mode, there is a step that performs object-file verification for checking SFI compliance. Google NaCl's verifier was used in the verification.

## 4.2 gzip

`gzip` is a command-line compression and decompression tool. We restructured `gzip` 1.2.4 to have three modules: the command-line processing module (CLP), the compression module (COMP), and the IO module. The CLP module passes the input and output file names to the IO module. The IO module reads input-file contents into a fixed-size input buffer, passes the input buffer to the COMP module, which performs compression/decompression and sends the result back to the IO module's output buffer. The IO module then writes the output buffer to the output file. Out of around 10k lines of `gzip` code, the restructuring required us to separate about 600 lines of code from three source files and add about 300 lines of code for interface functions. The build script was also adjusted to build the object files separately.

The COMP module performs complex compression or decompression, and its code contributes to 90% of `gzip`. Therefore, we assume a threat model in which the CLP (consisting of about 200 lines of reviewed code) and IO modules are trusted while the COMP module is not. Our desired policy is that only the specified input and output files will be accessed even if the COMP module has been hijacked (because of malicious input files).

We describe a security architecture for `gzip` that enforces the policy. We call it `ps-gzip` (privilege-separated `gzip`). In this architecture, three `gzip` modules are loaded in three separate protection domains. The CLP and COMP domains have no file-access privileges. It is straightforward to see that the security policy is enforced by this architecture.

We compared the performance of native `gzip` with `ps-gzip` by measuring the time spent on compressing and decompressing files with different sizes. Table 2 shows the results. CDCs, uPro context switches and security checks mainly contribute to the performance overhead.

We also experimented with a slightly different architecture where the CLP and IO modules are put into one domain. To change to the new architecture, no source code was changed. A new configuration file was written and the build script was adjusted. The perfor-

[4]Since there are other threads running in the system, there might be a third thread living in a different address space (e.g., in another process) being scheduled when switching from thread A to thread B in the communicator experiment. In that case, a context switch still happens between different address spaces.

| File size (MB) | 32 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|
| Native `gzip` (sec) | 1.57 | 2.90 | 5.85 | 12.04 | 24.08 |
| `ps-gzip` (sec) | 1.89 | 3.44 | 6.77 | 13.29 | 26.35 |
| Overhead | 16.9% | 15.8% | 13.6% | 9.3% | 8.6% |

**Table 2: `gzip` performance on files of different sizes.**

| | **Throughput**(req/sec) | **Latency**(ms/req) |
|---|---|---|
| Native `thttpd` | 6843.15±218.29 | 6.43±0.19 |
| `ps-thttpd` | 6484.56±588.49 | 7.09±1.06 |
| Overhead | 5.24% | 6.45% |

**Table 3: `thttpd` throughput and latency (the average and standard deviation of 10k static page requests).**

mance of the new architecture has no observable difference from `ps-gzip`.

## 4.3 thttpd

The HTTP server `thttpd` is a privileged program. A vulnerability can potentially allow an attacker to send the server malicious HTTP requests, take over the server, and change arbitrary system files. Since the only system file that `thttpd` is supposed to change is its log file, we restructured `thttpd` 2.25b to have two modules: the client-interactive (CI) module, which performs server initialization (including command line parsing and server configuration file processing), client connection management, HTTP request parsing, web page reading, and response sending; the logging (Log) module appends log entries to a log file. Out of around 10k lines of code in `thttpd`, the restructuring involves extracting the logging code (around 40 lines) in `main.c` and `libhttpd.c` to a separate `log.c` file. The build script was also changed so that the two modules were built separately. For simplicity, we disabled the CGI and http-authentication capabilities in `thttpd`.

We assume a threat model in which the client-interactive module may be hijacked by attackers. We further assume the Log module is trusted. The security policy in which we are interested is that attackers cannot make arbitrary changes to system files under the threat model. To enforce this policy, we developed a simple two-domain configuration, called `ps-thttpd`. The first domain holds the CI module; it is allowed to read the server configuration file and its HTML pages, but not allowed to write to system files. The second domain holds the Log module; it is allowed to write to the log file. The CI domain can send new log entries to the Log domain through a function exported by the Log domain. It is straightforward to see that this architecture enforces the desired policy.

We compared the performance (throughput and latency) of native `thttpd` and `ps-thttpd`. The throughput is the average client requests completed per second and the latency is the average time (millisecond) spent on a single request. The performance depends on the number of concurrent requests so we measured it with concurrent requests from 10 to 100 (with an interval of 10) and took the average. For each number of concurrent requests, 10k requests in total were sent from the client to the server and the experiment was repeated 20 times. Table 3 presents the average and standard deviation for `thttpd` and `ps-thttpd`. As we can see, the performance overhead of `ps-thttpd` compared to native `thttpd` is small.

To better understand the security benefit of `ps-thttpd`, we analyzed the publicly known vulnerabilities of `thttpd`. We searched the National Vulnerability Database [1] and found 10 relevant vulnerabilities for Linux `thttpd`. We analyzed each and checked whether the new architecture would alleviate the damage caused by attacks exploiting the vulnerability. We believe 8 of 10 cases fall into this category. Appendix B shows detailed analysis results. As an example, the buffer overflow vulnerability CVE-1999-1457 can enable attackers to execute arbitrary code. By contrast, `ps-thttpd` executes the vulnerable code in the CI domain, which has restricted privileges. As a result, it cannot execute arbitrary code and hence the damage is alleviated. As a negative example, `ps-thttpd` does not contain CVE-2002-0733. It is a cross-site scripting vulnerability that may result in arbitrary script execution on a victim client's browser; the new architecture improves the security of the server, not a client machine.

## 4.4 Apache

Similar to `thttpd`, the `Apache` web server is a privileged program whose vulnerabilities can allow attackers to wreak havoc on the server. Different from `thttpd`, which is run as a single process, the pre-fork model of `Apache` is a multi-process program. Specifically, each connection is handled by a separate process, ensuring the isolation between connections. Therefore, `Apache`'s pre-fork model serves as a good example to illustrate how to port a multi-process program to run in uPro.

For better security, we restructured `Apache` 2.2.21 (around 270k lines of code). First, we separated the code that writes to system files into separate modules, similar to the restructuring of `thttpd`. In particular, `Apache` writes to two sorts of system files: the log files and the pid file.[5] Therefore, `Apache`'s code is divided into three modules: the client-interactive (CI) module, the logging (Log) module and the pid-file writing (PW) module. The CI module handles the command-line parsing, server configuration file parsing, child process creation, client connection establishment, web page file reading, and response sending. The Log module opens the log files, writes log entries to those files and closes them. Its code was extracted to form a separate file, involving less than 100 lines of code changes. The PW module opens the pid file and writes the initial process ID to that file. The PW module was also separated into a different file (10 lines). `Apache`'s build script was also adjusted to build the object files from source files and the automatically generated CDC glue code.

The second restructuring was to migrate `Apache`'s multi-process architecture to a multi-domain, multi-execution-unit architecture. The code changes to make this happen was actually small. Specifically, we implemented a user-space library function `fork`, which first invokes `pd_create` to duplicate the parent domain and then invokes `eu_create` to create an execution unit to run in the new domain. By directing fork calls to this new function, a process is mapped to a combination of a domain and an execution unit.

A new architecture, called `ps-Apache` and visualized in Figure 4, was developed based on the above restructuring. During initialization, the three modules are put into three protection domains. The configuration also defines two execution-unit types (not instances): `EU_Main` and `EU_Worker`. An execution unit of type `EU_Main` is created during initialization. It is responsible for forking pairs of domains and execution units for handling connections. As discussed, a child process in the original `Apache` is mapped to a domain with an execution unit; the domain duplicates the CI domain and the execution unit is allowed to run in the duplicated CI domain and the Log domain. In terms of privileges, CI domains are allowed to read server configuration files and HTML pages. The single Log domain is allowed to write to log files; it is shared by

[5]The pid file records the ID of the initial server process, which is the parent of all child processes for handling connections; the file is used to facilitate administrators to terminate malfunctioning child processes.

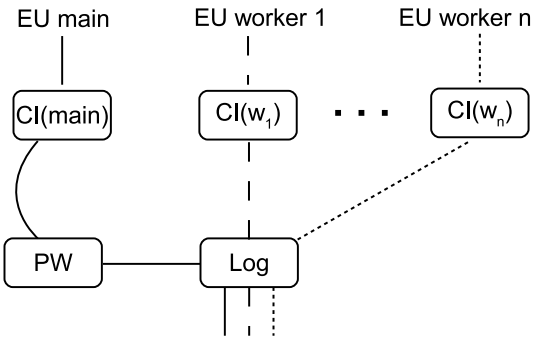


**Figure 4: A new architecture for `Apache`. Lines are used to indicate what domains an execution unit are allowed to enter. For instance, the main EU is allowed to enter the main CI domain, the PW domain, and the Log domain.**

all CI domains and exports functions to CI domains to log events. The single PW domain is allowed to write to the pid file and only the master execution unit is allowed to enter it. This architecture demonstrates the benefit of separating types and instances as the number of domains and execution units is not fixed statically.

Similar to `thttpd`, we assume a threat model in which the CI module may be hijacked by attackers and assume the other two modules are trusted. The architecture aforementioned enforces two interesting security policies. The first policy is that, even if a CI domain has been hijacked, the attacker cannot change any system files. The second policy is about the isolation between connections: even if an attacker has hijacked the CI domain of a connection, other connections' states stored in CI domains are not affected.

We measured and compared the throughput and latency of native `Apache` and `ps-Apache`. Different from `thttpd`, which has only one execution unit at runtime, native `Apache` and `ps-Apache` have multiple execution units when running. We configured both servers so that the number of execution units at runtime was fixed[6]. The measurement and result processing was performed in the same way as in `thttpd`'s experiment, given a specific number of execution units. Table 4 presents the results with the leftmost column showing the number of execution units. In most cases, `ps-Apache`'s overhead is small.

We also carried a security analysis on `ps-Apache` by inspecting the publicly known `Apache` vulnerabilities posted on the Apache website [2]. We picked those vulnerabilities in the Linux platform and with respect to the pre-fork model. There are 35 in total. After analyzing each vulnerability, we concluded `ps-Apache` can alleviate the damages of 10 out of 35 vulnerabilities. Most of the vulnerabilities that cannot be contained are denial-of-service attacks, which are out of the capability of uPro (but see discussion in the future-work section). Due to space limitation, the analysis table is not included in the paper.

## 4.5 Summary of experiments

The results show that uPro is promising. In all cases, uPro provides enhanced security and a high-level security architecture description to developers. The performance overhead is also modest. Each application does require effort to port to uPro, but we believe the security gains are well worth of the effort.

[6] `Apache` can dynamically adjust the number of processes to adapt to the server load.

# 5. FUTURE WORK

In uPro, developers statically assign privileges to code in protection domains. Its drawback is that it does not support dynamic creation/revocation of privileges and transferring of privileges from one domain to another; furthermore, there is no notion that allows treating 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. We believe the model of decentralized information-flow control (DIFC [32]) provides a nice solution, as demonstrated by recent process-level DIFC systems [16, 49, 26]. Therefore, a future improvement on uPro is to combine it with DIFC to implement a protection-domain-level DIFC system. The SAL configuration will allow specification of initial security labels of domains, but labels of domains may change dynamically because of new data received, explicit declassification, or newly created categories of labels. The uPro runtime will monitor inter-domain information flow and prevent forbidden information flow. We believe such an extension will provide much more flexibility to uPro, while maintaining its strength (low context-switch overhead, flexible configuration, and OS independence).

One benefit of uPro is it does not change the OS kernel and can be ported to other OS platforms, similarly to the case that a Java Virtual Machine is portable. We plan to port our Linux implementation to other OSes, including Windows and Mac OS. Because different OSes have different OS system-call interfaces, adopting a POSIX-like system-call interface in uPro while maintaining the ease of migrating legacy applications will be a challenge.

# 6. CONCLUSIONS

We advocate the architectural approach to security. A globally defined and enforced security architecture not only provides a good defense against attacks, but also serves as the basis for building informal and formal assurance cases (as Rushby has argued in his papers on separation kernels [38, 37]). A user-space framework such as uPro provides additional benefits of efficiency, flexibility, and OS independence. The key enabling technique is software-based fault isolation (SFI). Previous work has demonstrated that SFI is suitable for isolating untrusted third-party components such as browser plug-ins and device drivers. uPro demonstrates that SFI is equally applicable when enforcing more fine-grained security architectures within an application. Our experience suggests that this approach enhances security, and is practical in terms of performance and flexibility. We believe that the same architectural approach can be generalized to enforce a variety of security policies, from information-flow policies to availability policies for addressing denial-of-service attacks.

# 7. REFERENCES

[1] `http://web.nvd.nist.gov/view/vuln/search`. [Online; accessed on 17-April-2012].

[2] `http://httpd.apache.org/security/vulnerabilities_22.html`. [Online; accessed on 13-April-2012].

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

[4] external data representation. `http://tools.ietf.org/html/rfc4506`.

[5] Mig - the mach interface generator. `http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/abstracts/mig.html`.

[6] Selinux. `http://selinuxproject.org`.

[7] 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.

| # of EUs | **Throughput**(req/sec) | | | **Latency**(ms/req) | | |
|---|---|---|---|---|---|---|
| | Native `Apache` | `ps-Apache` | Overhead | Native `Apache` | `ps-Apache` | Overhead |
| **10** | 6545.18 ±31.59 | 4718.89 ±379.52 | 27.90% | 6.92 ±0.03 | 9.66 ±0.78 | 28.41% |
| **20** | 6665.18 ±65.17 | 6375.17 ±76.44 | 4.35% | 6.80 ±0.07 | 7.10 ±0.09 | 4.30% |
| **30** | 6699.87 ±76.59 | 6628.79 ±28.99 | 1.06% | 6.76 ±0.08 | 6.83 ±0.03 | 1.00% |
| **40** | 6694.28 ±75.64 | 6622.99 ±23.49 | 1.06% | 6.77 ±0.07 | 6.84 ±0.03 | 1.00% |
| **50** | 6637.37 ±50.78 | 6595.12 ±23.78 | 0.64% | 6.82 ±0.05 | 6.87 ±0.03 | 0.63% |
| **60** | 6628.08 ±61.28 | 6572.84 ±20.32 | 0.83% | 6.83 ±0.06 | 6.89 ±0.02 | 0.78% |
| **70** | 6644.14 ±68.31 | 6513.02 ±29.22 | 1.97% | 6.82 ±0.07 | 6.95 ±0.03 | 1.91% |
| **80** | 6623.34 ±54.48 | 6479.53 ±31.51 | 2.17% | 6.84 ±0.06 | 6.99 ±0.04 | 2.16% |
| **90** | 6683.27 ±202.34 | 6522.88 ±92.11 | 2.40% | 6.79 ±0.20 | 6.94 ±0.10 | 2.22% |
| **100** | 6674.80 ±325.15 | 6528.16 ±104.10 | 2.20% | 6.81 ±0.03 | 6.94 ±0.11 | 1.92% |

**Table 4:** `Apache` **throughput and latency (the average and standard deviation for 10k static page requests).**


[8] 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.

[9] AKRITIDIS, P., COSTA, M., CASTRO, M., AND HAND, S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *18th Usenix Security Symposium* (2009), pp. 51–66.

[10] ANSEL, J., MARCHENKO, P., ERLINGSSON, Ú., TAYLOR, E., CHEN, B., SCHUFF, D., SEHR, D., BIFFLE, C., AND YEE, B. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *ACM Conference on Programming Language Design and Implementation (PLDI)* (2011), pp. 355–366.

[11] BARTH, A., JACKSON, C., REIS, C., AND CHROME, G. The security architecture of the Chromium browser. Tech. rep., 2008.

[12] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), pp. 309–322.

[13] 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.

[14] CHEN, P., XING, X., MAO, B., XIE, L., SHEN, X., AND YIN, X. Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS '11, ACM, pp. 20–29.

[15] COX, R. S., GRIBBLE, S. D., LEVY, H. M., AND HANSEN, J. G. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy (S&P)* (2006), pp. 350–364.

[16] 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.

[17] 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.

[18] ERLINGSSON, Ú., AND SCHNEIDER, F. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW)* (1999), ACM Press, pp. 87–95.

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

[20] GARFINKEL, T. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS* (2003).

[21] GARLAN, D., MONROE, R., AND WILE, D. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, 2000, pp. 47–68.

[22] GOOGLE. The Chromium projects: Process models. `http://www.chromium.org/developers/design-documents/process-models`, 2008.

[23] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure execution via program shepherding. In *11th Usenix Security Symposium* (2002), pp. 191–206.

[24] KRISHNAMURTHY, A., METTLER, A., AND WAGNER, D. Fine-grained privilege separation for web applications. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)* (2010), pp. 551–560.

[25] KROHN, M., EFSTATHOPOULOS, P., FREY, C., KAASHOEK, M. F., KOHLER, E., MAZIÈRES, D., MORRIS, R., OSBORNE, M., VANDEBOGART, S., AND ZIEGLER, D. Make least privilege a right (not a privilege). In *Proceedings of the 10th Conference on Hot Topics in Operating Systems (HotOS)* (2005).

[26] 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.

[27] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *15th Usenix Security Symposium* (2006).

[28] MEDVIDOVIC, N., AND TAYLOR, R. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering 26*, 1 (2000), 70–93.

[29] METTLER, A., WAGNER, D., AND CLOSE, T. Joe-E: A security-oriented subset of Java. In *Network and Distributed Systems Symposium (NDSS)* (2010).

[30] MILLER, M. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, MD, 2006.

[31] MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21*, 3 (May 1999), 527–568.

[32] MYERS, A., AND LISKOV, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering Methodology 9* (Oct. 2000), 410–442.

[33] NEUMANN, P., AND WATSON, R. Capabilities revisited: A holistic approach to bottom-to-top assurance of trustworthy systems. In *Fourth Layered Assurance Workshop* (2010).

[34] PAYER, M., AND GROSS, T. R. Fine-grained user-space security through virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE)* (2011), pp. 157–168.

[35] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *12th Usenix Security Symposium* (2003), pp. 231–242.

[36] REIS, C., AND GRIBBLE, S. D. Isolating web programs in modern browser architectures. In *EuroSys* (2009), pp. 219–232.

[37] RUSHBY, J. Design and verification of secure systems. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (1981), pp. 12–21.

[38] RUSHBY, J. Proof of separability: A verification technique for a class of a security kernels. In *Symposium on Programming* (1982), pp. 352–367.

[39] SALTZER, J., AND SCHROEDER, M. The protection of information

in computer systems. *Proceedings of The IEEE 63*, 9 (Sept. 1975), 1278–1308.

[40] SCOTT, K., AND DAVIDSON, J. Safe virtual execution using software dynamic translation. In *Proceedings of the 18th Annual Computer Security Applications Conference* (2002), ACSAC '02, pp. 209–218.

[41] SEHR, D., MUTH, R., BIFFLE, C., KHIMENKO, V., PASKO, E., SCHIMPF, K., YEE, B., AND CHEN, B. Adapting software fault isolation to contemporary CPU architectures. In *19th Usenix Security Symposium* (2010), pp. 1–12.

[42] 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.

[43] 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.

[44] WALLACH, D. S., AND FELTEN, E. W. Understanding java stack inspection. In *IEEE Symposium on Security and Privacy* (1998), pp. 52–63.

[45] WATSON, R., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. Capsicum: Practical capabilities for UNIX. In *19th Usenix Security Symposium* (2010), pp. 29–46.

[46] WITCHEL, E., RHEE, J., AND ASANOVIĆ, K. Mondrix: memory isolation for linux using Mondriaan memory protection. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2005), pp. 31–44.

[47] 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).

[48] ZDANCEWIC, S., ZHENG, L., NYSTROM, N., AND MYERS, A. Secure program partitioning. *ACM Transactions on Compututer Systems (TOCS) 20*, 3 (2002), 283–328.

[49] 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.

[50] 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.

# APPENDIX

## A. SAL GRAMMAR

SAL grammar is shown below. In the grammar, non-terminals are enclosed in angle brackets and we use $\overline{\langle t \rangle}$ to stand for a list of ⟨t⟩s.

⟨cfg⟩ ::= $\overline{\langle \text{ty\_decl} \rangle}$ ⟨init_cfg⟩ $\overline{\langle \text{cdc\_decl} \rangle}$
⟨ty_decl⟩ ::= ⟨pd_ty_decl⟩ | ⟨eu_ty_decl⟩ | ⟨com_ty_decl⟩
⟨pd_ty_decl⟩ ::= `PD_Type` ⟨id⟩ {
  `import` = ⟨id⟩ : {$\overline{\langle \text{id} \rangle}$};
  `export` = {$\overline{\langle \text{id} \rangle}$};
  `pd_creation` = {$\overline{\langle \text{id} \rangle}$};
  `eu_creation` = {$\overline{\langle \text{id} \rangle}$};
  `com_creation` = {$\overline{\langle \text{id} \rangle}$};
  `sysfile` = {$\overline{\langle \text{string} \rangle}$};
}
⟨eu_ty_decl⟩ ::= `EU_Type` ⟨id⟩ {
  `master` = {⟨id⟩};
  `workers` = {$\overline{\langle \text{id} \rangle}$}
}
⟨com_ty_decl⟩ ::= `COM_Type` ⟨id⟩ {
  `kind` = ⟨com_kind⟩;
  `sender` = ⟨id⟩;
  `receiver` = ⟨id⟩
}
⟨com_kind⟩ ::= `single` | `double`
⟨init_cfg⟩ ::= `init_config` { $\overline{\langle \text{creation} \rangle}$ }
⟨creation⟩ ::= ⟨pd_creation⟩ | ⟨eu_creation⟩
⟨pd_creation⟩ ::= `create` ⟨id⟩ : ⟨id⟩ (⟨string⟩)
⟨eu_creation⟩ ::= `create` ⟨id⟩ : ⟨id⟩ (⟨id⟩, {$\overline{\langle \text{id} \rangle}$})
⟨cdc_decl⟩ ::= ⟨annotated_type⟩ ⟨id⟩ ($\overline{\langle \text{param} \rangle}$)
⟨param⟩ ::= ⟨annotated_type⟩ ⟨id⟩
⟨annotated_type⟩ ::= [⟨attr⟩] ⟨c_type⟩
⟨attr⟩ ::= `string` | `dim` :⟨int⟩ | `dim` :⟨id⟩

## B. THTTPD SECURITY ANALYSIS

Table 5 presents the security analysis on `ps-thttpd` in terms of containing native `thttpd` vulnerabilities.

| Vulnerability | Analysis |
|---|---|
| CVE-2009-4491 | Not contained. Log-escape-sequence injection vulnerability. The logging function does not sanitize non-printable characters in a log entry. The attack happens when the web server is executed in a foreground process that displays logs in the terminal or when the log files are viewed with tools such as "cat" or "tail". Since no escape sequence check is added before logging, `ps-thttpd` does not contain this vulnerability. |
| CVE-2007-0664 | Contained. This vulnerability leads to arbitrary file reading; `ps-thttpd` restricts the set of client-readable files to the set defined in the configuration file. |
| CVE-2003-0899 | Contained. A buffer overflow vulnerability in the `defang` function in the CI module. Its damage is contained by `ps-thttpd`. |
| CVE-2002-1562 | Contained. This vulnerability allows attackers to read arbitrary files in the root directory. `ps-thttpd` contains this vulnerability by restricting the system files that can be visited. |
| CVE-2002-0733 | Not contained. This cross-site scripting vulnerability may lead to arbitrary script execution on a victim client's browser and this cannot be contained by `ps-thttpd` because it protects only the server. |
| CVE-2001-1496 | Contained. An off-by-one buffer overflow vulnerability happening in CI is contained by `ps-thttpd`. |
| CVE-2001-0892 | Contained. This vulnerability allows attackers to bypass file permissions and read arbitrary files. `ps-thttpd` forbids arbitrary file access by restricting system-file operations. |
| CVE-2000-0359 | Contained. This is a buffer overflow vulnerability in CI; it is contained by `ps-thttpd`. |
| CVE-1999-1456 | Contained. This vulnerability allows attackers to read arbitrary files. `ps-thttpd` forbids arbitrary file access by restricting system-file operations. |
| CVE-1999-1457 | Contained. This is a buffer overflow vulnerability in a function that parses dates in CI. It is contained by `ps-thttpd` by limiting the privileges of CI. |

**Table 5: `thttpd` vulnerability analysis**