Trellis: Privilege Separation for Multi-User Applications Made Easy

Andrea Mambretti[1], Kaan Onarlioglu[1], Collin Mulliner[1], William Robertson[1], Engin Kirda[1], Federico Maggi[2], and Stefano Zanero[2]

[1] Northeastern University, Boston, USA
{mbr,onarliog,wkr,ek}@ccs.neu.edu,
collin@mulliner.org,
[2] Politecnico di Milano, Milano, Italy
{federico.maggi,stefano.zanero}@polimi.it

Abstract. Operating systems provide a wide variety of resource isolation and access control mechanisms, ranging from traditional user-based security models to fine-grained permission systems as found in modern mobile operating systems. However, comparatively little assistance is available for defining and enforcing access control policies within multi-user applications. These applications, often found in enterprise environments, allow multiple users to operate at different privilege levels in terms of exercising application functionality and accessing data. Developers of such applications bear a heavy burden in ensuring that security policies over code and data in this setting are properly expressed and enforced. We present Trellis, an approach for expressing hierarchical access control policies in applications and enforcing these policies during execution. The approach enhances the development toolchain to allow programmers to partially annotate code and data with simple privilege level tags, and uses a static analysis to infer suitable tags for the entire application. At runtime, policies are extracted from the resulting binaries and are enforced by a modified operating system kernel. Our evaluation demonstrates that this approach effectively supports the development of secure multi-user applications with modest runtime performance overhead.

1 Introduction

Operating systems provide a wide range of resource isolation and access control mechanisms to realize well-established computer security principles such as privilege separation and assignment of minimal privileges to users and tasks. For instance, UNIX-like systems employ the traditional access control model based on user and group identifiers assigned to resources, Linux uses a *capability* system for fine-grained process permission management, and modern mobile operating systems such as Android and iOS allow users to control application permissions during installation or runtime.

Although these techniques are effective at isolating users of a multi-user system from each other, or controlling access to operating system-owned resources

such as hardware devices and the filesystem, they do not address the problem of enforcing access control *within* an application itself. In particular, many complex programs that target enterprise markets (e.g., SAP CRM) support multiple user roles such as administrators and other unprivileged users, each allowed different access levels to sensitive application data.

Due to this lack of standard operating system support for developing applications with multiple privilege levels, the responsibility of implementing an application-specific access control model is relegated to application programmers. However, this can often result in implementation errors, or incorrect use of various application components as access control primitives, exposing the application to privilege escalation attacks. Recent work by Mulliner et al. [21] demonstrates this problem by showing that many enterprise applications rely on selectively hiding GUI elements to naïvely control the access to the respective, sensitive functionalities. This inadequate enforcement scheme was easily be subverted by the authors using existing GUI inspection tools, allowing them to access administrator-only features with an unprivileged user account.

There exists substantial prior work that has explored ways to separate applications into privileged and unprivileged components to contain privilege escalation attacks. For instance, Provos et al. [24] described a methodology to manually split programs into a privileged *monitor* and an unprivileged *slave* that communicate via IPC channels, and Kilpatrick [16] proposed a software library to ease the development of such applications. Brumley and Song [10] use source code annotation and static analysis, and Wu et al. [32] use dynamic analysis to automate parts of this process. However, this work primarily targets system services, and aims to minimize the code run with superuser privileges. They do not address the problem of building applications that support strong separation of multiple users, each with specific code and data access requirements.

In this paper, we introduce an approach called Trellis to develop and enforce secure hierarchical access control models within multi-user applications. Trellis provides a simple annotation mechanism for software developers to specify the required access levels for critical functions and static data, uses static analysis to derive an access control policy for the entire application, and compiles this source code into binary executables that are strongly protected by the operating system. At runtime, the operating system tracks the program control flow and data accesses, including dynamically allocated data, and enforces the statically-derived access control policy. Our prototype implementation implements code and data privilege separation using dynamic adjustment of memory segment permissions. Trellis does not require drastic modifications to the application architecture, making it easy to apply to existing source code. It also does not split applications into multiple components and thus, in contrast to previous work, does not incur IPC overhead during runtime.

To summarize, we make the following contributions in this paper.

- We propose Trellis, a novel operating system-supported application development framework to assist software authors with the development of hierarchical access control policies for multi-user applications. Trellis uses a

combination of source code annotation, static analysis, and dynamic analysis to automatically integrate these policies into applications, and to enforce them at the operating system level.
- We present a prototype implementation of our system based on LLVM/-Clang, the GNU C library, and the Linux kernel.
- Our evaluation including micro-benchmarks, and experiments on real-world applications demonstrate that Trellis imposes a low runtime performance overhead, an acceptable tradeoff for its additional security guarantees.

2 Threat Model

The environment we consider for this work is a large, multi-user application that runs on a shared machine. Typical examples of this scenario include kiosk applications, or large enterprise applications (e.g., SAP CRM) with several users (e.g., employees) each having a distinct profile. These users can use the application at different moments in time and each user, depending on her own profile, can access different subsets of the application's functionality and data.

In this setting, the attacker has local or remote access to a computer running such a multi-user application. The attacker further has access to a user account on the application. Note that the attacker could already be an ordinary user of the application; in other words, she may be authorized to legitimately access a subset of the application's features. Alternatively, the attacker could compromise a different user's account in order to gain unauthorized access to the application.

Our threat model includes two attack scenarios. In the first scenario, the attacker's goal is to exercise application features (i.e., execute code) reserved for higher-privileged application users. Likewise, in the second scenario, the attacker aims to gain access to data associated with a different, higher-privileged application user. Both attacks are made possible due to the fact that resources associated with different user accounts are managed in the same address space within the application.

Note that we assume sensitive code and data that could be targeted by an attacker is already protected by traditional operating system protections, and therefore, sensitive disk or memory contents cannot be accessed by the attacker directly. Instead, successfully carrying out one of the described attacks requires the attacker to exploit an application-level vulnerability, and bypass privilege-separation mechanisms provided by the application in question.

The trusted computing base (TCB) we assume for this work includes the software development toolchain (i.e., the compiler and linker), which ensures that an adversary cannot subvert the access control policies specified by developers. The TCB also includes the hardware and software stack up to and including the operating system kernel. This implies that adversaries cannot subvert the enforcement of the developer-specified policies, or tamper with the authentication procedure to transition between privilege levels. Finally, we assume that the adversary cannot tamper with the binary itself, which contains a machine-readable specification of the intended access control policy, nor with the loading of this

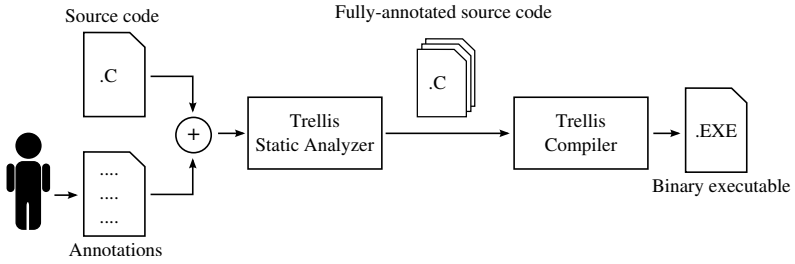

Fig. 1. An overview of compiling applications with Trellis.

specification into the kernel. Achieving these guarantees has several solutions, such as relying upon normal operating system-enforced user access control or, alternatively, using trusted computing primitives. The exact mechanism used is considered outside the scope of this work.

3 Design

Trellis spans both the development toolchain and the runtime environment. First, it provides a source code annotation mechanism for software developers to specify the different privilege levels (i.e., roles) required for effective access control in their applications, as well as compilation tools and system libraries capable of building Trellis-aware binaries. Second, it enhances the operating system kernel to monitor execution flows and authorize transitions between privilege levels consistent with specified access control policies.

At a high level, running a Trellis-enhanced application is a two stage process. First, an instrumented binary executable must be built according to the annotated source code. Then, the executable must be loaded with its initial privilege state and run.

An overview of the first stage, the compilation of binary executables, is shown in Figure 1. The application developer first (partially) annotates the program source code, which involves marking security sensitive code and data with their corresponding privilege levels (often, user roles) required to access them. The partially annotated source code is then inspected by the static analyzer component, which explores the program function call graph and automatically infers privilege level tags for all unmarked code and static data. Finally, the fully-annotated source code is compiled by an augmented tool chain, which instruments the binary according to the specified access control policy and creates an executable file including the necessary Trellis metadata.

The executable can then be loaded and launched. An overview of this process is shown in Figure 2. Here, Trellis uses an enhanced binary loader that first reads the access control policy metadata stored inside the binary, and communicates this information to the operating system to initialize the application. Once launched, the operating system monitors the application and its memory

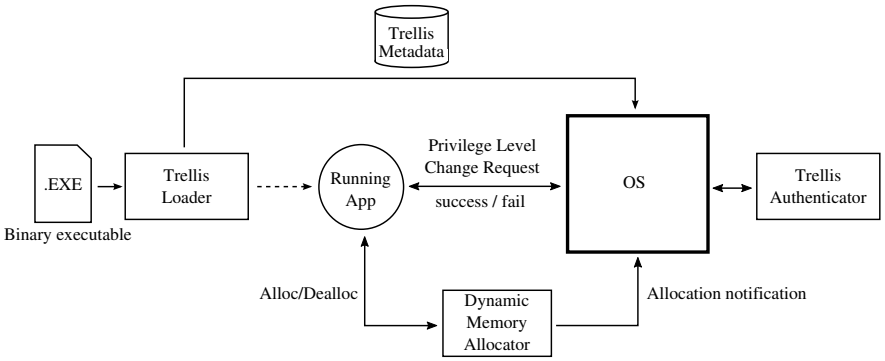

Fig. 2. An overview of running applications with Trellis.

accesses to enforce the implemented access control policy. In the case of dynamic memory allocation requests by the application, a modified system memory allocator notifies the operating system of the performed operation. In this way, access control checks can be applied to those memory regions as well. Of course, during runtime, the application might need to change the active privilege level to a more privileged one. In that case, the application issues a privilege level change request to the operating system, and the operating system in turn consults an authentication module to check whether the request should be served. If permission is granted, the system performs the change.

3.1 Access Control Model

The design of Trellis permits a flexible access control model, where developers can tag units of code (e.g., functions) with a set of privilege levels $l \in L$. A partial ordering is defined on L such that the usual notions of reflexivity, antisymmetry, and transitivity are satisfied. More formally, given

$$\begin{aligned} l \in L &\text{ a set of partially-ordered privilege levels,} \\ (C, l) \in \mathbb{C} &\text{ a set of units of tagged program code,} \\ (d, l) \in D &\text{ a set of tagged data,} \\ D \in \mathbb{D} &\text{ a set of all program data,} \end{aligned}$$

we define a program state as

$$S = \langle \mathbb{C}, \mathbb{D} \rangle$$

which describes the currently-executing unit of code at level l and the set of data upon which it can (potentially) compute. In the following, we refer to "units of code" as functions, although this need not be the level of granularity implemented in practice.

The developer is responsible for annotating functions with a privilege level according to application-specific requirements. These annotations, however, do

not need to be completely specified for each function. Instead, Trellis permits developers to *partially* specify privilege levels, and an inference procedure will propagate privilege levels according to the program call graph. More precisely, given a call relation $(\rightsquigarrow) : C \mapsto C$, the inference process will assign an unannotated function $(C', \cdot)$ the infimum, or greatest lower bound, of all of its callers:

$$(C', l') \text{ s.t. } l' = \inf \bigcup_i (C_i, l_i), C_i \rightsquigarrow C' \forall C_i. \tag{1}$$

Note that the inference procedure might require an iterative fixpoint computation to establish privilege levels for all unannotated functions.

Given an annotated program, we can then define a transition relation

$$(\Rightarrow) : S \times S$$

that specifies *(i)* how transitions between privilege levels can occur during execution, and *(ii)* how data is tagged at runtime.

First, for function invocation, we have:
$\mathsf{invoke} : \mathbb{C} \mapsto \{\mathbb{C} \ \cup \emptyset\}$

$$\begin{aligned} ((C_i, l_i), D_i) \Rightarrow ((C_j, l_j), D_i) \ \text{ where} \\ C_i \rightsquigarrow C_j, \\ (l_j \leq l_i) \vee (\mathsf{auth}\,((C_i, l_i), (C_j, l_j))) \end{aligned}$$

Here, $\mathsf{auth} : \mathbb{C} \times \mathbb{C} \mapsto \{\mathsf{t}, \mathsf{f}\}$ is an authentication predicate that is left as an open parameter of the implementation. The only requirement is that a binary decision is returned to either permit or deny an invocation between functions at given privilege levels. auth is only invoked if transitioning to a higher privilege level; otherwise, the transition is implicitly allowed to occur.

Another important operation we formally specify is data allocation:
$\mathsf{alloc} : \mathbb{C} \mapsto \mathbb{D}$

$$\begin{aligned} ((C_i, l_i), D_i) \Rightarrow ((C_i, l_i), D_j) \ \text{ where} \\ \mathsf{alloc}\,((C_i, l_i)) = (d, l_i), \\ (d, l_i) \in D_j \end{aligned}$$

For this transition rule, we further distinguish between statically-allocated data, stack-allocated data, and heap-allocated data. For statically-allocated data, the developer either provides a privilege level annotation for the system, or one is automatically inferred. Stack-allocated data, on the other hand, is always tagged with the privilege level for the associated function. Finally, heap-allocated data is tagged with the level of the enclosing function of the allocation site.

Finally, we specify the notion of data access, which subsumes both data reads and writes:

$$\mathsf{access} : \mathbb{C} \times D \mapsto \{t, f\}$$

$$((C_i, l_i), D_i) \Rightarrow ((C_i, l_i), D_j) \text{ where}$$
$$\mathsf{access}\,((C_i, l_i), (d_j, l_j)) \equiv l_j \leq l_i,$$
$$(d_j, l_j) \in D_j$$

Here, we simply state that reads or writes of data must only be permitted when the current privilege level is greater than or equal to the data's tag.

4 Implementation

In this section, we present a proof-of-concept implementation of Trellis in detail for Linux. We begin by describing the compile-time annotation and tag propagation procedure, and then discuss the implementation of runtime policy enforcement. Note that while the formal model we describe in Section 3.1 defines a partially-ordered set of privileges, our prototype implementation assumes a strict hierarchy of privilege levels for simplicity.

4.1 Compile-Time Component

The compile-time component of Trellis is implemented as a series of transformation passes for the LLVM/Clang compiler suite. As input, this toolchain takes a program that has been partially annotated by the application developers.

Privilege Level Annotations Privilege level annotations are applied using a custom attribute that allows developers to express the minimum required privilege level to execute a function or access a static variable. At compile time, the new attribute is processed by the Clang compiler front-end, which supports custom attributes by forwarding them to subsequent Clang or LLVM transformation passes alongside the associated function or global variable identifier.

Since we need to keep track of the attribute parameter (e.g., 2 or 6 in Figure 3's example), we modified Clang to forward the parameter value to LLVM. The attribute is considered valid for both functions and global variable declarations and specifies the privilege level of functions and global variables. For instance, Figure 3 exemplifies a function `fun` of privilege level 2 and a global variable `dat` at privilege level 6. In this example, the developer wants to prevent function `fun` from accessing the memory area where the variable `dat` will be stored at runtime.

The attribute value indicates the privilege level of the associated object, which ranges from 0 and a tunable constant `NUM_LEVELS`, where higher levels indicate higher privileges. The `main` function is automatically set to 0, the lowest privileve level, by Trellis. If the developer tries to use a level greater than `NUM_LEVELS`, she receives a compile-time error.

Therefore, the prototype implements a simple access control variant of our proposed model, where privilege levels exist in a strict, linear hierarchy. More

```
1 void fun() __attribute__((trellis_level(2)));
2 int  dat   __attribute__((trellis_level(6)));
```

Fig. 3. Example of function and global variable annotations using custom attributes.

flexible variants that can model an arbitrary lattice of privilege levels are possible; however, we consider their implementation an engineering exercise.

Tag Inference The first transformation pass implements privilege level tag inference. It analyzes attribute values specified by the developers and propagates them to non-tagged functions and data. Clearly, the developer could tag every function and global variable. However, tag inference improves the ergonomics of the system by allowing for a partial specification to be automatically extended to cover the entire program. Developers can inspect the output of the transformation to identify potential errors in the final policy, or modify it as necessary.

The pass first computes a queue of all annotated functions. Then, operating in a breadth-first fashion, the pass visits callees of the current function. If the callee has been manually annotated by the developer, then its tag is considered immutable and is not changed. If the callee is already tagged, a level is assigned that is the minimum of the new and existing tags (as dictated by Equation 1). Otherwise, the caller's privilege level is used to tag the callee. In either of the preceding two cases where the callee's tag has been modified, the callee is added to the queue of functions to visit. The pass continues processing the function queue until a fixpoint is reached (i.e., the queue is empty). The search is guaranteed to terminate because privilege levels never increase and there exists a global minimum privilege level.

Transition Instrumentation and Error Handling Trellis protects code and data tagged with a privilege level that is higher than the current level. However, the application should be able to change levels at runtime. To do so, Trellis leverages a new system call to inform the Linux kernel that the application requests a level transition. This system call, `trellis_switch`, is automatically injected by another transformation pass at every call site of the program when the caller has a lower privilege level than the callee. After the call site, another invocation of `trellis_switch`, is injected to inform the kernel that the application should return to the caller's privilege level. In particular, whenever this transformation pass encounters such a call site, it inserts a code snippet as exemplified in Figure 4.

In case of failure because the current user is not authorized for the requested level, a wrapper function, `trellis_exit_wrapper`, is called. Through the wrapper, Trellis allows the developer to specify her own custom failure handling, where she can for instance implement recovery from the failure. If nothing is specified by the developer, the system by default will invoke `exit` to safely terminate the application.

```
1 if (trellis_switch(x) != 1) {
2     trellis_exit_wrapper();
3 }
4
5 call();
6 trellis_switch(y);
```

Fig. 4. Example of privilege level transition instrumentation and authentication error handler invocation.

Code and Data Reordering The final transformation involves the reordering of all functions and static data. Every unit of code and static data is, at this point, tagged with a certain level. With a simple LLVM pass, each function and global variable is grouped by privilege level. Each group is then moved to a pair of separate sections of the binary, one for code and the other for data. The section names are created by concatenating the data type and level. For instance, in the case of code, the name will follow the pattern `fun_trellis_l` for every function that is tagged with level l. An analogous name `dat_trellis_l` is created for data tagged with level l.

The pass records all the levels used by the program under analysis and generates a custom linker script. The linker script is used to map each of the sections above to a unique program segment. The script also aligns the start address of each segment to a machine page boundary to avoid loader redefinition.

4.2 Run-Time Component

At this point, the binary has been produced and is ready to be run. The next subsections explain the runtime execution phase in detail.

Policy Loading During normal program execution, the dynamic loader is responsible for several tasks that include mapping any shared libraries as well as mapping program segments from an executable image into memory. Our prototype contains a modified loader that parses the privilege level for each Trellis segment added at compile time, and communicates this information to the kernel with a special system call added for this purpose: `trellis_init`.

`trellis_init` copies the program's memory layout from userspace to kernel space, and attaches a list of memory boundaries (see Figure 5) to the `task_struct` of the application. The `task_struct` is the canonical process descriptor for the running application inside the kernel, and contains all information regarding the process (e.g., credentials, memory maps).

`trellis_init` is executed during dynamic loading before control is passed to the application. Trellis allows this system call to be invoked only once for each process. This prevents an attacker from using a second invocation of this system call during execution to elevate privileges by relaxing the intended access control policy. For the same reason, after the execution of `trellis_init`, `mprotect` cannot be invoked by the process.

```
struct trellis_dyn_t{
    int priv_level;
    int size;
    void *mem;
    struct trellis_dyn_t *next;
};
```

Fig. 5. Memory chunk information element.

After dynamic loading has completed and `main` is ready to be invoked, the process is in a state where *(i)* only code and data at the lowest privilege level is accessible, and *(ii)* all other (higher-privileged) segments are *not* readable, writable, or executable.

Privilege Level Transitions The second system call added to the Linux kernel is `trellis_switch`. It allows an application to request a transition from the current privilege level to another, specified using the parameter `new_level`.

When transitioning to a higher level, the kernel wakes a daemon that blocks the resumption of the requesting application until authentication has completed; this allows the use of interactive authentication if desired. In the meantime, the process is moved by the operating system into the wait state and will be woken only at the end of the `trellis_switch` system call. If the authentication succeeds, the kernel changes the permissions of the code and data segments for the requested level and returns control to the application. Otherwise, the wrapper for authentication failure is invoked. When transitioning to an equal or lower level, authentication is not required, and therefore the inverse of the above segment re-permissioning procedure is performed automatically. An example of the dynamic memory segment permission process is shown in Figure 6.

Dynamic Data Tagging The third system call is `trellis_tracemalloc`, which manages dynamic memory allocations. Typically, applications allocate memory at runtime using standard functions from the `malloc` family, and release it using `free`. These functions are merely the interface to a heap allocator, and underlying this application-level interface are system calls such as `brk` and `mmap` that are used to request additional memory from the operating system.

This presents two main challenges for our protection mechanism. The first challenge is that it is not straightforward to assign privilege levels to heap-allocated data due to the additional indirection imposed by the heap allocator. That is, the page-level permission scheme used for code and static global data does not map cleanly into the variable-sized chunk allocation interface exposed by the heap allocator. The second challenge is that chunk metadata is stored inline with application data. This implies that page-level permissions would potentially restrict access not only to the data but also the chunk metadata that is used by the allocator.

To overcome these challenges, Trellis introduces a multi-heap allocator using `trellis_tracemalloc` that effectively partitions dynamic memory allocation ac-

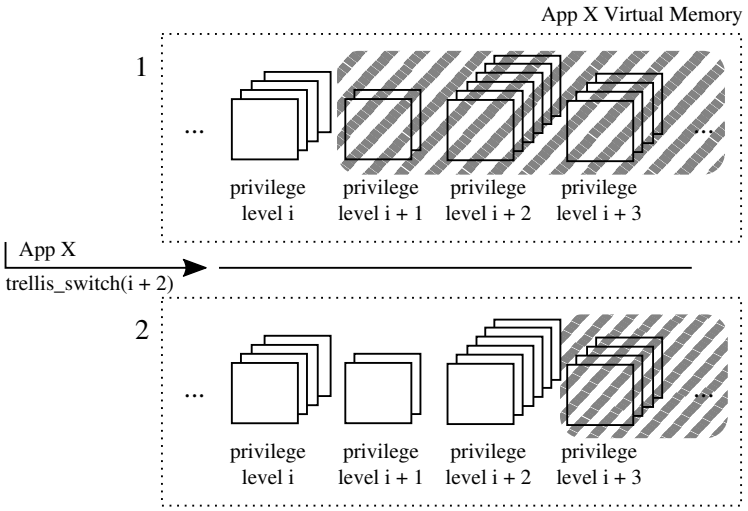

Fig. 6. Example of a dynamic memory segment permission update to transition between permission levels – in this case, from level i to $i + 2$. Shaded segments indicate inaccessible code and data regions.

cording to privilege levels. This allocator exposes two functions, `trellis_malloc` and `trellis_free`, to allocate and release memory, respectively. This allocator also maintains chunk metadata in a separate area of memory by tracking different lists of pages, where every list of pages corresponds to a separate heap. These structures are shown in Figure 7. Other standard allocation routines are implemented in terms of these two basic primitives. The compiler toolchain can optionally replace uses of the traditional allocation functions in a transparent fashion, or the developer can be responsible for doing so.

Authentication and Authorization Transition authorization is implemented by a userspace daemon, a kernel netlink interface, and `trellis_switch`. Before the application is executed, the netlink interface is activated through a kernel module and the daemon starts. This can be performed either manually or automatically when needed. When the application requests a privilege level transition, `trellis_switch` writes the requested privilege level into the netlink channel where the daemon receives it. The daemon then performs the authentication check and transmits a message over the netlink channel to notify the kernel of an authentication success or failure. The netlink communication channel is secure according to the attacker model defined in Section 2.

To implement authentication, the prototype simply forwards the request to the standard Pluggable Authentication Module (PAM) framework. PAM is modular, versatile, supports a variety of authentication mechanisms (e.g., passwords, smart cards, fingerprints), and is well-tested and widely used.

```
void *heaps[NUM_LEVELS] = { NULL, };

/* Memory chunk */
struct trellis_chunk {
    size_t size;
    struct trellis_chunk *next;
    struct trellis_chunk *prev;
    void *ptr;
};

/* Page struct with list of empty and used chunks */
struct trellis_page {
    struct trellis_chunk *free;
    struct trellis_chunk *used;
    struct trellis_page *next;
    struct trellis_page *prev;
    size_t size;
};
```

Fig. 7. Chunk and page list structures for the multi-heap dynamic memory allocator.

5 Evaluation

In this section, we first describe our experiments to measure the performance overhead introduced when compiling, loading and running applications with Trellis. Specifically, we present *micro-benchmarks* to characterize the cost of the newly introduced Trellis operations, and *end-to-end* performance tests reflecting the overhead of running complete applications with Trellis. Next, we present an empirical evaluation of the developer effort required to adapt an application to work with Trellis, and test the system's security.

All experiments were performed on a machine with an Intel i7-3520M CPU and 4 GB of memory, running Gentoo Linux with a Trellis-patched kernel version 3.9.11 and glibc 2.19. The test binaries were built using LLVM/Clang version 3.5. The results for non-Trellis experiments were obtained on an identical setup running the vanilla versions of the Linux kernel and glibc.

5.1 Micro-Benchmarks

Privilege Level Change. In this experiment, we measured the time required to change the privilege level of a running application. We created a benchmark program, and performed measurements at the entry and return points of the function that requests the privilege change. That is, our measurement includes the switch from userspace to kernel space, the call to the corresponding Trellis system call and all subsequent operations, and the switch back to userspace. During this experiment, the authentication module was configured to automatically allow all privilege change requests in order to avoid any human interaction overhead. Note that since a system without Trellis does not contain a corresponding operation, we cannot obtain any baseline to compare these results with. Therefore, here we only report the absolute runtime cost of the tested operation.

Dynamic Memory Allocation. We designed this experiment to characterize the overhead of Trellis's modified `malloc` operation. We created a simple

benchmark program that takes measurements before and after a call to `malloc` to compute the elapsed time, and repeated the experiment with the standard, unmodified glibc memory allocator for comparison.

In order to see the effects of allocated chunk size on performance, we repeated this experiment with varying allocation sizes of 1 KB, 2 KB, 4 KB, 1 MB, and 100 MB. We did not observe a significant correlation between chunk size and runtime, and Trellis's overhead remained nearly constant in all experiments. Therefore, we only report the worst-case performance we observed in this section.

Executable Loading. In this experiment, we measured the overhead incurred by the dynamic loader for reading the Trellis metadata associated with a binary executable, and setting up the initial active privilege level inside the kernel prior to launching the application. To this end, we instrumented the executable loader with timing functions, and experimented with launching a test binary executable both with and without Trellis support.

5.2 End-to-end Performance

Because we did not have source code access to an actual commercial multi-user application that would benefit from Trellis's features, we instead opted to follow a different evaluation strategy. First, we ran experiments on an application developed in-house to test Trellis with, which we will call *StoreManager*. Next, we took an existing open-source application, *HomeBank* [2], extended it with multi-user capabilities, and adapted it to work with Trellis.

StoreManager is a store inventory management software that supports three distinct user roles. "Unprivileged users" are ordinary employees that can browse and view details of the items registered in the database, or view aggregate reports about the inventory status. "Managers" have additional privileges to manage the inventory, such as adding, removing, or editing the details of items. Finally, "system administrators" hold the highest level of privileges, and are able to create or delete user accounts on the system, or directly manipulate the inventory.

HomeBank is a popular accounting software that provides account management, analysis, and reporting features. While HomeBank is originally designed for personal use, we extended it support four different user roles. Prior to authentication, the application runs under an "unprivileged user" role, with access to only the basic features. "Analysts" are only able to access analysis and reporting functionalities, but cannot add or modify accounts. "Accountants" have the additional privileges to schedule new transactions on existing accounts. Finally, "managers" hold the highest privileges and are able to access critical features including creating and modifying accounts, transactions, and budgets.

We created two instances of each application. One was built to run on an ordinary Linux system, and access control was enforced by disabling access to the GUI elements that perform privileged operations inside the application. The other's source code was manually annotated by the authors according to the aforementioned access control policy, and built to run with Trellis. All of the following experiments were performed using these applications.

Experiments	Baseline	Trellis	Overhead
Privilege Level Change	-	159.91 µs	-
Dynamic Memory Allocation	34.04 µs	57.27 µs	68.24 %
Executable Loading	108.44 µs	136.45 µs	25.83 %
StoreManager Compilation Time	850.17 ms	933.29 ms	9.78 %
HomeBank Compilation Time	28.62 s	28.72 s	0.36 %
StoreManager Runtime	14.75 s	15.20 s	3.05 %
HomeBank Runtime	14.37 s	14.94 s	4.02 %

Table 1. Trellis performance evaluation results. See Section 5 for detailed explanations.

Compilation Time. In this test, we measured the overhead incurred for Trellis-specific code analysis, annotation propagation, and source code instrumentation performed during compilation and linking of the binary executables. We first compiled the two test applications using the unmodified toolchain, and then with Trellis to compare the results.

Application Performance. As the final part of our evaluation plan, we designed a comprehensive use-case scenario for each of the two test applications that exercises all program features using different user profiles, and measured the end-to-end runtime. For this task, we used the GUI automation tool Linux Desktop Testing Project (LDTP) [3], and configured the Trellis authentication module to automatically allow privilege level change requests—as we are not interested in timing the user interaction. For the StoreManager experiment, our use-case involved viewing numerous inventory items as an ordinary user, creating and deleting items as a store manager, and manipulating user accounts as a system administrator. The HomeBank use-case involved setting up new accounts and budgets as a manager, scheduling transactions as an accountant, and reviewing reports as an analyst. We ran these use-cases with and without Trellis enabled, and computed the overall runtime overhead of our system.

5.3 Experiment Results

We performed the above micro-benchmarks 1000 times, compilation time measurements 50 times, and application runtime measurements 300 times. The average runtimes over all runs are presented in Table 1.

The compilation time experiment shows that building a Trellis-aware executable for StoreManager takes about 10% longer than compiling an unprotected program. Although this overhead could suggest discernibly longer compilation times for complex applications, we note that this is a one-time performance trade-off for the significant access control enforcement benefits provided by Trellis. Also note that, compared to StoreManager, the compilation time overhead for HomeBank is much lower at only 0.36 %. This is because HomeBank's codebase

is considerably larger than StoreManager's, which leads to its normal, lengthy compilation time over-shadowing the overhead imposed by Trellis. Similarly, despite the seemingly large executable loading overhead of about 26%, we stress that the absolute application launch time difference is only on the order of micro seconds, and is unlikely to be noticed by users.

For the application runtime measurements, even though we do not have a baseline to compare the privilege level change performance against, we expect the overhead to be acceptable since privilege level changes are not expected to be common operations during runtime. Moreover, they are likely to be completely over-shadowed by human response times if an interactive authentication scheme is used. The Trellis dynamic memory allocator, however, is shown to perform significantly worse than the default glibc allocator. This could potentially lead to performance drops in applications that perform heavy dynamic memory allocation in small chunks, and could require optimization techniques such as using pre-allocated memory caches. Despite this potential shortcoming, our final performance tests show that the runtime impact on real-world applications, with typical use-cases, is only around 4%.

5.4 Developer Effort

Unfortunately, systematically quantifying the developer effort required to adapt an existing application to work with Trellis is a non-trivial task, requiring a large-scale study with an extensive corpus of Trellis-enabled software. Therefore, we provide anecdotal results obtained during our modifications to HomeBank.

The development of our multi-user version of HomeBank was carried out by a single developer, one of the authors of this paper. While the developer had over ten years of C programming experience, he had no prior experience with the HomeBank application's codebase. We measured the time taken during all phases of active development, and report the results in the following.

Surprisingly, significant effort before development went into adapting HomeBank's compilation chain (i.e., autoconf scripts, Makefiles, etc.) that is designed around using GCC, to Trellis's LLVM/Clang environment. We spent around 2 hours to be able to compile HomeBank with our system. Next, 4 hours were taken to understand the codebase, and identify the relevant points that should be modified to achieve the desired privilege separation. Finally, all Trellis annotations and changes were applied and tested in another 2 hours. Overall, with a single developer, and without prior familiarity with the application, HomeBank could be adapted to work with Trellis in under 8 hours, which we believe to be a reasonable and acceptable degree of developer effort.

5.5 Security Experiments

Trellis' security properties hold by definition, and there are no heuristics for attack detection or any probabilistic decisions involved in the system. In order to empirically verify the effectiveness of Trellis against concrete, practical attack scenarios, we created a set of exploits following the methodology laid out in [21].

Specifically, in [21], the authors define a novel class of access control vulnerabilities called GUI Element Misuse (GEM), which involves bypassing an application's built-in access control checks through manipulation of its GUI elements (e.g., by un-hiding a hidden button that allows access to privileged functionality). GEM vulnerabilities exemplify a recent, high-impact instance of the type of attacks that Trellis aims to address. Unfortunately, we were not able to use the same set of applications that were exploited in [21] due to them being closed-source Windows applications. Thus, we opted to perform our experiments on StoreManager, and our extended multi-user version of HomeBank.

Following the steps outlined in [21], we first analyzed the test applications using a GUI explorer tool *Parasite* [4]), and identified buttons that would trigger privileged functionality. Using the same tool, we then attempted to forcefully enable and interact with these GUI elements with an unprivileged user account, using both the vanilla and Trellis-protected versions of the two test applications.

The vanilla versions of the programs were vulnerable to GEM attacks as we expected, and we were able to execute privileged operations as an unprivileged user. On the contrary, Trellis-enabled versions were protected against our attacks; Trellis blocked our access attempts to privileged code pages, and simply rolled back the applications to a default state that we defined in the corresponding `trellis_exit_wrapper` routines.

6 Discussion & Limitations

A prerequisite for using Trellis is access to the application's source code. As we have discussed in Section 7, virtually all previous work also has not explored partitioning of binary executables directly. Given that many commercial multi-user applications that would be suitable targets for privilege-level partitioning are provided on a closed-source model, privilege separation on binary code appears to be a challenging, yet promising future research direction. However, note that Trellis aims to enforce privilege separation and access control over application-specific functionality and data, and therefore, annotating or recompiling third-party libraries used by the target application is not necessary.

Although the Trellis implementation we present in this paper is for applications written in C and C++, the high-level design we propose could be applied to other compiled languages. However, application of our ideas to interpreted or just-in-time-compiled languages requires a significant rethinking of the design. In a similar vein, the Trellis static analyzer can only propagate the privilege level tags up to the statically reachable portion of the call graph (i.e., as seen by the compiler), thus leaving out callbacks or dynamic calls in general. Likewise, our prototype implementation does not currently handle control flow transitions that deviate from normal function calls and returns (e.g., jumps into signal handlers, `longjmp`, C++ exception handlers across multiple levels of functions); however, Trellis could be extended to support these cases in principle. Furthermore, complex applications that use pre-allocated memory pools may necessitate further

developer effort to ensure that all dynamic memory regions are annotated correctly according to their appropriate privilege levels.

As evidenced by the dynamic memory allocator experiments in Section 5, Trellis may lead to a discernible performance impact when applied to memory allocation intensive applications. Although this problem could be alleviated through manual code optimization techniques such as using memory caching techniques instead of frequent calls to `trellis_malloc`, similar mechanisms could also be built directly inside Trellis's allocator to make the process transparent to application developers. Also note that using the Trellis memory allocator is only required for protecting sensitive application data, and the system's default allocator (or any custom allocator) could still be used for all other, non-sensitive memory allocations to avoid incurring runtime overhead.

One operation that Trellis does not yet support in a flexible, first-class manner is declassification of data to a lower privilege level. This capability can be useful in situations where the application developer can certify that information computed in a higher privileged context can safely be released to a lower privileged context in accordance with the application's security policies.

Since Trellis is an application-level access control mechanism, it is not designed to provide protection against attacks at lower-level system components (e.g., operating system exploits, hardware backdoors, etc.). Likewise, Trellis is designed strictly to support privilege separation; it does not aim to address common memory corruption vulnerabilities that may expose code and data *within the same privilege level* to attacks. These threats are outside the scope of this work, and Trellis should be used in conjunction with established defense mechanisms that support memory integrity such as ASLR [27] and stack canaries [13].

In future work, we plan to investigate extensions of Trellis to address limitations of the current approach. While our prototype implementation assumes a strict ordering of user privileges, it provides the necessary primitives to enable the lattice-based privilege model we describe in Section 3.1. One promising research direction is extending Trellis further to allow more complex access control models, to address concerns such as privacy of user profiles sharing the same privilege level. Other potential avenues of research include lifting the guarantees provided by the approach to higher-level languages, removing the need for source code, and reification of declassification within the framework. Finally, the dynamic memory segment permission management component of Trellis could be applied to other settings such as binary attack surface reduction by temporarily disabling access to unneeded portions of an application's address space.

7 Related Work

The principles of least privilege and privilege separation have long been studied as prominent software design principles to minimize trusted code in programs and contain damage in the face of security exploits [24, 25]. The architectures of many prevalent programs such as vsftpd [15], Postfix [28], and Sendmail [11] are explicitly designed around these principles.

Prior work most similar to Trellis aims to apply these principles to existing programs. Provos et al. [24] present a design methodology for applying privilege separation to security sensitive system services. Here, applications are separated into a privileged *monitor* and unprivileged *slave* processes that communicate via IPC channels. The authors also demonstrate their approach by discussing its application to OpenSSH. Kilpatrick [16] introduces a reusable framework to ease the implementation of partitioned applications. Bittau et al. [9] propose an application compartmentalization framework that provides a set of operating system primitives to assist developers with partitioning programs. In this approach, developers mark allocated memory regions with tags, and create special compartmentalized threads with specific access rights to the tagged memory.

While the above work lays down the guidelines for manually partitioning applications and provides tools for assistance, later work attempts to automate this process. Brumley and Song [10] use a combination of code annotation and static analysis to automatically separate C code into the aforementioned monitor and slave parts. Wu et al. [32] instead employ a dynamic data dependency analysis, which constructs a data dependency graph over the program's functions, and then partitions this into subgraphs representing least privilege components.

Trellis also uses code annotation and static analysis to perform privilege separation; however, unlike the previous work, it does not partition applications into separate processes. Instead, code and data at different privilege levels are segregated into separate memory pages, and access control is enforced by the operating system. Moreover, our system supports any number of privilege levels, instead of only a privileged and an unprivileged partition.

Other works apply similar application partitioning techniques to different security contexts. Kim and Zeldovich [17] present a Linux kernel module that allows unprivileged system users to utilize Linux security features (e.g., allocating new user IDs, setting up firewall rules, setting up `chroot` environments, etc.) to confine applications and reduce the amount of code running as root. Murray and Hand [22] discuss early ideas of segregating the trusted computing base of an application into small, dynamic libraries. Chong et al. [12] proposes a system that partitions web applications into JavaScript client-side code and Java server-side code according to the specified information-flow policies. Zdancewic et al. [33], and Zheng et al. [35] employ automatic code and data partitioning to address the problem of secure distributed computation. In contrast to the above, Trellis aims to address the problem of enforcing access control on multi-user applications that transitions between different privilege levels during runtime.

Various software security frameworks provide access control features analogous to Trellis for higher-level languages such as Java and Python, where access control enforcement capabilities are provided on top of the corresponding virtual machines or language interpreters. For instance, Apache Shiro [1] and Spring Security [6] allow Java applications to implement a role-based access control model for enterprise applications. In comparison, Trellis is applicable to multi-user programs written in C and C++, and it introduces a novel architecture to enforce hierarchical access control policies at the operating system level.

Trellis has comparable goals to History-Based Access Control (HBAC) [7], a model that assigns privileges to code during runtime based on previous execution history. This model largely relies on a runtime framework such as Java Virtual Machine or Common Language Runtime. In contrast, Trellis allows software developers to define privilege levels statically at compile time, and enforces access control at runtime. Similarly, Trellis shares some of its design principles with Decentralized Information Flow Control (DIFC) (e.g., Asbestos [14], Flume [18], HiStar [34]), which allows labeling data, and restricting its flow between application and operating system components. Other related program confinement solutions include various operating system mechanisms [5, 8, 23, 30], capability systems [26, 31], and software-based fault isolation techniques [19, 20, 29]. Unlike these, Trellis specifically addresses the problem of enforcing access control *within* an application. In particular, our system protects applications against vulnerabilities resulting from incorrect access control implementations, such as the misuse of GUI elements as access control primitives [21].

8 Conclusion

In this paper, we presented Trellis, a novel approach for specifying and enforcing access control policies in multi-user applications to separate code and data that logically belongs to different privilege levels. Enforcing such policies in multi-user applications is a responsibility that has heretofore been borne by application developers; Trellis automates this critical, error-prone aspect of application security. Our prototype implementation demonstrates that Trellis imposes a low end-to-end runtime performance overhead.

Acknowledgments. We would like to thank our shepherd Vasileios P. Kemerlis for his helpful feedback. This work was supported by the National Science Foundation (NSF) under grant CNS-1409738, and Secure Business Austria.

References

1. Apache Shiro. `https://shiro.apache.org/index.html`
2. HomeBank. `http://homebank.free.fr`
3. Linux Desktop Testing Project. `http://ldtp.freedesktop.org/`
4. Parasite. `https://chipx86.github.io/gtkparasite`
5. SELinux. `http://selinuxproject.org`
6. Spring Security. `http://projects.spring.io/spring-security`
7. Abadí, M., Fournet, C.: Access Control based on Execution History. In: NDSS (2003)
8. Badger, L., Sterne, D., Sherman, D., Walker, K.M., Haghighat, S.A.: A Domain and Type Enforcement UNIX Prototype. In: USENIX Security (1995)
9. Bittau, A., Marchenko, P., Handley, M., Karp, B.: Wedge: Splitting Applications into Reduced-privilege Compartments. In: USENIX NSDI (2008)
10. Brumley, D., Song, D.: Privtrans: Automatically Partitioning Programs for Privilege Separation. In: USENIX Security (2004)
11. Carson, M.E.: Sendmail without the Superuser. In: USENIX Security (1993)

12. Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L., Zheng, X.: Secure Web Applications via Automatic Partitioning. In: ACM SOSP (2007)
13. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: USENIX Security (1998)
14. Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., Morris, R.: Labels and Event Processes in the Asbestos Operating System. In: ACM SOSP (2005)
15. Evans, C.: very secure FTP daemon. `http://security.appspot.com/vsftpd.html`
16. Kilpatrick, D.: Privman: A Library for Partitioning Applications. In: USENIX ATC (2003)
17. Kim, T., Zeldovich, N.: Making Linux Protection Mechanisms Egalitarian with UserFS. In: USENIX Security (2010)
18. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information Flow Control for Standard OS Abstractions. In: ACM SOSP (2007)
19. McCamant, S., Morrisett, G.: Evaluating SFI for a CISC Architecture. In: USENIX Security (2006)
20. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: RockSalt: Better, Faster, Stronger SFI for the x86. In: ACM PLDI (2012)
21. Mulliner, C., Robertson, W., Kirda, E.: Hidden GEMs: Automated Discovery of Access Control Vulnerabilities in Graphical User Interfaces. In: IEEE S&P (2014)
22. Murray, D.G., Hand, S.: Privilege Separation Made Easy: Trusting Small Libraries Not Big Processes. In: EuroSec (2008)
23. Peterson, D., Bishop, M., Pandey, R.: A Flexible Containment Mechanism for Executing Untrusted Code. In: USENIX Security (2002)
24. Provos, N., Friedl, M., Honeyman, P.: Preventing Privilege Escalation. In: USENIX Security (2003)
25. Saltzer, J.H.: Protection and the Control of Information Sharing in Multics. Communications of ACM 17(7), 388–402 (1974)
26. Shapiro, J.S., Smith, J.M., Farber, D.J.: EROS: A Fast Capability System. In: ACM SOSP (1999)
27. The PaX Team: PaX Address Space Layout Randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt` (2003)
28. Venema, W.: The Postfix Homepage. `http://www.postfix.org/`
29. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient Software-based Fault Isolation. In: ACM SOSP (1993)
30. Walker, K.M., Sterne, D.F., Badger, M.L., Petkac, M.J., Sherman, D.L., Oostendorp, K.A.: Confining Root Programs with Domain and Type Enforcement (DTE). In: USENIX Security (1996)
31. Wilkes, M.V.: The Cambridge CAP Computer and Its Operating System. North-Holland Publishing Co. (1979)
32. Wu, Y., Sun, J., Liu, Y., Dong, J.S.: Automatically Partition Software into Least Privilege Components Using Dynamic Data Dependency Analysis. In: IEEE/ACM ASE (2013)
33. Zdancewic, S., Zheng, L., Nystrom, N., Myers, A.C.: Secure Program Partitioning. ACM Transactions on Computer Systems 20(3), 283–328 (2002)
34. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making Information Flow Explicit in HiStar. In: USENIX OSDI (2006)
35. Zheng, L., Chong, S., Myers, A.C., Zdancewic, S.: Using Replication and Partitioning to Build Secure Distributed Systems. In: IEEE S&P (2003)