SHILL: A Secure Shell Scripting Language

SHILL: A Secure Shell Scripting Language

Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong
Harvard School of Engineering and Applied Sciences

Abstract

The Principle of Least Privilege suggests that software should be executed with no more authority than it requires to accomplish its task. Current security tools make it difficult to apply this principle: they either require significant modifications to applications or do not facilitate reasoning about combining untrustworthy components.

We propose SHILL, a secure shell scripting language. SHILL scripts enable compositional reasoning about security through contracts that limit the effects of script execution, including the effects of programs invoked by the script. SHILL contracts are declarative security policies that act as documentation for consumers of SHILL scripts, and are enforced through a combination of language design and sandboxing.

We have implemented a prototype of SHILL for FreeBSD and used it for several case studies including a grading script and a script to download, compile, and install software. Our experience indicates that SHILL is a practical and useful system security tool, and can provide fine-grained security guarantees.

1 Introduction

Users of commodity operating systems often need to execute untrustworthy software. In fact, this is the common case: due to errors or malicious intent, software regularly does not behave as expected. The Principle of Least Privilege (POLP) [31] requires that software should be given only the authority it needs to accomplish its functionality. If adhered to, this principle (also known as the Principle of Least Authority) can help protect systems from erroneous or malicious software.

However, commodity systems and their secure tools fail to adequately support POLP. First, it is difficult for the user of a commodity system to determine what authority a given piece of software requires to execute correctly. Second, current mechanisms for limiting authority are difficult to use: they are either coarse-grained or require significant changes to existing software, and are often not available to all users [16]. For both of these reasons, users tend to execute software with more authority than is necessary.

For example, consider scripts to grade homework submissions in a computer science course. Students submit source code, and a script `grade.sh` is run on each submission to compile it and run it against a test suite. The submission server must execute `grade.sh` with sufficient authority to accomplish its task, but should also restrict its authority to protect the server from student-submitted code and ensure the integrity of grading. At a coarse grain, the server should allow `grade.sh` to access files and directories necessary to compile, run, and record the scores of homework submissions, and deny access to other files or resources. This ensures, for example, that a careless student's code won't corrupt the server and a cheating student's code won't modify or leak the test suite. At a fine grain, each call to `grade.sh` to grade a single submission should be isolated from the grading of other submissions. This ensures, for example, that a cheating student cannot copy solutions from another submission.

Securing a script such as `grade.sh` is difficult, as it requires balancing functional and security requirements. To begin with, it is a priori unclear what authority `grade.sh` needs to execute correctly. While the author of the script may know, the user must examine the code to try to determine what authority it requires. If the user can identify the required resources, she can use existing tools for sandboxing program execution (e.g., [20, 3, 15, 14]) to achieve the coarse-grained security requirements. However, it is difficult to use the same tools to enforce the fine-grained security requirements described above. This is because achieving these requirements requires that each invocation of `grade.sh` is given different privileges, i.e., it must be executed in a differently configured sandbox. Configuring all of these sandboxes correctly is error prone, so users often forgo

```
provide grade :
  {submission : is_file && readonly,
   tests : is_dir && readonly,
   working : dir(+create_dir with full_priv),
   grade_log : is_file && writeable,
   wallet : ocaml_wallet} → void;
```

Figure 1: SHILL contract for a grading script

fine-grained security and violate POLP.

To address these issues, we introduce the SHILL programming language. SHILL is a secure shell scripting language with features that help apply POLP in commodity operating systems.[1] At the core of SHILL are declarative security policies that describe and limit the effects of script execution, including effects of arbitrary programs invoked by the script.

These declarative security policies can be used by producers of software to provide fine-grained descriptions of the authority the software needs to execute. This, in turn, allows consumers of software to inspect the software's required authority, and make an informed decision to execute the software, reject the software, or apply a more restrictive policy on the software. The SHILL runtime system ensures that script execution adheres to the declared security policy, providing a simple mechanism to restrict the authority of software.

Two key features enable SHILL's declarative security policies: language-level *capabilities* and *contracts*. SHILL scripts access system resources only through capabilities: unforgeable tokens that confer privileges on resources. SHILL scripts receive capabilities only from the script invoker; SHILL scripts cannot store or arbitrarily create capabilities. Moreover, SHILL uses *capability-based sandboxes* to control the execution of arbitrary software. Thus, the capabilities that a user passes to a SHILL script limit the script's authority, including any programs it invokes. SHILL's contracts specify what capabilities a script requires and how it intends to use them. SHILL's runtime and sandboxes enforce these contracts, hence they serve as fine-grained, expressive, declarative security policies that bound the effects of a script.

For example, Figure 1 shows a SHILL contract for a script to grade a single student submission (corresponding to the `grade.sh` script described above). It is a declarative security specification for the function grade, which takes 5 arguments: a read-only file submission (i.e., the student's source code), a read-only directory tests (containing the test suite), a "working directory" in which the script may create subdirectories with full privileges, a writeable file grade_log for recording the student's grade, and a "wallet" that provides sufficient capabilities to invoke the OCaml compiler. This contract serves two purposes: it clearly describes what grade needs to execute correctly and it also provides guarantees about what grade may do when invoked. Given this contract, a user can be confident that grade satisfies the security requirements described above, even though grade will compile and execute student-submitted code. Specifically: grade will not read any other student's submission; grade will not communicate over the network (as it has no capability for network access); grade will not corrupt the test suite nor write any files other than the grade log and subdirectories it creates within the working directory. The implementation of grade (not shown) focuses solely on the functionality for grading, and is not concerned with enforcing security requirements.

SHILL offers language abstractions for reasoning about the authority of pieces of software and their composition. Specifically, SHILL (1) introduces a capability-based scripting language with language abstractions (such as contracts and wallets) to use capabilities effectively, and (2) implements, on a commodity operating system, capability-based sandboxes that extend the guarantees of the scripting language to binary executables and legacy applications. These language abstractions, and the enforcement of these abstractions, make it possible to manage authority and follow POLP, even when using and combining untrusted programs.

The rest of the paper is structured as follows. In Section 2 we present the design of SHILL. Our implementation of SHILL in FreeBSD 9.2 is described in Section 3. We evaluate SHILL by using it to implement several case studies, and measure the overhead of SHILL's security mechanisms. We present the evaluation results in Section 4. Section 5 describes related work.

2 Design and security of SHILL

SHILL aims to meet the following five goals:

1. Script users can control the authority of a script, i.e., what system resources it can access or modify.
2. Script users can understand what authority a script needs in order to accomplish its functionality.
3. Security guarantees of scripts apply transitively to other programs the script may invoke, including arbitrary executables.
4. SHILL separates the security aspects of scripts from functional aspects, reducing the impact of security concerns on the effort required to write scripts.
5. SHILL is compatible with commodity operating system abstractions.

[1] SHILL is not an interactive shell, but rather a language that presents operating system abstractions to the programmer and is used primarily to launch programs. Other languages currently used for this purpose include Perl, Python, and the scripting portion of Bash.

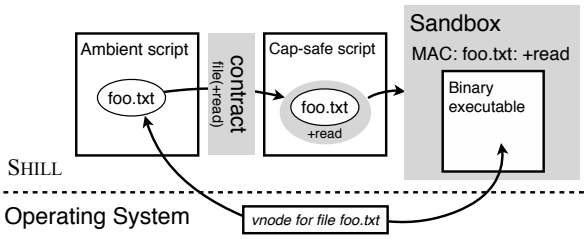

Figure 2: SHILL in a nutshell

To meet these goals, SHILL uses a combination of language design and mandatory access control-based sandboxing.

In most scripting languages, scripts can access a resource (such as a file) using the resource's well-known global name. Access control is based on the user on whose behalf the script executes. Thus, a script's authority is *ambient* (i.e., it derives from the script's execution context) [25], and a script may access any and all resources that the invoking user may access. SHILL's security is based on capabilities instead of ambient authority.

There are two kinds of SHILL scripts: *capability-safe* SHILL *scripts*, and *ambient* SHILL *scripts*. Capability-safe SHILL scripts play the same role as regular shell scripts, but do not have ambient authority and must be given capabilities to access resources. Ambient SHILL scripts are used to create the initial set of capabilities to give to capability-safe scripts. They do have ambient authority, but are very restricted: ambient scripts can only create capabilities for system resources and invoke capability-safe SHILL scripts.

Each capability-safe SHILL script comes with a contract that is enforced by the language runtime. A capability-safe SHILL script can use the capabilities it possesses to access resources using SHILL's built-in functions, if allowed by the contract. SHILL scripts can also invoke arbitrary executables in *capability-based sandboxes*. A capability-based sandbox is created with a set of capabilities, and enforces a mandatory access control policy that restricts the executable's behavior based on those capabilities and their contracts.

Figure 2 depicts the life cycle of a capability for a file named foo.txt. First, an ambient script acquires a capability for the file from the operating system using the user's ambient authority. This capability is then passed to a capability-safe script via a contract, which restricts the privileges on the capability to +read (i.e., the capability can be used only to read foo.txt, not to write to it, etc.). The capability-safe script then runs an executable in a sandbox, granting it the capability to read the file.

Threat model In SHILL's threat model, some capability-safe scripts (and the executables they invoke) are not trusted. However, their behavior is restricted by their contracts and the capabilities they are given: a capability-safe script (and any executables it invokes) can access resources only as permitted by its contract and the capabilities it possesses. Of course, the contract that accompanies a script may also be untrustworthy: a user should inspect the contract and understand its security implications before passing capabilities to the script. The benefit of SHILL's approach is that it is much easier to inspect and understand the declarative contract than to examine the script itself.

SHILL's trusted computing base includes the operating system kernel and SHILL runtime. SHILL does not explicitly defend against malicious scripts or executables that exploit security flaws in the kernel or SHILL itself.

The rest of this section describes how SHILL's design and features contribute towards these goals, and provides an introduction to SHILL via several small examples.

2.1 Controlling script authority

Ambient authority makes writing scripts easy: if a script needs to access a resource, it can simply use the resource's name to access it. However, ambient authority makes it difficult to understand and control the potential effect of executing a script. First, the authority of a script is not easily deducible from its code, a problem that is exacerbated when the script invokes other scripts or executables. Second, commodity operating systems do not provide easy mechanisms to limit authority of an execution context, for example, by allowing a user to temporarily restrict permissions in a fine-grained way.

Authority in SHILL is controlled by *capabilities*. In order to access a resource, a SHILL script must have a capability for that resource. SHILL scripts can only acquire capabilities as arguments provided by the user, or by deriving them from other capabilities (e.g., using a directory capability to acquire a capability for a file in the directory). These restrictions, known as *capability safety*, lie at the heart of the security of SHILL scripts. Capability safety makes it possible for users to control the authority of SHILL scripts they invoke (Goal 1).

Figure 3 presents a snippet of SHILL code that demonstrates how SHILL scripts use capabilities. It defines a function find_jpg for recursively finding all the files with extension .jpg within a given directory. Argument cur is a capability for either a file or a directory. In contrast with standard scripting languages, cur is not a string that names a file, but is a capability that denotes it, much like a file descriptor. If cur is a file capability and the name of the file ends with .jpg, then the script uses the built-in function path to get the string for the path to the file,[2] and

[2]The library function has_ext also uses path.

```
1  find_jpg = fun(cur,out) {
2     # if cur is a file with extension jpg,
3     # output its path to out.
4     if is_file(cur) && has_ext(cur, "jpg") then
5        append(out, path(cur));
6
7     # if cur is a directory, recur on its contents
8     if is_dir(cur) then
9        for name in contents(cur) {
10          child = lookup(cur, name);
11          if !is_syserror(child) then
12             find_jpg(child, out);
13       }
14 }
```

Figure 3: SHILL script snippet to find .jpg files

appends it to the pipe or file capability out (lines 4–5).

If cur is a directory capability, then the built-in function contents is used to get the list of names of children of cur. For each child, the script calls lookup(cur, name) to obtain a capability for the child (line 10), which is then used in a recursive call to find_jpg (line 12).

Conceptually, SHILL capabilities correspond to operating system representations of resources, such as file descriptors, and built-in functions such as append and lookup are wrappers for the corresponding system calls.

SHILL enforces capability safety by restricting the expressiveness of the scripting language. While SHILL offers full-fledged language features and rich libraries, comparable to other scripting languages, the built-in functions for using resources require capabilities as arguments. In addition, SHILL does not have mutable variables and capabilities are not serializable. This means that SHILL scripts cannot store or share capabilities through memory, the filesystem, or the network. For controlled sharing of capabilities, SHILL provides *wallets*, capabilities for packaging and managing collections of capabilities. We discuss wallets further in Section 2.4.1.

SHILL scripts provide the same protection from confused deputy attacks [12] as traditional capability systems. Furthermore, filesystem operations that produce new capabilities (such as lookup) do not allow scripts to arbitrarily traverse the filesystem. For instance, a script cannot use the capability for the current directory cur and lookup(cur,"..") to obtain the parent directory of cur.

2.2 Contracts

Capability safety makes it possible to limit the authority granted to a SHILL script by carefully selecting what capabilities to pass as arguments. Unfortunately, needing to pass capabilities explicitly makes it harder for script users to deduce how to use scripts and compose them to complete more complicated tasks. At its core, this is a problem of defining the script's interface: how does the script communicate what resources it requires and how it will use those resources?[3]

SHILL addresses these issues by providing expressive, fine-grained and enforceable interfaces for scripts (Goal 2) following the *Design by Contract* paradigm [23, 24]. Every function that a SHILL script exports (i.e., makes available to users of the script) is accompanied by a *contract* that describes the arguments the function expects and the result it returns. For example, the following snippet is a contract for the find_jpg function from Figure 3:

```
provide find_jpg :
  {cur : is_dir ∨ is_file, out : is_file} → void;
```

The **provide** keyword indicates that the function find_jpg is exported. The contract for the function is {cur : is_dir ∨ is_file, out : is_file} → void. Each function contract has two parts: the precondition and the postcondition. The precondition of our example states that find_jpg takes two arguments: a capability cur that is either a directory or a file capability, and a file capability out. Following Unix convention, file capabilities include capabilities for files, pipes, and devices. The postcondition void means that no value is returned.

The precondition of the contract above describes what kind of capabilities find_jpg needs, but does not indicate how the function intends to use these capabilities. SHILL allows us to give a more precise contract for find_jpg:

```
provide find_jpg :
  {cur : dir(+contents, +lookup, +path) ∨ file(+path),
   out : file(+append)} → void;
```

This version specifies not only what kind of capabilities the function consumes but also what privileges it requires on these capabilities. Each privilege, such as +path or +contents, corresponds to an operation on a capability. A capability contract with a set of privileges restricts what operations that capability can be used for.

Some operations on capabilities, such as lookup, produce more capabilities. Capability contracts can specify the privileges a script should have on these derived capabilities. For example, privilege +lookup with { +path, +stat } indicates that any capabilities derived using the lookup operation should only have the +path and +stat privileges. When a privilege confers the right to derive new capabilities but does not come with a modifier (such as the +lookup privilege in the contract for find_jpg), the derived capability has the same privileges as its parent capability.

Each contract establishes an agreement between two

[3] Traditional shell scripting languages such as Bash or Python also suffer from these issues, but the use of ambient authority masks them: scripts typically receive much more authority than needed.

parties: the provider of the value with the contract and the value's consumer. As part of the agreement, each party promises to live up to its contractual obligations. In this way, a contract both describes a guarantee one party provides and a requirement the other party demands. For function contracts, the consumer's obligations are to supply function arguments that satisfy the precondition, and the provider must produce a result that satisfies the post-condition. For capability contracts, the provider agrees to provide a capability of the appropriate kind with *at least* the specified privileges while the consumer promises to use the capability as if it has *at most* the specified privileges. For example, according to the find_jpg contract, users of find_jpg must supply a file capability that permits the append operation for the out argument, while find_jpg itself promises not to call other operations on the capability, such as read.

The SHILL runtime checks whether parties live up to their obligations by monitoring execution and checking that values are used in accordance with their contracts. For example, when find_jpg is called with a capability for a directory and a capability for the output file, the body of find_jpg does not receive the capabilities themselves. Instead, each contract wraps the underlying capability with a *proxy*. These proxies enforces the contracts for cur and out by intercepting calls to operations on the capabilities and allow them only if permitted by the contract. If the body of find_jpg attempts to perform an operation that isn't permitted—such as reading the contents of out or unlinking cur—the proxy will indicate that a contract violation has occurred. If a contract is violated, the SHILL runtime aborts execution and, to help with auditing and debugging, indicates which part of the script failed to meet its obligations.

2.3 Securing arbitrary executables

SHILL security guarantees must be completely enforced: even if a script calls other scripts or runs arbitrary executables, its authority should be restricted to its capabilities, and it should meet its contract obligations (Goal 3). When SHILL scripts invoke only other SHILL scripts, we achieve SHILL's security guarantees easily because of the language's semantics. However, scripts also invoke executable programs.

To ensure that these programs cannot violate SHILL's security guarantees, SHILL scripts may only invoke executables inside a *capability-based sandbox*. When a sandbox is created, it is given a set of capabilities. The SHILL sandbox limits the authority of the sandboxed executable to the authority implied by the set of capabilities.

Scripts can invoke an executable in a sandbox by calling the built-in function exec. For example, the following snippet executes the file jpeginfo in a sandbox with the arguments -i and a given file:

```
exec(jpeginfo, ["jpeginfo","-i",file], stdout = out,
     extras = [libc,libjpeg])
```

The exec function has two required arguments. The first is a file capability with the +exec privilege. The second is a list of string arguments to provide to the executable. SHILL programmers can also provide as arguments to executables capabilities for files or directories instead of string representations of their paths. In this case, the path to the given file is passed to the executable as an argument. The exec function also takes some optional arguments, including capabilities to use for standard input, output, or error (stdout = out), and extra capabilities needed by the program (extras = [libc,libjpeg]). This set of extra capabilities is often quite large. In Section 2.4.1, we describe abstractions to help manage capabilities for sandboxes.

SHILL sandboxes enforce a capability-based mandatory access control (MAC) policy on the sandboxed execution. For example, the sandbox for jpeginfo allows access only to resources indicated by capabilities passed as arguments to exec (which, for the jpeginfo example above, are the jpeginfo, file, out, libc, and libjpeg files and directories). Moreover, if any of these capabilities comes with a contract, the MAC policy further limits access to the resource according to the capability's contract.

This capability-based MAC policy is enforced *in addition* to the operating system's discretionary access control (DAC) policies: an operation on a resource by a sandboxed execution is permitted only if it passes the checks performed by the operating system based on the user's ambient authority and is also permitted by the capabilities possessed by the sandbox. Note that sandboxed executables never possess capabilities that allow them to circumvent the MAC policy. For example, no sandboxed executable has a capability to unload kernel modules, including the module that enforces the MAC policy. Section 3.2 describes how we implement capability-based sandboxes using the TrustedBSD MAC framework.

2.4 Writing SHILL scripts

SHILL's security benefits come at the cost of extra effort to write scripts. Nonetheless, we strive to make it easy to write SHILL scripts while obtaining stronger security guarantees than traditional shell scripting languages. To make it easier to write scripts, SHILL offers security abstractions such as *capability wallets* and pushes security concerns to the interfaces between scripts.

2.4.1 Security abstractions

SHILL requires that any access of a protected resource requires an appropriate capability. However, even sim-

```
1 provide jpeginfo :
2   {wallet : native_wallet, out : file(+write,+append),
3    arg : file(+read,+path)} → void;
4
5 jpeginfo = fun (wallet,out,arg) {
6   jpeg_wrapper = pkg_native("jpeginfo",wallet);
7   jpeg_wrapper(["-i",arg],stdout = out);
8 }
```

Figure 4: Executing jpeginfo in a sandbox using wallets

ple executable programs require access to a surprising number of files. For example, executing `cat` in a sandbox requires providing eight capabilities to libraries and configuration files in addition to capabilities for the executable itself and the input and output.

Consider a SHILL script that executes `cat` in a sandbox. One can imagine a contract that requires a separate argument for each of the eight capabilities that `cat` requires. While precise, such a contract imposes a significant burden on both the script writer (since the need for these capabilities will be exposed in the interface for the script) and the script user (who will need to supply these capabilities individually).

Another possibility is a contract that takes important capabilities separately (e.g., for the executable and the input and output) and takes all other capabilities in a list. Although succinct, this contract burdens the script's user, who has no idea what capabilities should be in this list.

We introduce *capability wallets* as a mechanism to automate and simplify the discovery, packaging, and management of capabilities that sandboxes need to run executables. Conceptually, a capability wallet is a map from strings to lists of capabilities. To reduce the burden on script writers, SHILL provides *wallet contracts*, which describe contracts for the capabilities associated with individual keys or groups of keys. To reduce the burden on script users, SHILL provides library functions to automate the collection and packaging of capabilities into wallets.

Figure 4 shows a script that uses a capability wallet to create a sandbox for the program `jpeginfo`. The first argument to the jpeginfo function has the contract native_wallet (line 2). A native_wallet is a particular kind of capability wallet that can be built using functions from SHILL's standard library. It collects together the capabilities needed to invoke executables and can be used with other functions from the SHILL standard library that present a familiar path-based interface for identifying and running executables. The capabilities in a wallet are derived from capabilities the user explicitly grants to the script. Thus despite its path-based interface, a native wallet is still capability safe.

This script uses one of the standard library functions, pkg_native, to create a wrapper containing all of the capabilities needed to run the `jpeginfo` executable in a sandbox (line 6). The script then calls the wrapper, supplying the executable arguments and input and output capabilities (line 7).

SHILL's standard library comes with a rich collection of functions that construct and manipulate wallets, wallet contracts and wallet-derived sandboxes. Section 3.1.4 presents these utilities in further detail.

2.4.2 Pushing security to interfaces

SHILL's contracts allow the programmer to separate the security specification of a script from the implementation of its functionality (Goal 4). The SHILL runtime ensures that contracts are enforced, removing the need for defensive code that checks and protects the use of capabilities. Consider the find_jpg function from Figure 3: the implementation is simple, and the security guarantee is provided by its contract. This separation makes it possible to strengthen or relax a script's security guarantees by modifying its contract. Indeed, in Section 2.2 we saw two different contracts for the find_jpg function, one of which provides a more precise security guarantee.

SHILL's contract system is rich and expressive, allowing precise specifications of security guarantees. For example, users can define their own contracts by creating contract combinators and user-defined predicates written in SHILL itself.

SHILL's contracts can also be used to write security specifications that provide different guarantees to different script users. Consider the script in Figure 5. This script recursively finds files and performs an action on these files. (It is more general than the find_jpg script of Figure 3.) The function find takes three arguments: a file or directory capability cur, a function filter that is used to select files, and a function cmd to apply to all selected files. Lines 5–16 implement find's functionality. Note that this code is straightforward, and does not directly address security concerns.

Lines 1–3 define the contract for find, using a *bounded parametric-polymorphic contract*. The polymorphic contract declares that for any contract X, the function find can be called with arguments cur, filter, and cmd such that cur satisfies contract X, filter satisfies contract X → is_bool (i.e., filter is a function that expects a value that satisfies X and returns a boolean), and cmd satisfies contract X → void (i.e., cmd is a function that expects a value that satisfies X and returns no value).

The polymorphic contract is *bounded* because the contract X on capability cur that the caller provides must have at least the privileges +lookup and +contents. Moreover, the contract requires that find can use only the +lookup and +contents privileges of the cur argument or

```
1  provide find :
2  forall X with {+lookup,+contents} .
3    {cur : X, filter : X → is_bool, cmd : X → void} → void;
4
5  find = fun(cur, filter, cmd) {
6    if is_file(cur) && filter(cur) then
7      cmd(cur);
8
9    # if cur is a directory, recur on its contents
10   if is_dir(cur) then
11     for name in contents(cur) {
12       child = lookup(cur, name);
13       if !is_syserror(child) then
14         find(child, filter, cmd);
15     }
16 }
```

Figure 5: A find script with a polymorphic contract

derived capabilities, even though contract X may specify more privileges. Importantly, the contracts for arguments filter and cmd allow these functions to use all of the privileges that X specifies. In essence, the contract of find dynamically seals [28] the argument cur as it flows into the body of the function through contract X, and unseals it as it flows out to the functions filter and cmd.

The contract on find allows clients to use find in different ways. For example, one client may use it with a filter that examines file creation times (which requires the +stat privilege). Another client may use find with a filter that inspects a file's name (which requires +path, but not +stat). For both clients, the contract guarantees that the implementation of find itself cannot use either the +stat or +path privileges, even though it invokes the functions filter and cmd.

2.5 Interaction with ambient authority

Figures 3, 4, and 5 show SHILL scripts that consume and use capabilities. But where do capabilities come from? SHILL is intended for use with commodity operating systems, and so we must provide a mechanism to transition from the ambient world of the operating system to SHILL's capability-safe world (Goal 5).

To that end, in addition to the capability-safe scripts we have described so far, users of SHILL scripts write *ambient scripts* which inherit the authority of the invoking user and are *not* capability safe. Ambient scripts are used to create capabilities and pass them to functions that capability-safe scripts provide. Consequently, the language of ambient scripts is extremely restricted: ambient scripts contain straight line code that can import capability-safe scripts, create capabilities for resources using file paths and other global names, and call

```
1  #lang shill/ambient
2
3  require shill/native;
4  require "jpeginfo.cap";
5
6  root = open-dir("/");
7  wallet = create_wallet();
8  populate_native_wallet(wallet,root,
9    "~/Downloads/jpeginfo",
10   "/lib:/usr/local/lib",
11   pipe_factory);
12
13 dog = open-file("~/Documents/dog.jpg");
14 jpeginfo(wallet,stdout,dog);
```

Figure 6: Ambient script to call jpeginfo

functions exported by capability-safe scripts. Ambient scripts are brief and delegate all interesting tasks to the capability-safe scripts they import. Also, capability-safe scripts cannot import ambient scripts, which ensures that capability-safe scripts cannot use ambient scripts to obtain additional capabilities. Ambient scripts must reason carefully about their interaction with untrusted scripts. Contracts and capabilities help with this.

Figure 6 shows an ambient script that creates appropriate capabilities and then invokes the jpeginfo function from the script in Figure 4. The annotation #lang shill/ambient on line 1 indicates that this is an ambient script.[4] Line 3 loads a SHILL library script that helps create capability wallets. Line 4 loads the capability-safe script from Figure 4.

Lines 8–11 create an appropriate capability wallet to run `jpeginfo` by calling the trusted standard library function populate_native_wallet. Line 13 creates a capability for ~/Documents/dog.jpg. The capability has all privileges that the invoking user is allowed for this file; when the capability passes through a capability contract, it loses all privileges except those stated in the contract. Line 14 invokes jpeginfo with the capability wallet, a capability to standard out, and the capability to `dog.jpg`.

3 Implementation

We have implemented a prototype of SHILL as a kernel module and set of userspace tools for FreeBSD 9.2. The userspace tools include the SHILL compiler, runtime, and standard library. The kernel module implements capability-based sandboxes and provides capability-safe versions of several POSIX system calls.

[4] Capability-safe scripts have the annotation #lang shill/cap on the first line; we omitted this annotation in Figures 3, 4, and 5.

3.1 Language

We implement the SHILL language as an extension to Racket [9] using Racket's macro system and tools for building languages [39]. Prototyping SHILL in this way allows us to use Racket functionality where it meets our security requirements. In particular, we used Racket's contract mechanism to implement SHILL contracts.

A distinguishing feature of SHILL is capability safety: access to resources occurs only through capabilities, and creation of capabilities is limited. To achieve capability safety at the language level, we (1) provide language-level capabilities and capability contracts; (2) restrict the expressiveness of the language; and (3) provide a capability-based language runtime for SHILL.

3.1.1 Capabilities and their Contracts

Capabilities in the SHILL language are object-like values that encapsulate low-level capabilities such as file descriptors or sockets. Each operation on a capability is implemented by calling the corresponding operation on the low-level capability. Different kinds of capabilities support different operations. For example, supported operations on files and pipes include reading, writing, and changing modes. Directories also have capabilities for listing, adding, or removing directory entries. Each operation has a corresponding privilege that can be present or absent on a given capability. In total, SHILL has twenty-four different privileges for filesystem capabilities and seven different privileges for sockets. Socket privileges are further refined by connection type.

We chose privileges and operations to align closely with the operations that our capability-based sandbox can interpose on, so that we can ensure that giving a capability to a sandbox conveys the same authority as giving that capability to a SHILL script. There are two kinds of SHILL capabilities that do not encapsulate a system resource directly: the pipe_factory and socket_factory capabilities. These capabilities encapsulate, respectively, the right to create new pipes or sockets. The pipe_factory capability has a create operation that returns a pair of pipe ends. Each pipe end is a file capability. In our prototype implementation, SHILL scripts cannot create or manipulate sockets directly (which can be addressed by adding built-in functions for socket operations to the language). We do restrict a sandbox's permitted socket operations: a sandbox must possess a socket_factory capability to be allowed to create and use sockets.

We implement SHILL contracts using Racket contract combinators [8, 7] that create proxies [38] for capabilities, allowing us to interpose on operations and check privileges before allowing an operation. These proxies also store information about the privilege restrictions each contract imposes.

Resource	Language	Sandbox
Directories, files, links	Capabilities	Capabilities
Pipes	Capabilities	Capabilities
Character Devices	Capabilities	Capabilities†
Sockets (IP,Unix)	Capabilities	Capabilities
Sockets (other)	Denied	Denied
Processes	ulimit‡	Confinement
Sysctl	Denied	Read-only
Kernel environment	Denied	Denied
Kernel modules	Denied	Denied
POSIX IPC	Denied	Denied
System V IPC	Denied	Denied

Figure 7: System resources and how each is protected in the SHILL language and capability-based sandboxes.
†: In our prototype, character devices are only partially controlled by capabilities, see Section 3.2.3.
‡: SHILL allows calls to the exec function to specify ulimit parameters for the child process.

3.1.2 Restricting the SHILL language

To achieve capability safety in SHILL, we carefully choose which language features and libraries of Racket are available in SHILL. We allow access to certain Racket libraries, such as the regular expression library, but prevent access to all others, including Racket's system library and Racket's macro system. SHILL scripts are allowed to import only SHILL capability-safe scripts.

The ambient SHILL language (see Section 2.5) has further restrictions: it may not do anything other than import capability-safe SHILL scripts, create strings and other base values, define (immutable) variables, and invoke functions. However, unlike the capability-safe SHILL language, it may create capabilities using ambient authority.

3.1.3 Capability-based runtime

We implemented a capability-based language runtime for SHILL that provides operations to access files and other resources through file descriptors. (The Racket libraries for accessing files and other resources rely on ambient authority, and are thus not suitable for our use.) File descriptors provide unforgeable tokens that can serve as low-level capabilities for directories, files, links, pipes, sockets, and devices. Our capability-based runtime provides wrappers for the `*at` family of system calls which provide a file-descriptor based interface to common operations like opening, reading, and writing files. Our runtime further restricts these system calls by requiring that arguments that specify sub-paths contain only a single component. For example, the pathname argument to `openat` may be `alice` but not `alice/dog.jpg` or `../bob`. Our runtime also provides wrappers for standard system calls which can be used by SHILL's ambient language to create capabilities for system resources.

Most but not all FreeBSD system calls that manipulate the filesystem have a version that consumes file descriptors rather than paths. The `linkat`, `unlinkat`, and `renameat` system calls use file descriptors to designate target directories, but rely on paths to designate files. Thus, a call to `linkat` can not be guaranteed to link to the correct file without risking a time-of-check-to-time-of-use vulnerability. Our kernel module adds three system calls to address these deficiencies: `flinkat`, which installs a link to a file in a directory given file descriptors for both the file and the directory; `funlinkat`, which takes a name and file descriptors for a file and a directory and removes the link at the given name if it refers to the file; and `frenameat`, which is similar to `funlinkat` but also installs a link to the file in a target directory. The module also provides a version of `mkdirat` that returns a file descriptor for the newly created directory.

We also add a new `path` system call that attempts to retrieve an accessible path for a file descriptor from the filesystem's lookup cache. SHILL uses this system call to provide a relatively robust mechanism to translate SHILL capabilities into paths to provide as arguments to sandboxed executables. If the `path` system call fails, SHILL uses the last known path at which the file was accessible.

Our prototype implementation of SHILL does not provide support for all system resources. Interaction with resources that do not correspond to capabilities is either restricted or denied entirely. Figure 7 lists system resources and how SHILL controls access to these resources in the language and in capability-based sandboxes. There is no fundamental obstacle to providing capability support for all resources, though doing so would require additional modifications to the system call interface. For example, we would need to provide a low-level capability for processes, similar to Capsicum's *process descriptors* [43].

3.1.4 Standard Library

SHILL's standard library provides a number of capability-safe scripts that help programmers write SHILL scripts. The filesys script provides capability-based functions that emulate common tasks such as resolving paths and symlinks. The io script provides printf-like wrappers around write and append for formatted output. The contracts script provides abbreviated definitions of common contracts. For example, a programmer can specify the contract readonly rather than the more verbose

```
dir(+read-symlink,+contents,+lookup,
    +stat,+read,+path) ∨ file(+stat,+read,+path).
```

Capability wallets Recall that capability wallets are maps from strings to lists of capabilities that help automate and simplify the discovery, packaging, and use of capabilities to invoke executables in sandboxes. SHILL provides functions for creating and using capability wallets. For example, the native script in the standard library provides two functions for using native wallets to invoke executables (as in Figures 4 and 6): populate_native_wallet and pkg_native.

Function populate_native_wallet helps create a native wallet. Its arguments include path specifications for where to search for executables and libraries (i.e., colon-separated strings, analogous to environment variables `$PATH` and `$LD_LIBRARY_PATH`), and a directory capability to use as a root for the path specifications. In addition, it takes a map (of strings to lists of strings) from known libraries to the file resources those libraries depend on. Function populate_native_wallet uses the directory capability to resolve the path specifications (i.e., converts the lists of strings to lists of capabilities), and places these capabilities in a native wallet. It also resolves the known dependencies (i.e., the map from known libraries to the file resource path names) into a map from strings to lists of capabilities, and places the resolved map into the native wallet.

Function pkg_native takes a native wallet and a file name (of an executable file) and searches the path capabilities in the native wallet for a capability for the executable. The function then invokes `ldd` to obtain a list of libraries that the executable depends on, and searches the library-path capabilities for capabilities for the required libraries. Once these capabilities are gathered, pkg_native uses the map of known dependencies to gather additional capabilities needed to run the executable. Function pkg_native then returns a function that encapsulates a call to exec with all capabilities needed to run the executable. Figure 4 shows an example script that uses pkg_native.

3.2 Capability-based sandbox

The SHILL sandbox is implemented as a policy module for the TrustedBSD MAC Framework [41] (hereafter, "the MAC framework"). The MAC framework allows FreeBSD's access control mechanisms to be extended with third-party mandatory access control policies by mediating access to sensitive kernel objects and invoking access control checks specified by third-party policy modules. The framework also provides a policy-agnostic mechanism for attaching security labels to kernel objects. Mechanisms with similar functionality are available on Linux and Apple's OS X.

3.2.1 Session lifecycle

Each process executing in a SHILL sandbox is associated with a *session*. Processes in the same session share the same set of capabilities and can communicate via sig-

nals. Processes spawned by a process in a session are by default placed in the same session. However, sessions are hierarchical: a sandboxed process inside session S_1 can spawn a process inside a new session S_2, which has fewer capabilities that S_1. This allows SHILL-aware executables to further attenuate their privileges.

New sessions are created by invoking the system call `shill_init`, which creates a session and associates it with the current process. A new session initially has no capabilities of its own. Capabilities possessed by the parent session can be granted to the new session until the process invokes the `shill_enter` system call. Once `shill_enter` is called, the session allows only operations permitted by capabilities it was granted explicitly.

3.2.2 From capabilities to MAC labels

Each system resource protected by a SHILL capability corresponds to an underlying kernel object: a filesystem vnode, pipe, device, or socket. Using the MAC framework's ability to attach labels to kernel objects, SHILL labels these kernel objects with a *privilege map*: a map from sessions to sets of privileges. A privilege map records the privileges that each session has for the given kernel object. Privileges in the privilege map correspond directly to privileges of SHILL capabilities.

When a SHILL script calls exec, the SHILL runtime sets up a sandbox by forking a new process, creating a new session, and granting the session the capabilities passed to exec. It then calls `shill_enter` before transferring control to the executable.

When a sandboxed process invokes a system call relevant to a resource protected by SHILL, we use the privilege map for that resource to check whether the process's session has sufficient privileges for the operation. If there are insufficient privileges, the system call aborts with an error but the process is otherwise allowed to continue.

Derived capabilities In the SHILL language, some operations on SHILL capabilities yield derived capabilities. For example, using a directory capability, a script might obtain capabilities for children of the directory, or might obtain a capability for a new file created in that directory. In the sandbox, we track these derived capabilities by updating privilege maps in response to operations on kernel objects. To enable this, we extended the MAC framework with two additional hooks: `mac_vnode_post_lookup` and `mac_vnode_post_create`. These entry points are invoked after a lookup or create operation completes successfully, and allow the SHILL policy module to update the privilege map on the resulting vnode. For example, if session S has privilege +lookup with {+stat,+path} on a vnode for a directory `d`, and a process in that session successfully invokes system call `openat(d, "child", flags)`, then the SHILL policy module updates the privilege map for the vnode for file `child` to add privileges +stat and +path for session S.

Path traversal To achieve fine-grained confinement in the filesystem, SHILL scripts are not permitted to follow the "`..`" entry of a file or directory capability. However, simply disallowing use of "`..`" in SHILL's capability-based sandboxes would break many existing programs. Instead, the sandbox allows any lookup operation on a directory if the session has the +lookup privilege, but only propagates privileges when the lookup would have been permitted in the SHILL language, that is, when the directory entry requested is not "`..`".[5]

Example Consider a sandboxed process attempting to call `open("../alice/dog.jpg", O_RDONLY)` from the current working directory `/home/bob`. This system call invokes a series of low-level `lookup` operations on filesystem objects to resolve the path and create a file descriptor for the designated resource.

Figure 8 depicts the process of completing these operations in a SHILL sandbox. Shaded boxes around nodes in the file system denote privileges held by the current session. The current working directory is indicated with a solid arrow. Dashed arrows represent low-level lookup operations, and a dashed box around a node represents privileges propagated in response to a lookup operation.

In the left diagram, the current session has a capability to the vnode corresponding to `/home/alice` and a capability to the current working directory. The first operation (lookup "`..`" in `/home/bob`) is permitted because the process has the +lookup privilege, but privileges are not propagated to the vnode for `/home`. Thus, the second operation (lookup `alice` in `/home`) fails because the session does not have the necessary privileges. The `open` system call returns `EACCES` to indicate that the process had insufficient privileges.

The right diagram considers the same scenario, but where the session also has a +lookup privilege to the directory `/home`. In this case, the session is permitted to look up `alice` in `/home`. The final operation (lookup `dog.jpg` in `/home/alice`) also succeeds. These two lookups propagate privileges from the parent nodes to the results of the lookup. Looking up `dog.jpg` in `/home/alice` grants the session the privilege +read on the vnode representing `dog.jpg`, since the session had privilege +lookup with {+read} on the vnode for `/home/alice`. Thus, the call `open("../alice/dog.jpg", O_RDONLY)` succeeds.

[5] We also do not propagate privileges when the directory entry is "`.`", since this can lead to privilege amplification. For example, if session S has only the privilege +lookup with +stat on directory `d`, then calling `openat(d, ".", flags)` would give S the +stat privilege on `d`.

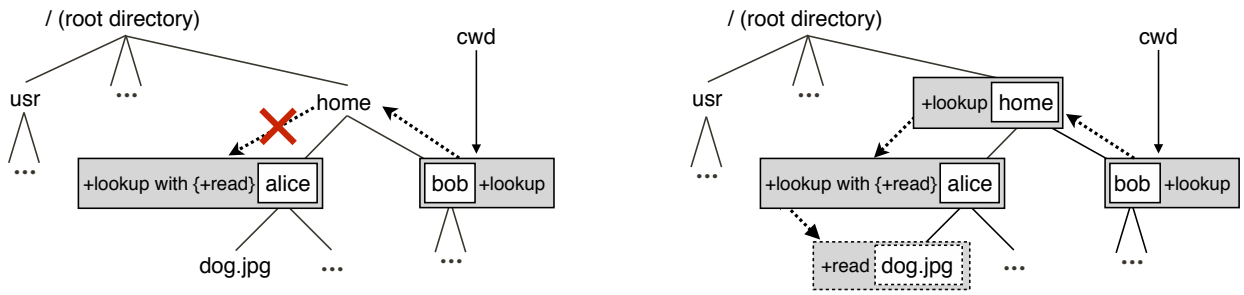

Figure 8: Resolving system call `open("../alice/dog.jpg", O_RDONLY)` in a capability-based sandbox. Left: the session has privileges for `/home/alice` and `/home/bob`, but not `/home`, so the operation fails. Right: the session also has a lookup privilege for `/home`, so the operation succeeds and the lookup privilege on `/home/alice` is propagated to `/home/alice/dog.jpg`.

Note that unlike SHILL scripts, sandboxed executables are vulnerable to confused deputy attacks if they allow clients to specify resources with paths rather than, e.g., file descriptors. However, the authority of the sandboxed execution is still limited by the capabilities it is granted.

Avoiding privilege amplification In the SHILL language, capabilities both designate resources and confer privileges. As a consequence, it is possible to have two separate capabilities to the same resource with different privileges. These separate capabilities may confer less privilege than a single capability with the combined privileges. For example, consider a pair of capabilities to create a network socket, one with sufficient privileges to send but not receive messages at a particular port, and one with sufficient privileges to receive but not send messages on the same port. Because only a single socket can be bound to a port, a program with these capabilities must choose to either send or receive messages.

Since in the SHILL language, scripts cannot combine capabilities, possessing multiple capabilities for the same resource does not lead to privilege amplification. In the capability-based sandbox, however, to avoid privilege amplification the sandbox must prevent two separate capabilities to the same object from being combined to allow additional operations.

For file system operations that create new objects (e.g., creating new files or directories), SHILL requires that a session is never granted conflicting privileges to the same object. For example, if session S currently has privilege +create-file with {+read,+stat,+path} for a directory d, (i.e., the privilege to create read-only files), and due to a lookup from the parent directory we want to propagate privilege +create-file with {+write}, we would *not* merge these privileges, i.e., we would *not* give S the privilege +create-file with {+write,+read,+stat,+path}. While more sophisticated techniques to track privileges are possible, we have found that this conservative approach to prevent privilege amplification works well in practice, and does not break functionality of any of our case studies.

Process interaction The SHILL language provides limited support for operations on processes: SHILL does not have capabilities to control the creation of processes, process synchronization, interprocess communication, etc.

Within capability-based sandboxes, we enforce a simple security policy for operations related to processes: processes in a session can only interact with processes in the same session or a descendent session. A process in a sandbox cannot debug, send signals to, or wait for a process outside of its session.

Debugging SHILL provides several tools for debugging processes running in SHILL sandboxes. First, there is a command-line tool for running a single shell command with capabilities specified in a policy file. Second, for all SHILL sandboxes, logging can be enabled and viewed by privileged users. The log records all of the capabilities and privileges granted during a session in addition to all operations that were denied because of insufficient privileges. Using the command-line tool, a session can be created in debugging mode, which automatically grants the necessary privileges if an operation would fail. We found that running programs in a debugging sandbox and then viewing the logs was a useful starting point for identifying necessary capabilities to provide to a SHILL script. However, as we developed additional standard library support to run common executables, this became less necessary. In most cases, the utilities in the standard library automate the retrieval and collection of capabilities needed to run an executable.

3.2.3 Limitations

SHILL's capability-based sandboxes rely on the MAC framework to implement access control checks based on

capabilities. Thus, the granularity of the MAC framework's mechanism determines the granularity at which our sandboxes protect resources. For example, the MAC framework exposes a single entry point for operations that write to filesystem objects, so we cannot distinguish write and append operations. Conservatively, we enforce that to write (or append) to a file, a session must have both the +write and +append privileges for the file. (Note that in SHILL scripts, privileges can be enforced at fine granularity, since capability safety in scripts relies on language abstractions, not on the MAC framework.)

The MAC framework does not interpose on `read` or `write` operations on character devices. Thus, while the SHILL language exposes `stdin`, `stdout`, and `stderr` as file capabilities and enforces restrictions on how they can be used, sandboxed processes can bypass these restrictions if one of these capabilities abstracts a pseudo-terminal or other device. This limitation is not fundamental and can be resolved by adding entry points to the MAC framework around unprotected operations. It can be mitigated by not granting capabilities to such devices to sandboxes.

4 Evaluation

We evaluate the expressiveness of SHILL through four case studies: a grading script for a programming assignment, a package management script for the GNU Emacs editor, sandboxing the Apache web server, and a find and execute task similar to the example in Section 2. We measure the performance of SHILL via case studies and microbenchmarks. Our evaluation indicates that (1) SHILL is a practical security tool for typical system tasks, (2) SHILL can provide fine-grained security guarantees when scripts are used to compose untrusted software and, (3) its performance cost is pay-as-you-go, i.e., weak security guarantees incur little overhead.

4.1 Case studies

Grading submissions We used SHILL to securely grade student submissions written in OCaml for an undergraduate programming languages course. As a baseline, we wrote a 61-line Bash script that compiles the OCaml source code of each submission and runs the compiled program against a test suite. Results of executing student submissions against the test suite are recorded in a grading directory, one file per student.

With minimal effort, we secured this Bash script in a SHILL sandbox. The capability-safe script that executes the Bash script in a sandbox is 22 lines, of which 14 are the contract for the script. The ambient script that invokes capability-safe script is also 22 lines. The contract guarantees that the grading script can at most: read files in directories containing student submissions and tests; create, modify, and delete new files in a working directory and the output directory; and access the system resources needed to run the compiler and compiled programs.

To demonstrate the finer-grained guarantees of SHILL, we also wrote a version of the grading script exclusively in SHILL. The capability-safe grading script is 78 lines of code, of which six are the script's contract. The ambient script that invokes it is 16 lines. The SHILL script provides all the security guarantees of the sandboxed Bash script, and also ensures that while grading a student's submission, no other student's submission, working-directory files, or results file can be accessed.

The capability-safe SHILL script was developed by manually translating and modifying the original Bash script. String-based references to files were replaced with appropriate capabilities. Calls to programs like `gmake`, `diff`, and `ocamlrun` were replaced with calls to the SHILL standard library to package and execute those programs. To enable this, the ambient script creates a native_wallet initialized with a standard `PATH` and `LD_LIBRARY_PATH`. Contracts for the capability-safe SHILL script ensure that each student's grading file is isolated from other students and that students' programs can't directly modify their grade file. These fine-grained guarantees—which the Bash script does not provide—are achieved by ensuring that the contract on the grading directory allows only the creation of new append-only files, and the functions that compile and execute a student's submission are given no capabilities to other students' grading files.

In developing this script, we debugged several cases where the script had too few privileges to run successfully. In one case, we wrote too restrictive a contract for the submissions directory, forgetting the +lookup privilege. The resulting contract failure indicated which argument had insufficient privileges. After verifying that this privilege was necessary and did not compromise the security guarantees, we fixed the script. We encountered two issues with sandboxed executables. First, the wallet used to launch executables was missing some necessary capabilities: when trying to compile students' submissions, `ocamlc` reported that it was unable to read a file in `/usr/local/lib/ocaml`. Investigating, we realized that OCaml searches for libraries in this directory. Adding the directory to the wallet as a dependency for OCaml executables fixed the issue but revealed another: `ocamlyacc` could not write to `/tmp`. After adding a capability to `/tmp` when invoking `gmake`, the script ran successfully. To ensure isolation between different invocations of `gmake`, we used a contract on the `/tmp` capability to specify that sandboxed processes can only read, modify, or delete files or directories they create.

Package Management We used SHILL to write an installation script for GNU Emacs (similar to what may be found in a package manager). The script provides functions to download, compile, install, and uninstall Emacs. Unlike a typical package manager, the script has a detailed security interface for each function. For example, only the function for downloading the source code can access the network, and only the install function can write to the intended installation directory. In addition, the install function is restricted from reading, altering, or removing any existing files in the installation directory, and the uninstall function's contract gives a list of files that it is permitted to remove. The package manager comprises 114 lines of ambient code, and 91 lines of capability-safe code, of which 45 specify contracts.

Apache web server To showcase how SHILL handles networking applications, we used SHILL to develop a sandbox for the Apache webserver, version 2.2. We tested the performance of the web server by using the Apache Benchmark tool to download a 50MB file served by Apache five thousand times using up to 100 concurrent connections. In addition to its required libraries, the script's contract gives the webserver read-only access to configuration files and web content directories, the ability to create and use sockets, and write-only access to log files. The ambient script is 27 lines, and the capability-safe script is 30 lines, of which 20 lines are contracts.

Find As another example of how programmers can use SHILL to gradually strengthen the guarantees of scripts, we developed two versions of a SHILL script for a find and execute task. Our scripts find all files with extension .c in the BSD source tree that contain the string "`mac_`", the prefix on entry points for the MAC framework. Completing this task requires visiting 57,817 files and invoking `grep` on the 15,376 files with extension .c.

The simpler version is a SHILL script that launches a sandbox for the command

```
find /usr/src -name "*.c" \
              -exec grep -H mac_ {} \;
```

The ambient script is 11 lines and calls a 27-line capability-safe script, of which 5 lines are contracts. The contract ensures that the sandbox has access only to `/usr/src` and files necessary to run `find` and `grep`.

The second version uses the find function (Figure 5) to find files with the extension .c and invokes `grep` in a sandbox for each matching file. In addition to the guarantees of the previous version, this script provides the fine-grained guarantee that the files that `grep` operates on are exactly the files selected by the find function. Note that our first script does not provide this guarantee: paths passed to `grep` may resolve to different files. The ambient script is 9 lines, and the capability-safe script is 60 lines, of which 11 are contracts.

4.2 Performance Analysis

Our prototype implementation focuses on providing fine-grained security guarantees, and we have not yet optimized performance. However, to verify that the performance costs of SHILL are commensurate with the security guarantees, we use the case studies as benchmarks. We also develop benchmarks for sub-tasks of the Emacs installation script (download, untar, configure, make, make install, make uninstall). For each benchmark, we derive a command line invocation to achieve the same task as the case study outside of SHILL (if such a command was not already part of the case study).

We measured the performance of each benchmark in three different configurations. The "Baseline" configuration executes the command on FreeBSD without the SHILL kernel module installed. The "SHILL installed" configuration executes the command with the kernel module installed (but not active). The "Sandboxed" configuration uses a SHILL script to create a sandbox for the command. Where applicable, we also executed a "SHILL version" of the case study that replaces the command.

We ran each configuration of each benchmark 50 times and computed the mean time to completion along with a 95% confidence interval. The performance measurements were conducted on a six core, 3.33GHz Xeon server with 6GB of RAM running FreeBSD 9.2. Figure 9 presents the results. We compare performance with "Baseline" using a two-sided t-test on the difference in mean run time. Statistical significance was determined at the 0.05 level after a Bonferroni correction for multiple hypothesis testing within each benchmark.

First, observe that the overhead of our system for programs that are not secured by SHILL scripts is negligible. Second, the slowdown for "Sandboxed" and "SHILL version" configurations ranges from negligible to 1.21×, except for a few extreme cases: the "Sandboxed" configurations of the Download and Uninstall benchmarks and the "SHILL version" of the Find benchmark. These tasks are 1.73×, 6.61×, and 6.01× slower than the baseline, respectively. We explore these high overheads below. Third, the SHILL version of the package management benchmark has no significant overhead and the SHILL version of the grading script is only 1.13× slower, despite the finer-grained guarantees these scripts provide.

Profiling To better understand the performance of SHILL, we profiled the "SHILL version" configurations of the Grading and Find benchmarks, and the "Sandboxed" configurations of Download and Uninstall. We inserted instrumentation to measure the total execution

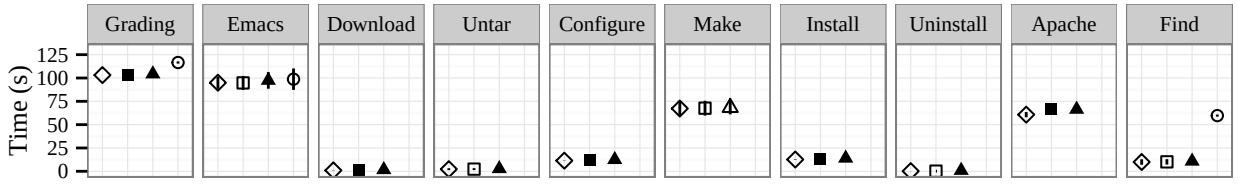

Figure 9: Performance of SHILL for a variety of tasks. Running time is given for the "Baseline" (◇), "SHILL installed" (□), "Sandboxed" (△), and "SHILL version" (◯) configurations. 95% confidence intervals are indicated by vertical bars. Bars may be hidden by plotting symbols when confidence intervals are small. Configurations that differ significantly from "Baseline" are filled (e.g., ■).

time, Racket startup (which includes script compilation, and starting the runtime), setup of sandboxes, and sandboxed execution for each benchmark. Figure 10 shows the results. Remaining time (i.e., time not spent on Racket startup, sandbox setup, or sandboxed execution) is time spent executing SHILL scripts, including contract checking. We used a Racket profiler [36] to estimate how SHILL's features affect the running time. Most time spent executing SHILL scripts is in capability-safe scripts (more than 99% for both Find and Grading) and in particular checking contracts (86% for Find and 87% for Grading). The contract on the result of pkg-native accounts for almost all contract checking time (92% and 93% of contract checking time for Find and Grading respectively) because it is checked once per sandbox. (The remaining time for the Download and Uninstall benchmarks was insufficient for the profiler to produce meaningful data.)

For these benchmarks, most time outside of sandboxed execution is spent enforcing security guarantees: checking contracts and setting up sandboxes. The Grading benchmark creates 5,371 sandboxes, Find creates 15,292, Uninstall creates one, and Download creates two (one for pkg-native and one for the executable, `curl`). Grading and Find create many sandboxes, each of which takes a relatively small amount of time to set up and a relatively small amount of time to check the contract from pkg-native. Racket startup cost is responsible for the high overhead of Download and Uninstall. The high overhead of Find is due to contract checking and sandbox setup, but also due to high sandboxed execution time. A small portion of the latter cost is due to overhead on system call interposition for privilege checking (see microbenchmarks below). We conjecture that the remaining cost stems from the high number of short-lived sandboxes that Find creates, which causes contention between threads using privilege maps and the kernel's asynchronous cleanup of expired SHILL sandbox sessions.

Microbenchmarks To understand the overhead added to system calls due to privilege checking during sandboxed execution (see Section 3.2.2), we evaluated microbenchmarks for several representative system calls under both the "SHILL installed" and "Sandboxed" configurations. The `pread`-1B microbenchmark reads one byte from an opened file; `pread`-1MB reads 1 megabyte. The `create-unlink` microbenchmark creates a new file, closes, and unlinks it. The `open-read-close` benchmarks open a file, reads one byte, and closes it. In one version of this benchmark, the path argument to `open` has length one, and in the other it has length five (i.e., the file is nested in 4 subdirectories).

We timed one million iterations of each microbenchmark, except for `pread`-1MB, which was executed one thousand times. Figure 11 shows the mean execution time and 95% confidence intervals. All differences were statistically significant. The overhead of executing system calls in a SHILL sandbox ranges between 18% (`open-read-close`, 5 lookups) and 1% (`pread`-1MB). For the `open-read-close` benchmarks, further experiments (not shown) indicate that overhead increases linearly in the length of the path (i.e., linearly with the number of lookup system calls required).

	Uninstall	Download	Grading	Find
Total time	0.82 s	1.66 s	116.38 s	61.20 s
Racket startup	0.65 s	0.63 s	0.92 s	0.65 s
Sandbox				
setup	0.01 s	0.01 s	6.98 s	18.04 s
execution	0.14 s	0.96 s	104.09 s	27.61 s
Remaining time	0.03 s	0.07 s	4.39 s	14.90 s

Figure 10: Performance breakdown of four benchmarks.

Operation	SHILL Installed	Sandboxed	Difference
`pread`-1B	516 ± 80 ns	560 ± 64 ns	44 ± 102 ns
`pread`-1MB	199 ± 4 ms	202 ± 6 ms	3 ± 7 ms
`create-unlink`	13 ± 3 ms	14 ± 4 ms	1 ± 4 ms
`open-read-close`			
1 lookup	3.7 ± 0.4 ms	4.0 ± 0.4 ms	$0.3 \pm .6$ ms
5 lookups	5.3 ± 0.3 ms	6.4 ± 0.5 ms	1.1 ± 0.6 ms

Figure 11: Overhead of SHILL for microbenchmarks.

5 Related work

Much research is devoted to controlling the authority of untrusted software and applying the Principle of Least Privilege (POLP), spanning operating system design, systems security, and programming languages.

Operating Systems Capabilities are a well-known and effective mechanism to support POLP. Capability-based operating systems [6] such as KeyKOS [11, 5], EROS [34], Coyotos [33] and PSOS [29] use operating system and hardware capabilities to limit the authority of users and processes. Numerous microkernels inspired by the L4 family [19] employ capabilities as an access control mechanism [4, 13, 18]. HiStar [44] and Asbestos [44] track information flow to enforce fine-grained security policies. SHILL is not an operating system and is built on a commodity operating system. However, it shares similar goals and draws inspiration from these novel systems. For instance, the source of certain kinds of capabilities in KeyKOS is the *command system*: the only program in the system with ambient access to a user's directory. SHILL's ambient scripts serve the same purpose.

Capsicum [43] extends the FreeBSD operating system with capabilities but requires programs to be rewritten to use the capability-based interfaces in order to make use of capability mode. By contrast, SHILL's capability-based sandbox does not require executables to be aware of capabilities. In addition, SHILL capabilities are more expressive than Capsicum capabilities; for example, a SHILL capability can express the permission to create files in a directory and delete only files that were created with the capability.

Systems security Laminar [30] integrates operating system and programming language abstractions to enforce decentralized information flow control (DIFC). Its high-level architecture resembles that of SHILL. However, Laminar provides fine-grained security only for programs that use Laminar's security abstractions, and does not provide declarative security specifications. Hails [10] uses declarative information-flow control policies as a mechanism for composing mutually distrusting web applications. Unlike SHILL, it provides limited support for securing legacy applications. Flume [17] uses a user-space reference monitor for DIFC at the granularity of operating system abstractions. While both SHILL and Flume can enforce security restrictions on untrusted applications, SHILL uses capabilities and contracts rather than DIFC labels.

A plethora of sandboxing tools have been developed for commodity operating systems, including SELinux [20], Seatbelt [42], AppArmor [1], GrSecurity [35], LXC [3], and Docker [2]. Unlike SHILL, these sandboxes deny or grant access based on a profile rather than a programmable capability-based interface. Mbox [15] and TxBOX [14] create sandboxes with transactional semantics that can reverse the effects of misbehaving processes, but enforce strong isolation between sandboxed processes and the rest of the system. Notably, programs running in a SHILL sandbox are not isolated from the rest of the system. For example, in our Apache case study, concurrently executing programs can dynamically add new web content or view logs as they are generated. Many of these sandboxes require root privileges, but some are available to all users [15]. PLASH [32] is a capability-based interactive shell for creating sandboxes in which to execute shell commands, similar to SHILL's exec. All of these tools lack the reasoning principles SHILL provides for composing multiple sandboxes together.

Programming languages The use of language-level capabilities to support POLP has a long history [28]. The E programming language [26] is a seminal *object capability language*, where capabilities are object references. CapDesk [40, 37] is a desktop shell for launching applications written in E. Applications are granted limited authority initially and can gain more capabilities through *powerboxes*, which mediate requests for authority from the application to the user. In contrast to SHILL, CapDesk does not have a scripting interface and applications launched by CapDesk must be capability-aware and designed to work with the CapDesk framework.

Joe-E [22] restricts Java to an object-capability-safe subset. Similarly, Caja [27] introduces an object-capability-safe subset of JavaScript. Maffeis et al. [21] prove that these subsets are indeed capability safe. Unlike other capability-safe languages, SHILL targets a particular domain (shell scripting) instead of general programming and that it uses contracts to manage capabilities instead of capability-based design patterns [26].

Acknowledgments

We thank Dan Bradley for his contributions to an early version of this work, and Jennifer Kirk for her help with statistical analysis. We are grateful to Leif Andersen, Vincent St-Amour, and Matthias Felleisen for their help profiling SHILL code. We thank Eddie Kohler, the Programming Languages Group at Harvard, and the reviewers for their helpful comments. Many thanks to Frans Kaashoek for his thoughtful shepherding. This research is supported by the Air Force Research Laboratory.

References

[1] Apparmor. https://wiki.ubuntu.com/AppArmor.

[2] Docker. https://www.docker.io.

[3] LXC. https://linuxcontainers.org.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 29–44, 2009.

[5] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*. USENIX Association, 1992.

[6] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9 (3):143–155, 1966.

[7] R. B. Findler and M. Blume. Contracts as pairs of projections. In *Proceedings of the 8th International Symposium on Functional and Logic Programming*, pages 226–241, 2006.

[8] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the International Conference on Programming*, pages 48–59, 2002.

[9] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. http://racket-lang.org/tr1/.

[10] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–60. USENIX, 2012.

[11] N. Hardy. KeyKOS architecture. *Operating Systems Review*, 19 (4):8–25, 1985.

[12] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[13] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[14] S. Jana, D. E. Porter, and V. Shmatikov. TxBox: Building Secure, Efficient Sandboxes with System Transactions. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, May 2011.

[15] T. Kim and N. Zeldovich. Practical and effective sandboxing for non-root users. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, pages 139–144, Berkeley, CA, USA, 2013. USENIX Association.

[16] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek, E. Kohler, D. Mazières, R. Morris, M. Osborne, S. VanDeBogart, and D. Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Conference on Hot Topics in Operating Systems*, page 21, Berkeley, CA, USA, 2005. USENIX Association.

[17] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 321–334, 2007.

[18] A. Lackorzynski and A. Warg. Taming subsystems: Capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, pages 25–30, 2009.

[19] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 237–250, 1995.

[20] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[21] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *IEEE Symposium on Security and Privacy*, pages 125–140, May 2010.

[22] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2010.

[23] B. Meyer. Design by contract. In *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, 1991.

[24] B. Meyer. Applying "Design by Contract". *Computer*, 25(10): 40–51, 1992.

[25] M. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University, 2003.

[26] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[27] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. Google white paper. http://google-caja.googlecode.com, 2008.

[28] J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, January 1973.

[29] P. G. Neumann and R. J. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 208–216, Dec 2003.

[30] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 63–74, 2009.

[31] J. H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974. ISSN 0001-0782.

[32] M. Seaborn. PLASH: the principle of least authority shell, 2007. http://www.cs.jhu.edu/~seaborn/plash/html/.

[33] J. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In *Proceedings of the NICTA Invitational Workshop on Operating System Verification*, pages 1–19. USENIX, 2004.

[34] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 170–185, 1999.

[35] B. Spengler. Grsecurity ACL docmentation v1.5, 2003. http://grsecurity.net/gracldoc.htm.

[36] V. St-Amour and M. Felleisen. Feature-specific profiling. Technical Report NU-CCIS-8-28-14-1, Northeastern University, August 2014.

[37] M. Stiegler and M. Miller. A capability based client: The DarpaBrowser. Technical Report BAA-00-06-SNK, COMBEX Inc., June 2002.

[38] T. S. Strickland, S. Tobin-Hochstadt, R. Findler, and M. Flatt. Chaperones and impersonators. In *Proceedings of the ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 943–962, 2012.

[39] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 132–141, 2011.

[40] D. Wagner and D. Tribble. A security analysis of the Combex DarpaBrowser architecture. Online at: http://www.combex.com/papers/darpa-review/, Mar. 2002.

[41] R. Watson and C. Vance. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *In USENIX Annual Technical Conference*, pages 285–296, 2003.

[42] R. N. M. Watson. A decade of OS access-control extensibility. *Communications of the ACM*, 56(2):52–63, 2013.

[43] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX Security Symposium*, pages 29–46. USENIX Association, 2010.

[44] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19, 2006.