Fides: Selectively Hardening Software Application Components against Kernel-level or Process-level Malware

Raoul Strackx
IBBT-DistriNet, KU Leuven
Celestijnenlaan 200a
3001 Heverlee, Belgium
raoul.strackx@cs.kuleuven.be

Frank Piessens
IBBT-DistriNet, KU Leuven
Celestijnenlaan 200a
3001 Heverlee, Belgium
frank.piessens@cs.kuleuven.be

ABSTRACT

Protecting commodity operating systems against software exploits is known to be challenging, because of their sheer size. The same goes for key software applications such as web browsers or mail clients. As a consequence, a significant fraction of internet-connected computers is infected with malware.

To mitigate this threat, we propose a combined approach of (1) a run-time security architecture that can efficiently protect fine-grained software modules executing on a standard operating system, and (2) a compiler that compiles standard C source code modules to such protected binary modules.

The offered security guarantees are significant: relying on a TCB of only a few thousand lines of code, we show that the power of arbitrary kernel-level or process-level malware is reduced to interacting with the module through the module's public API. With a proper API design and implementation, modules are fully protected.

The run-time architecture can be loaded on demand and only incurs performance overhead when it is loaded. Benchmarks show that, once loaded, it incurs a 3.22% system-wide performance cost. For applications that make intensive use of protected modules, and hence benefit most of the security guarantees provided, the performance cost is up to 14%.

Categories and Subject Descriptors

D.4.6 [**Operating Systems Security and Protection**]: Access Controls, Invasive Software

General Terms

Design, Security

Keywords

Trusted Computing, Secure Execution, Fully Abstract Compilation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1651-4/12/10 ...$15.00.

1. INTRODUCTION

A significant fraction of Internet-connected computers is infected with malware, usually with kernel-level access. Yet, many of these computers are used for security-sensitive tasks, and handle sensitive information such as passwords, corporate data, etc. While efforts to increase the security of commodity operating systems [30] and applications [36, 5, 10] are important and ongoing, their sheer size makes it very unlikely that they can be made secure enough to avoid infection [23, 11] with kernel-level malware. Due to the layered design of commodity systems, kernel-level malware can break the confidentiality and integrity of all data and code on the system.

This unfortunate situation has triggered researchers to design systems that can execute security sensitive code in an isolated area of the system, thus improving the security guarantees that can be offered. Of course, an important design goal (and design challenge) is to realize this while remaining compatible with current operating systems and hardware. Most of these proposed systems leverage recent hardware extensions for trusted computing or virtualization to execute code, and differ in the granularity of protection they offer (protection of full applications [10, 36, 5, 13, 19] versus protection of small pieces of application logic [20, 21, 28, 33, 4]), and in their root of trust (a correctly booted system [13, 33] or a hardware security module such as a TPM chip [14, 21, 28, 20, 4]). We provide a more complete overview of existing work in section 6.

State-of-the-art systems for protection of software modules focus on attesting the correct and secure execution of a single module to a third party [20, 21, 28, 4]. We focus on the complementary case of increasing the security guarantees of applications for the owner of the system. We propose an approach to selectively harden security-critical parts of an application. An SSL-enabled webserver, for example, could be built in a modular way where sensitive information is passed between trusted modules until it is finally encrypted and passed to the TCP/IP stack. This would reduce the power of a kernel-level attacker to one with only access to the network. Current systems are ill-equipped for this task: writing co-operating protected modules is too hard, messages passed between modules may never be delivered and a lack of support for multiple instances of the same module prevents a modular application design.

In this paper, we propose a system consisting of two parts: a run-time security architecture and a compiler. The security architecture implements a program-counter based access control model. A protected module is divided into a public

and a secret section. The secret section stores the sensitive data and is only accessible from within the module. The public section contains the module's code and can be read from outside of the module. This enables authentication and secure communication between modules in a cheap and secure way: an attacker is not able to intercept, modify or masquerade any messages between protected modules.

The compiler provides an easy way to compile standard C-code into protected modules. Since the program-counter dependent access control model allows modules and unprotected code to share the same virtual address space, their interaction is straightforward. This significantly simplifies the hardening of applications.

Modules compiled with our compiler effectively reduce the power of kernel-level malware and in-process attackers to only being able to interact with the modules through a public API. In earlier work[2] we have proven for a simplified model of our access control mechanism and compiler that with a proper API design and implementation the module is fully protected: an attacker that is able to inject arbitrary assembly code at kernel-level is only as strong as an attacker interacting through the module's API.

More specifically, we make the following contributions:

- We propose *Fides*, a security architecture for fine-grained protection of software modules, based on a memory access control model that makes access privileges dependent on the value of the program counter (instruction pointer). The access control model is strong enough to support fully abstract compilation [1] of modules; low-level attacks against a compiled module exist iff the source-level module can also be exploited.

- We show how this access control model supports novel features, such as

    1. the ability to support function pointers to trusted modules. Secrecy and integrity of any data passed as arguments is ensured by the authentication of the pointer's destination.
    2. the ability to update modules after they are deployed, thereby allowing legacy software to be ported easily and incrementally with minimal modification.

- We report on a fully functioning prototype implementation, demonstrating that Fides can be implemented on commodity hardware while remaining fully compatible with legacy systems.

- We present a compiler that compiles standard C source code modules into protected binary modules.

- We show that Fides has an average performance overhead of around 3% on the overall system, which is reduced to 0% when no modules are loaded. Macrobenchmarks show an overhead of up to 14% for applications that intensively use Fides' services.

We do not consider trusted I/O and leave it as future work. However, a trusted path between an I/O module and I/O devices can be established as in related work [22, 40].

The remainder of this paper is structured as follows. First, we clarify our objectives by defining the attacker model and desired security properties in Section 2. Section 3 gives an overview of the security architecture and its key concepts. In Sections 4 and 5, we discuss how the run-time system and compiler were implemented and evaluate performance. We finish with a discussion of related work and a conclusion.

2. OBJECTIVES

High-level programming languages offer protection facilities such as abstract data types, private field modifiers, or module systems. While these constructs were mainly designed to enforce software engineering principles, they can also be used as building blocks to provide security properties. Declaring a variable holding a cryptographic key as private, for example, prevents direct access from other classes. This protection however does not usually remain after the source code is compiled. An attacker with in-process or kernel level access is not bound by the type system of the higher language. We will show that Fides *is* able to provide such strong security guarantees. We first discuss the abilities of an attacker and then discuss how Fides provides these guarantees.

2.1 Attacker model

We consider an attacker with two powerful capabilities. First, an attacker can execute arbitrary code – user-level or kernel-level – in the legacy operating system. This kind of root-level access is a realistic threat: legacy operating systems consist of millions of lines of code and this unavoidably leads to the presence of programming bugs, such as buffer overflows [23], that can be exploited by an attacker to inject code [11, 39].

With kernel-level privileges, the attacker can try to corrupt or read the state of protected modules, modify the virtual memory layout of applications containing protected modules or intercept their loading process to tamper with security-sensitive code or data. The attacker can also try to intervene in the communication between modules, or to attack data that protected modules wish to store persistently.

Second, the attacker can build and deploy her own protected modules. Our security architecture does *not* assume that software modules that request protection can be trusted. In other words, it is our goal to ensure the security of a protected module by one stakeholder, even if modules of malicious stakeholders are also loaded in the system.

With respect to the cryptographic capabilities of the attacker, we assume the standard Dolev-Yao model [8]: cryptographic messages can be manipulated, for instance by duplicating, re-ordering or replaying them, but the underlying cryptographic primitives cannot be broken.

We assume the attacker has *no physical access to the hardware.* An attacker with control over the physical system may disconnect memory, place probes on the memory bus, or perform a hard reset. Since remote exploitation of commodity systems is far more common than exploitation through physical access, this is a reasonable assumption.

2.2 Security properties

To provide strong security guarantees, we use a combination of a run-time system and a compiler.

2.2.1 *Fides run-time system*

The Fides run-time system enforces a program-counter based access control mechanism. It guarantees the following security properties:

- *Restriction of entry points.* Protected modules can only be invoked through specific entry points, preventing an attacker from jumping to an incorrect location in the module and executing on unintended execution paths [32].

- *Confidentiality and integrity of module data.* A protected module can store sensitive data in a way so it can only be read or modified by the module itself.

- *Authentication of modules.* Modules can authenticate each other securely. This also implies that code of modules is integrity protected.

- *Secure communication between modules.* Fides guarantees integrity, confidentiality and delivery of data exchanged between modules.

- *Minimal TCB.* The correct and secure execution of a module only depends on (1) the hardware, (2) the Fides architecture and (3) the module itself and any other module that it calls. In particular, the operating system is excluded from the TCB.

2.2.2 Secure compilation of modules

Using the protection mechanisms offered by the run-time system as building blocks, the compiler allows the compilation of standard C source code into protected modules. It provides the following security guarantees:

- *Integrity of execution.* An attacker is not able to influence the correct execution of the module

- *Secure communication between modules.* The compiler ensures that sensitive information is passed only between modules using a secure channel.

- *Secrecy of sensitive information.* Only information that is passed explicitly to unprotected memory or to another module exits the module. Leakage of possibly sensitive information, for example information lingering in save-by-caller registers, is prevented.

Note that it is *not* our objective to protect against vulnerabilities in protected modules: the security of a protected module can be compromised if there are exploitable vulnerabilities in its implementation. Examples include logical faults (i.e. a faulty API design [17]), or memory errors [11, 39, 35]. Instead, our goal is to protect the module from malware that exploits vulnerabilities in the surrounding applications or underlying operating system. A vulnerable module however, can only affect the security of other modules if they explicitly place trust in the former and, for example, exchange sensitive information. An attacker introducing malicious modules in the system does *not* gain any more power as they are not trusted by any other module. In section 5 we will show that a low-level attack against modules exist iff also a high-level attack exists.

3. OVERVIEW OF THE APPROACH

In Fides, an application and the protected modules it uses, share the same virtual address space. Protection of the modules is provided by enforcing a memory access control model: access rights to memory locations depend on the value of the program counter. Roughly speaking, while the processor is executing within the boundaries of a specific protected module (i.e. the program counter points to an instruction that is part of the module), it can access memory allocated to that module. If the processor is executing outside the boundaries of the module, it has only limited rights to access the module's memory. In particular, destruction of the module is also only possible from within the module: this is why we use the term *self-protecting module (SPM)* for protected modules in Fides.

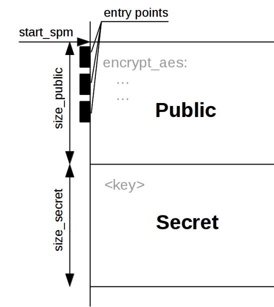


Figure 1: The layout of an SPM.

from\to	Entry pnt.	Public	Secret	unprot.
Entry pnt.		x		
Public	rx	rx	rw	rwx
Secret				
Unprot./ other SPM	rx	r		rwx

Table 1: The enforced memory access control model.

This section gives an overview of how this basic mechanism is used to support communicating protected modules. First, we discuss in more detail the layout of an SPM and the enforced memory access control (Section 3.1). Next, we describe the four primitive operations offered by Fides (Section 3.2). Then, in Sections 3.3 and 3.4, we discuss the typical life cycle of an SPM, and how SPMs can authenticate each other and collaborate securely. Finally we discuss in section 3.5 how modules can be updated.

3.1 Layout of a Self-Protecting Module

A self-protecting module is a chunk of memory divided in two sections (see Fig. 1). The *Secret* section contains the module's sensitive data. This not only includes sensitive data processed by the module, such as cryptographic keys, but also data used for the correct execution of the module, such as its call stack. Read and write access to the Secret section is only allowed from within the module. Access from outside the module, including from another instance of the same SPM, is prevented. These access restrictions give modules total control over any data stored in their Secret section.

The *Public* section contains information that is non-confidential and should only be integrity-protected. This includes all[1] the module's code and constant values such as strings. Once the module is protected, the contents of this section

[1]Self-modifying code and interpreted code could be supported easily by making the Secret section executable, but we consider such support of out scope for this paper.

can no longer be modified, it can only be read and/or executed. Read access is allowed from unprotected code as well as from other SPMs, allowing authentication of modules.

SPMs are able to access unprotected memory in the same address space. While Fides' design does not impose any access limitations on these locations, access restrictions set by the legacy kernel are enforced to prevent malicious modules, for example, from overwriting kernel space.

Each SPM comes with a list of memory locations in the Public section that are valid entry points into the SPM. Fides will guarantee that an SPM can only be executed by jumping to a valid entry point. This prevents attacks that attempt to extract information by selectively executing code snippets [32].

Table 1 summarizes the enforced access control rules. It shows for instance that code in unprotected memory or other SPMs can read the Public section of an SPM, or can execute an address that is an entry point of the SPM (from this point on, the program counter is within the Public section and the access rights are elevated).

3.2 Primitive operations

Fides implements four primitive operations to create, destroy and query the location and layout of SPMs.

The `crtSPM` primitive is used to create an SPM. It takes the location and size of the Public and Secret sections and a list of entry points. First, Fides verifies whether all referenced logical pages are mapped to physical pages, that they do not overlap with any existing SPMs, and that all entry points point into the Public section. Then, Fides creates an identifier `spm_id` for the SPM. Fides guarantees that until it is rebooted, no other SPM will receive the same identifier. Therefore the identifier can be used to differentiate instances of the same module. Fides also clears the Secret section (set to all zeroes) to remove the initial contents of the Secret section from the attack surface. Finally, memory access protection of the SPM is enabled.

The `killSPM` primitive will destruct the SPM that called it (or generate a fault if called from unprotected code). Enforcing that only SPMs can destruct themselves is important for security: it prevents attacks where an attacker destroys an SPM unexpectedly e.g. during a callback to unprotected memory. It also allows SPMs to pause their destruction until its sensitive data is stored securely to disk and overwritten in memory. Note that this does not prevent Fides from interrupting non-responsive modules (see section 4.1).

Fides supports two primitives to allow authentication of SPMs. The `lytSPM` primitive is given any virtual address and returns the base address, layout and `spm_id` of the module that is mapped at the specified location. The `tstSPM` primitive is more efficient and returns whether the SPM with a given `spm_id` is loaded starting at the specified location. Both primitives check whether the referenced SPM is loaded correctly: as SPMs are loaded in processes' virtual address space, pages may not be mapped, mapped to incorrect physical pages, or mapped out of order.

3.3 Life Cycle of a Self-Protecting Module

Fig. 2a and 2b describe the life cycle of an SPM. We explain the steps from creation to destruction in detail.

Setting up an SPM.

First (Fig. 2a, step 1), the legacy operating system provides a user process with a chunk of (possibly physically non-contiguous) memory and the SPM is placed in unprotected memory.

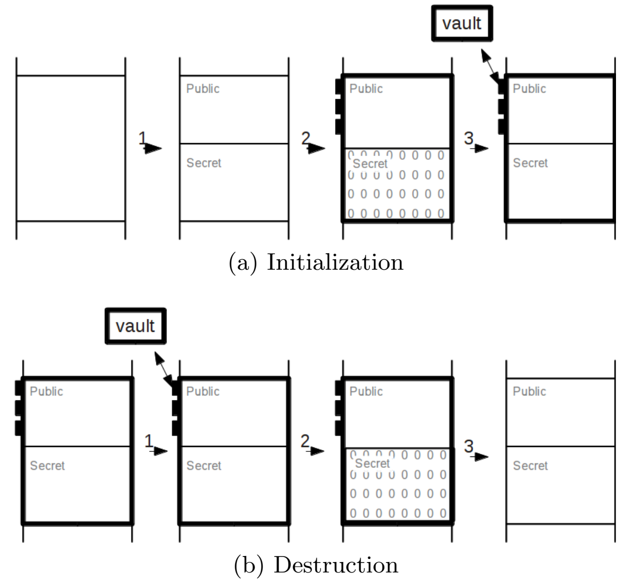


Figure 2: The life cycle of an SPM from (a) initialization to (b) destruction.

In the second step, the `crtSPM` primitive is called. An attacker that compromised the previous step(s), will be detected later on and mitigated. At this point, the SPM can be authenticated and service other modules. However, most SPMs will need to restore their secret state from persistent storage after creation (step 3). In Fides, this is handled by a special SPM, called *the Vault* that will only pass previously stored data over a secure channel. Details of authentication and secure communication will be discussed in Section 3.4. In appendix A we elaborate on the workings and security of the Vault.

Destroying an SPM.

When the SPM is no longer needed, it should be destroyed properly (Fig. 2b). First, the module accesses the Vault to store any secret data that must be available for later executions. In step 2, the secret data of the SPM is overwritten to prevent it from being disclosed in unprotected memory. Finally, the module calls the `killSPM` primitive to lift the imposed access control of the module's memory.

3.4 Secure local communication

One of Fides' objectives is to support a system of collaborating modules (see Fig. 3), each with its own secrets and own services that it offers to other modules, adhering to the principle of least privilege [29]. Hence, SPMs must be able to authenticate each other, and establish secure communication channels. We explain both aspects in detail.

The identity of an SPM is captured in what we call a *security report*. It contains four parts:

- *A cryptographic hash of the Public section* allowing verification that the Public section was not compromised before protection was enabled. It is essentially

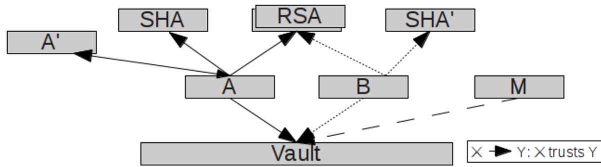


Figure 3: Fides' ability to establish secure channels, enables the easy creation of complex trust networks. Modules A and B are able to explicitly place trust on (possibly another instance of) an SPM implementing RSA operations without trusting each other. Similarly, a malicious module M will not be able to cause any harm as it is not trusted by any module.

the SPM's identity as the access control model only allows code execution from this section.

- *The layout of the SPM, including the sizes of the Public and Secret sections, and the list of entry points (relative to the Public section)* to verify that the protection request was not compromised. Modification of the size of the Secret section in the SPM's initialization phase for example, may cause the use of unprotected memory to store sensitive information.

- *A serial and version number.* The authentication mechanism is flexible enough to support SPMs to be updated easily. As the cryptographic hash of the Public section will differ between versions, a serial number is required to link different versions of the same SPM together. A version number prevents the re-use of old (e.g. security vulnerable) SPMs.

- *Cryptographic signature.* The security report is signed by its issuer. SPMs have a list of trusted certificate authorities (CAs) to verify the signature of SPMs they authenticate.

Since a security report is signed by its issuer, it can be stored in unprotected memory. Any working mechanism to retrieve an SPM's security report will suffice. For simplicity, we will assume in the rest of the paper that it is stored in front of the SPM.

One-way authenticated service call.

Consider a *SecureRandom* module that provides cryptographic random numbers, and a *Client* module that authenticates and requests its service (Fig. 4a).

First, the Client calls the `lytSPM` primitive, locates the security report, and verifies (1) its signature on the security report, (2) the hash of the Public section and the layout of the SPM against the information in the security report, and (3) whether serial and version number are as expected.

Second, the SecureRandom module is called. This is similar to calling a function: parameters are loaded in registers and a jump to the appropriate entry point is performed. An important difference with regular function calls on the x86 platform is that the return address must also be passed in a register. Under normal operation return addresses are pushed on the call stack of the caller. However, to protect the integrity of their execution, modules are not allowed to access each others call stack and the return address cannot be retrieved. Hence, a continuation entry point, in this case `receive_random`, is provided as a parameter (similar to continuation-passing-style programming[26, 3]).

In the final step, SecureRandom generates a random number and returns it by performing a jump operation to the `receive_random` entry point.

The bandwidth of the secure channel can be increased significantly by storing large messages in memory shared between sender and receiver. We will further discuss this mechanism in Section 4.1.

In case the Client module requires additional random numbers, the SecureRandom module can be re-authenticated using the `tstSPM` primitive. Based on the `spm_id` returned by `lytSPM` when the module was first authenticated, it ensures that the *same instance* of the module will be accessed and rechecking the security report is not required. We will show in Section 5 that repeated authentication is significantly more efficient than initial authentication.

Two-way authenticated service call.

Two-way authentication is very similar. Assume that a module Client wishes to communicate with another module, Server, and that mutual authentication is required (Fig. 4b).

First, Client locates the Server's security report and authenticates the module as in one-way authentication. In step two, a message is sent to the Server containing the entry point, `receive_secret`, where the response should be returned to. In step three, the Server locates the Client's security report using the provided return point and the `lytSPM` primitive. Only after successful authentication of the client, sensitive data is returned.

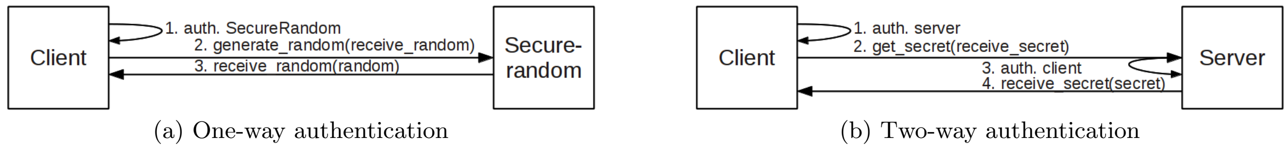
In case the origin of service requests must be proven, a secret session token can be passed to the authenticated endpoint during the initial authentication. The session-token should be passed in all future service requests.

3.5 Updating SPMs

Fides' authentication scheme allows a module to be updated easily without requiring any modification of modules or unprotected code that are clients of the updated module.

Updating works as follows. A client authenticates a module starting from just a function pointer: using the `lytSPM` primitive, the SPM queries Fides for the base address of the referenced module and locates its security report. After authenticating the issuer and verifying its serial number[2], the module is authenticated as described in Section 3.4. In addition the client should check the version number. Updated versions might contain API inconsistencies to previous versions e.g. services may be serviced on new entry points. Similarly, the version number should be high enough to prevent attacks where a module is downgraded to an older, vulnerable version. To transfer secret state from an old version to a newer, special support is required. An update protocol could be implemented by the modules to pass the information, or a support SPM could be built to pass persistent sensitive information to updated SPMs. This approach allows the Vault to remain simple and easy verifiable: it will only return sensitive data to the same SPM that previously requested storage.

[2]The issuer/serial number combination is assumed to uniquely determine functionality, and this should be stable over updates of the module.



(a) One-way authentication (b) Two-way authentication

Figure 4: Communication protocols between two SPMs.

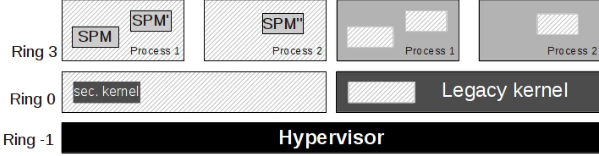


Figure 5: Layout of the Fides architecture. Hatched areas represent partially accessible memory regions.

Security of updating modules depends on the ability to create crytographically signed security reports. Since it can safely be assumed that only the creator of the initial report has access to the private key, an attacker is not able to fabricate his own new versions.

4. A PROTOTYPE IMPLEMENTATION

A key element of Fides is the program-counter dependent memory access control model. Since access rights have to be checked on each memory access, implementing this completely in software would have a huge performance cost. Alternatively, modifying hardware, an approach taken by related research [37, 38, 9], has serious drawbacks. In this section, we describe an efficient implementation of the runtime system and the compiler on readily available hardware.

4.1 The Fides architecture

The key observation is that the memory access control rules only change when entering and exiting SPMs. In our implementation, we use a small dynamic hypervisor to isolate SPMs from the rest of the system, and we ensure that the correct memory permissions are set on entering/exiting SPMs. Hence, there is only an overhead on entering and exiting SPMs, leading to a reasonable performance overhead.

We introduce a minimal hypervisor that runs two virtual machines, the *Secure VM* and the *Legacy VM* (see Fig. 5). Both VMs have the same guest physical view of host physical memory, but they have a different configuration of memory access control. Note that there is no duplication of memory, only two virtual views of the same physical memory.

Our prototype implementation can be loaded and unloaded when required, avoiding any overhead when no SPMs are in use. Fides is bootstrapped by loading a device driver in the legacy kernel to gain supervisor privileges. Then, physical contiguous memory is allocated to store a hypervisor and the Secure VM. Next, a dynamic root of trust is started and the hypervisor and Secure VM are launched. Finally, the running legacy kernel is pulled in the Legacy VM, and memory access control of both VMs is configured [15, 27].

The Legacy VM.

The legacy kernel and user applications continue their operation without any interruption: the only difference after the start of Fides is that access to certain parts of memory is now prohibited in the Legacy VM. More specifically, the memory where SPMs and parts of the TCB are stored, is protected. If the legacy VM accesses this memory (for instance tries to read or write the Secret section of an SPM), this will trap to the hypervisor and the access attempt is prevented.

The Fides device driver that was used to bootstrap Fides also provides an interface to the security architecture, for instance to create and query SPMs. This interface can not be exploited: no sensitive information is ever returned. However, given our attacker model, an attacker may change the returned results before code in the Legacy VM can process it. Hence, these primitives can only be used securely from within an SPM thereby avoiding the results to leave the Secure VM. This problem does not occur when SPMs are created from unprotected memory: when an attacker interfered with the creation of an SPM, this will be detected by the authentication protocol and no sensitive information will be passed to it.

This design ensures excellent compatibility with legacy operating systems: the only change from the OS's viewpoint is that certain memory regions (that are unused during normal operation) are rendered inaccessible.

Hypervisor.

The hypervisor is executing at the most privileged level and serves three simple purposes. First, it provides coarse-grained memory isolation of the legacy VM, secure VM and itself. It also prevents any access to SPMs that is not allowed from unprotected memory. However, it does not implement the fine-grained program-counter dependent memory access control. This is implemented in a separate security kernel in the secure VM and will be discussed later.

Second, the hypervisor schedules both virtual machines for execution based on a simple request passing mechanism. The secure VM is scheduled only when a request is passed to it (i.e. when an SPM is called), or when it did not yet finish executing the previous request. Hence, the Secure VM consumes no CPU cycles when no SPMs are being executed.

Third, the hypervisor creates a new dynamic root of trust (DRTM aka late launch) when it is loaded. This allows the attestation of the correct launch of the security architecture. It also allows the TPM chip to store sensitive information based on this measurement, such as the cryptographic keys used by the Vault. If Fides was compromised before it was protected in memory and launched or a hypervisor was already present, the result of this measurement differs and sensitive data is inaccessible.

Secure VM.

To allow easy access to the unprotected memory, the Secure VM has the same view of physical memory as the Legacy VM but with different access control settings: SPMs can be accessed but are protected by a security kernel

Security kernel.

To reduce the size of the TCB, only a minimal amount of features are used: memory paging, a separation of user and kernel mode, page fault handling and a few system calls. We now discuss how these features are used to enforce the fine-grained access control model and how SPMs can use Fides' primitive operations.

To ensure isolation, SPMs are executed in user mode. When a module is invoked, the security kernel receives a request specifying the virtual address of the entry point called. This address is translated to a physical address by directly traversing the (untrusted) page tables in the Legacy VM. Next the containing module is located. When no module is found an error is returned to the Legacy VM, else a new address space is created mapping the entire module. As modules are always mapped at the same virtual addresses as in the Legacy VM, it is easy to access unprotected memory locations. When these are not yet mapped, a page fault will be generated. At that time the referenced physical page is located, checked whether it is part of an SPM and checked against the access rights of the SPM. To prevent an SPM from receiving unauthorized access to memory locations, the address space is rebuilt each time an SPM is invoked. Note that the page tables of the Legacy VM are *not* trusted: they are only used to check which physical page was referenced.

The security kernel also ensures that modules are properly loaded: since the untrusted page tables of the legacy kernel are used, an attacker may try to only load modules partially in memory or rearrange the order of its pages. To mitigate this threat, the security kernel records the order of the *physical pages* when a module is created and ensures that the same order is used when the module is called or its presence tested using the `lytSPM` and `tstSPM` primitives.

TOCTOU attacks are mitigated by preventing concurrent execution of modules. As modules can only destroy themselves, an authenticated module must still be mapped in memory when it is called. This is however overly restrictive as only the presence of the called module must be ensured.

Besides passing information between SPMs in registers, the security kernel also provides support to pass bulk data using a special shared memory segment that is accessible only to the currently executing SPM. Hence, the receiver automatically gains access when it is called. To prevent information leakage, the called SPM must overwrite the passed data before execution returns to unprotected code. Access to this memory segment from the legacy VM is prevented using Extended Page Tables (EPT), the same hardware mechanism used to isolate different VMs.

Limitations of the prototype implementation.

To prevent time-of-check-to-time-of-use attacks, SPMs must not be destroyed after they were authenticated but before they are called. This would cause sensitive information stored in registers to leak to untrusted code. Our prototype currently handles this by preventing SPMs to be interrupted. However, this is largely a prototype limitation, and *not* fundamental. Fides could, for example, support interrupts by suspending and resuming the executing SPM after the interrupt is handled. Entries to other SPMs are denied to prevent destruction of already authenticated communication endpoints by the interrupted module. Non-responsive modules on the other hand, may be destroyed by the security kernel without further consideration. Alternatively, support for multicore processors could also be added and SPMs can be run on one specific core. In that case, unprotected code is able to execute uninterrupted. This may be acceptable, as it can be expected that SPMs are only responsible for a small fraction of all computation.

In production systems the use of DMA should be prevented from overwriting an SPM. Just as the prototype currently prevents the Legacy VM to access SPM locations, the IOMMU should be configured to prevent DMA accesses to modules.

4.2 Automated compilation of modules

We modified the LLVM[3] compiler to compile standard C source code modules into protected modules. More specifically, the compiler ensures the following:

- Each module implements its own stack.
- When returning to unprotected memory, registers and condition flags are cleared.
- Function pointers point to unprotected memory or to a function in the SPM with a correct signature.
- Function call annotations specify that the referenced module must be authenticated before the function is called and possibly sensitive information is passed.
- The entry point handling returns from callback functions, cannot be exploited. The entry point is only serviced when a callback actually took place.

We now discuss two notable implementation details: supporting function calls to SPMs and the use of function pointers by modules.

Supporting function calls to SPMs.

For each SPM, a wrapper is created to allow easy invocation. The wrapper serves two purposes. First, it loads and unloads the SPM when appropriate. Second, it creates a stub function for each available entry point. Fig. 6 displays a schematic overview of an invocation. In step 1 untrusted code accesses a stub in the SPM's interface as a normal function. Arguments are passed together with the entry point to the security kernel via the Fides driver and hypervisor (step 2). After all consistency checks pass, the SPM is invoked (step 3). The SPM's execution stops when it tries to execute unprotected memory, either because the SPM's service returns or because an external function is called. In both cases the security kernel returns the contents of all registers to the stub (steps 4-6). There appropriate action is taken: returning to its caller or invocation of the function pointer before re-entering the SPM.

Supporting function pointers.

Support for function pointers dereferenced within a module is added in two steps. In the first step, an LLVM function pass replaces every function pointer dereference with a call to one of two support functions, depending on whether the call should be made to a trusted module or unprotected code. The developer of the module should specify the type of the target of the function pointer by annotating the source code. These support functions will be compiled as part of the

[3] `http://llvm.org/`

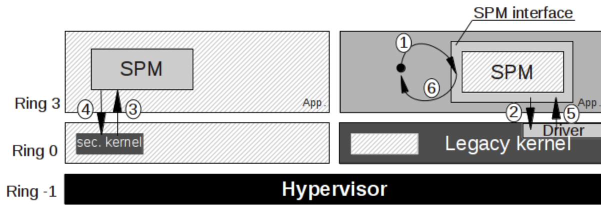


Figure 6: Implementation of the Fides architecture.

module and allows easy handling of function pointers without having to use the LLVM intermediate representation. In step two, the module of the target of the function pointer is authenticated, if required. The `spm_id` of the destination is stored within the boundaries of the module to limit the cost of subsequent calls. Finally, possible side channels that leak information about the inner state of the module, such as the condition flags and save-by-caller registers are cleared and the value of the stack pointer is stored before a jump to the function pointer is made. A special return entry point is added to the module (one per module) to facilitate returns from the function. As with any entry point, the return location after the module has serviced the request, is passed through register `%rbp`.

Compiler limitations.

While modules can be written in standard C-code, some source code annotations are required. Allocation requests using the standard `malloc` function, for example, must specify where the memory should be allocated. A support library has been created to statically link functions of the libc library with the module.

5. EVALUATION

5.1 Security Evaluation

Fides offers strong security guarantees to compiled modules by relying on a run-time system of only limited size and a compiler. We now discuss both components.

Run-time system.

The run-time system implements a program-counter based access control mechanism in three layers. At the lowest level we rely on the TPM chip and assume that an attacker is not able to launch physical attacks against the system. When the Fides architecture is loaded, a dynamic root of trust measurement (DRTM) is started, measuring the memory state of the system. Based on this measurement the cryptographic keys used by the Vault are sealed in the TPM. An attacker that compromised the correct loading of Fides, for example by modifying the binary on disk, will cause an incorrect measurement and access to the sealed keys is prevented. As the Vault is the only SPM that stores persistent data, access to sensitive data is prevented.

In the second layer, the hypervisor protects all security sensitive memory locations against faulty legacy applications and a compromised kernel. This includes secrecy of confidential data as well as integrity of code.

The third layer, the security kernel, protects modules from potentially malicious SPMs by realizing the program-counter dependent access control model.

VMM	Secure kernel	Shared	Total TCB
1,045	1,947	4,167	7,159

Table 2: The TCB consists of only 7K lines of code.

An important enabler for formal verification of the TCB is to make sure that the size of the TCB is small. Table 2 displays the TCB of Fides for its different parts, as measured by the SLOCCount[4] application. Only the hypervisor (VMM) and the secure kernel are trusted. They contain 1,045 and 1,947 lines of C and assembly code respectively. This does not include the 4,167 lines of code that is shared between the parts. The driver (690 LOC) used to support communication with Fides is not security sensitive and thus is excluded from the TCB. This totals the size of the TCB to only 7,159 LOC.

Compilation of SPMs.

Facilities offered by high-level languages such as a private field modifier, allow easy reasoning about an application's security guarantees and its verification. A low-level attacker, not bound by these restrictions, may however still exploit a vulnerability anywhere in the application and break these security guarantees. It has been proven [2] that a simplified version of Fides' fine-grained access control mechanism, is able to support fully-abstract compilation of a high-level language with private fields: when no source-level attack against a module exists, it also can't be exploited after compilation.

To achieve such high security guarantees, modules must be compiled securely. Each module's stack, for example, must be placed in its Secret section. Our modified compiler is able to compile standard C source code to modules meeting these requirements.

5.2 Performance evaluation

We performed three types of performance benchmarks on our prototype. First, we measure the system-wide performance impact of Fides. Next, we measure the cost of local communication (Section 5.2.2) and finally we benchmark an SSL-enabled web server (Section 5.2.3).

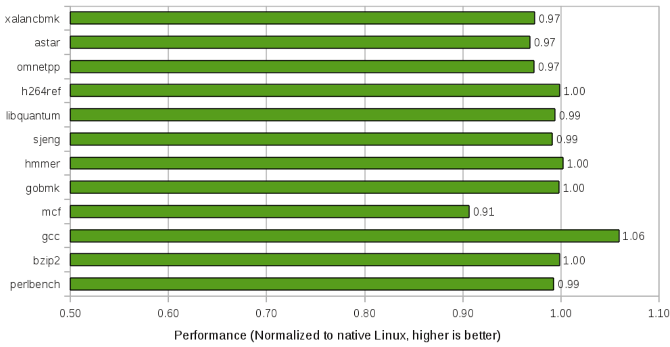
All our experiments were performed on a Dell Latitude E6510, a mid-end consumer laptop equipped with an Intel Core i5 560M processor running at 2.67 GHz and 4 GiB of RAM. Due to limitations of our prototype, we disabled all but one core in the BIOS. An unmodified version of KUbuntu 10.10 running the 2.6.35-22-generic x86_64 kernel was used as the operating system.

5.2.1 System-wide performance cost

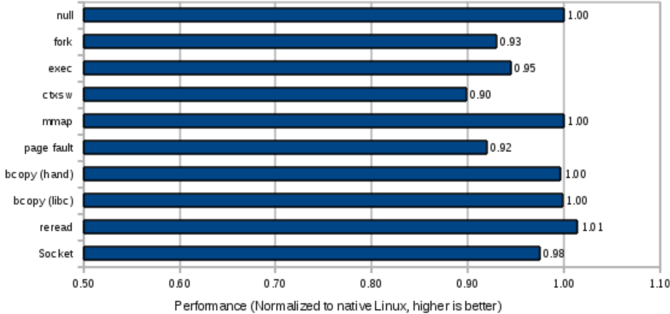
To measure the performance impact of Fides on the overall system, we ran the SPECint 2006 and lmbench benchmarks. Fig. 7a displays the results of the former. With the exception of the `mcf` application (10.36%), all applications have an overhead of less than 3.28%. We contribute the performance increase of gcc to cache effects.

Fig. 7b displays the results of 9 important operations of the lmbench suite: null (null system call), fork, exec, ctxsw (context switch among 16 processes, each 64 KiB in size), mmap, page fault, bcopy (block memory copy), mmap read

[4] `http://www.dwheeler.com/sloccount/`



(a) SPECint 2006



(b) lmbench

Figure 7: The performance impact on the overall system

(read from a file mapped into a process), and socket (local communication by socket). All tests show a performance overhead of less than 10% and on average even as low as 3.22%. As our implementation does not require any additional computation when no SPMs are executed, this performance overhead can be contributed completely to the hardware virtualization support. We expect that as this support matures, overhead will be reduced further. Also note that Fides can be unloaded when it is no longer required, reducing the overhead to 0%.

5.2.2 Local communication between SPMs

To measure the impact of communication, two microbenchmarks were used. The first one measures the cost of a call to an SPM compared to a call to a similar driver in the legacy operating system. A simple SPM of two 4 KiB pages was used for the test. When its entry point is executed, it immediately passes control back to the calling application. The driver used for comparison is similar. When it is accessed using the ioctl interface, it immediately returns. Table 3 displays two results. The Entry row shows the measurement of the time between the point of call in the user application and the point of delivery in the SPM or driver. The Round trip row measures the time between call and return. Each test was executed 100,000 times. Results show a performance overhead of 8,167% and 8,781% respectively. This significant overhead is caused by the fact that for each SPM invocation two VM entries and exits are required to pass execution control from the legacy VM to the secure VM and back, as well as four context switches from supervisor to user mode are required (two in each VM, see Fig. 6). However,

	SPM	Driver	Overhead
Entry	4.35	0.05	8,167.14%
Round trip	6.58	0.07	8,781.73%

Table 3: SPM vs. driver access overhead (in μs).

	one-way auth.		two-way auth.	
	tstSPM	sha512	tstSPM	sha512
timing	7.82	95.72	8.28	110.63
overhead	6.03%	1,198%	12.22%	1,400%

Table 4: Microbenchmarks measuring the cost (in μs) of calling an (authenticated) SPM.

given the substantial security guarantees, these costs seem very acceptable.

The second microbenchmark measures the cost of different authentication techniques, over an average of 100,000 runs. Two SPMs were created, called Ping and Pong. Ping invokes a service in Pong that simply returns a static response. Four different setups were used, shown in table 4. Performance results without any authentication measured 7.37μs and is used as the baseline. Columns one and two display results for one-way authentication. Column one measures repeated authentication where Fides is requested to check the `spm_id` of the called module and column two measures an initial authentication using SHA-512 and the `lytSPM` system call, which is less flexible than authentication using a security report, but used, for example, by the Vault. Initial authentication has a performance penalty of 1,197.97%. Repeated authentication is much less expensive at 6.03%. Similar tests were conducted for two-way authentication. Performance penalties increased to 1,400.06% and 12. 22% respectively.

5.2.3 SSL-Enabled Web Server

As a macrobenchmark, we protected an SSL-enabled web server. Our goal was not only to protect the web server's long term secret, but the entire SSL-connection, including session information. This prevents a kernel-level attacker from hijacking the connection and renders him only as powerful as an attacker with complete control over the network.

We used the PolarSSL cryptographic library[5] and some functions of the diet libc library that are security sensitive in our use case (i.e. `sscanf`) to implement the SPM. We also implemented our own simple `malloc` memory management. The NSPR[6] library was used to create a multithreaded server. Each connection is handled in a separate thread with its own SPM.

We used the Apache Benchmark to benchmark a web server returning a static 74-byte page to the client over an SSL-connection protected by a 1024-bit RSA encryption key. Table 5 displays the server's performance with a varying number of concurrent tranctions, each setup receiving 10,000 requests. Repeated context switches during the SSL negotiation phase lead to a performance cost of up to 13.93%.

6. RELATED WORK

There exists a vast body of research on software security. For system-level software, memory safety related vul-

[5] `http://polarssl.org/`
[6] `https://www.mozilla.org/projects/nspr/`

Concurrency	Unprotected	Protected
1	50.27	50.09
5	83.72	83.26
10	97.34	83.44
50	102.73	89.27
100	103.10	88.74

Table 5: HTTPS-server performance (in #req/s).

nerabilities are an important threat. We refer the reader to Younan et al. [39] for a comprehensive survey and to Erlingsson et al. [12] for a gentle introduction. Practical countermeasures however, cannot defend against all possible attacks and countermeasures with strong guarantees typically come with a significant cost.

An alternative approach is to turn to formal verification of systems and applications to provide very strong assurance of security properties. Impressive achievements include the verification of the HyperV hypervisor [6], and the complete seL4 microkernel [16]. While seL4 is also able to provide strong security guarantees, a key design objective for Fides is compatibility with legacy operating systems: what should *minimally* change to a commodity OS to support protection of critical software components against kernel-level malware. For that design objective Fides outperforms seL4 easily.

Other research results proposes hardware modifications to increase security guarantees [37, 38]. However, one of our objectives is to remain compatible with existing systems.

Yet another line of research takes advantage of virtualization techniques to increase the protection of sensitive data by increasing protection of the kernel [30] or applications [10, 36, 5] in the presence of malware. While these research results present interesting solutions, we are not convinced that they can ever be made provable secure due to a possibly very large TCB. The line of research most related to the work in this paper sets out to bootstrap trust in commodity computers by securely capturing a computer's state. An excellent survey of this research field is given by Parno et al. [25]. We only discuss the most relevant work.

Existing research can be categorized based on the root of trust. Some works assume a trusted boot sequence to start a hypervisor before the commodity operating system is loaded. Terra [13] takes this approach to isolate closed boxes of software. Possible attack vectors are minimized by preventing additional code to be loaded in the box. Nizza [33] takes a more integrated approach, executing small pieces of code in isolation on the Nizza microkernel. While this architecture is similar to Fides, its TCB of 100,000 lines of code is an order of magnitude larger.

The root of trust can also be started dynamically, after the system has booted. Pioneer [31] and Conqueror [18] take this approach completely in software. However, many assumptions are hard to guarantee in practice and confidentiality of data cannot be provided.

Stronger guarantees can be provided when the TPM chip is used [7, 14]. Seminal work in this field has been conducted by McCune et al. Their Flicker architecture [21] can execute pieces of code, called PALs, in complete isolation while secrecy of sensitive information is guaranteed. The TPM chip is used intensively by Flicker, leading to a significant performance cost. The TrustVisor architecture [20] mitigates many of these disadvantages by using a hypervisor and a software delegate of the TPM chip. P-MAPS [28] operates similar to TrustVisor but does allow protected code to access unprotected pages. More recently, Azab et al. showed [4] that the System Management Mode (SMM) can be used to implement a hypervisor-like security measure, ensuring integrity and security of module code and data. While these systems also offer strong isolation of modules, their focus is on remote attestation. They are ill-equipped for practical implementation of applications with a large number of (interconnected) modules: (1) writing co-operating protected modules is hard since modules do not share the same address space. (2) Messages sent between modules may never be delivered. (3) A lack of support for multiple instances of the same module makes it extremely challenging to build modular systems. Our approach mitigates these disadvantages by combining a run-time system and a compiler to allow programmers to easily develop protected modules that are able to seamlessly interact with unprotected code and other modules. Fides' dual VM architecture also ensures 7 to 2,000 times (TrustVisor and Flicker, resp.) faster context switches from unprotected memory to SPM's.

Finally, our own previous work on trusted subsystems in embedded systems [34] proposed a program-counter dependent memory access control model. An implementation technique was sketched for embedded systems with a flat address space and with special-purpose hardware support. El Defrawy et al. [9] implemented attestation of code in embedded devices based on this access control model, but limited themselves to only a single module. This significantly reduces the complexity of a hardware implementation as no primitives for module creation, destruction or authentication is needed. A full hardware implementation for high-end CPUs may not be feasible as the access control mechanism would require interaction with existing memory translation mechanisms. A key contribution of Fides is that it shows that similar ideas can be implemented on commodity hardware while remaining fully compatible with legacy software.

7. CONCLUSIONS

Commodity operating systems have been proven hard to protect against kernel-level malware. This paper presented a combined run-time system and compiler approach to selectively harden modules. Using a program-counter based access control model, programmers are able to develop modules in standard C-code that co-operate seamlessly with unprotected code and other modules. It has been proven that such modules are fully protected while system-wide performance impact is limited.

8. ACKNOWLEDGMENTS

The authors thank all reviewers and proofreaders of the paper for their useful comments. We also explicitly thank Gijs Vanspauwen for his work on the compiler.

This research was done with the financial support from the Prevention against Crime Programme of the European Union, the IBBT, the IWT, the Research Fund KU Leuven, and the EU-funded FP7 project NESSoS.

9. REFERENCES

[1] Abadi, M., and Plotkin, G. D. On protection by layout randomization. In *Computer Security Foundations Symposium (CSF)* (2010), pp. 337–351.

[2] Agten, P., Strackx, R., Jacobs, B., and Piessens, F. Secure compilation to modern processors. In *Computer Security Foundations Symposium* (2012), pp. 171–185.

[3] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[4] Azab, A., Ning, P., and Zhang, X. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 375–388.

[5] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. K. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS* (2008).

[6] Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. Vcc: A practical system for verifying concurrent c. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, 2009), TPHOLs '09, Springer-Verlag, pp. 23–42.

[7] Datta, A., Franklin, J., Garg, D., and Kaynar, D. A logic of secure systems and its application to trusted computing. In *30th IEEE Symposium on Security and Privacy* (2009), IEEE, pp. 221–236.

[8] Dolev, D., and Yao, A. C. On the security of public key protocols. *IEEE Transactions on Information Theory 29*, 2 (1983), 198–208.

[9] El Defrawy, K., Aurélien Francillon, D., and Tsudik, G. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Proceedings of the Network & Distributed System Security Symposium (NDSS), San Diego, CA* (2012).

[10] England, P., Lampson, B., Manferdelli, J., and Willman, B. A trusted open platform. *Computer 36*, 7 (July 2003), 55 – 62.

[11] Erlingsson, Ú. Low-level software security: Attacks and defenses. *Foundations of Security Analysis and Design IV* (2007), 92–134.

[12] Erlingsson, U., Younan, Y., and Piessens, F. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer, 2010.

[13] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A virtual machine-based platform for trusted computing. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 193–206.

[14] Kauer, B. Oslo: improving the security of trusted computing. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–9.

[15] King, S., Chen, P., Wang, Y., Verbowski, C., Wang, H., and Lorch, J. SubVirt: Implementing malware with virtual machines. *IEEE Symposium on Security and Privacy (Oakland)* (2006).

[16] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 207–220.

[17] Longley, D., and Rigby, S. An automatic search for security flaws in key management schemes. *Computers & Security 11*, 1 (1992), 75–89.

[18] Martignoni, L., Paleari, R., and Bruschi, D. Conqueror: tamper-proof code execution on legacy systems. In *Proceedings of the 7th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)* (July 2010), Lecture Notes in Computer Science, Springer, pp. 21–40. Bonn, Germany.

[19] Martignoni, L., Poosankam, P., Zaharia, M., Han, J., McCamant, S., Song, D., Paxson, V., Perrig, A., Shenker, S., and Stoica, I. Cloud terminal: Secure access to sensitive applications from untrusted systems.

[20] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2010).

[21] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)* (Apr. 2008), ACM, pp. 315–328.

[22] McCune, J. M., Perrig, A., and Reiter, M. K. Safe passage for passwords and other sensitive data. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)* (Feb. 2009).

[23] One, A. Smashing the stack for fun and profit. *Phrack magazine 7*, 49 (1996).

[24] Parno, B., Lorch, J. R., Douceur, J. R., Mickens, J., and McCune, J. M. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2011).

[25] Parno, B., Mccune, J. M., and Perrig, A. Bootstrapping trust in commodity computers. In *In Proceedings of the IEEE Symposium on Security and Privacy* (2010).

[26] Reynolds, J. Definitional interpreters for higher-order programming languages. In *proceedings 25th ACM National Conference* (1972), pp. 717–740.

[27] Rutkowska, J. Subverting VistaTM Kernel For Fun And Profit. *Black Hat Briefings* (2006).

[28] Sahita R, Warrier U., D. P. Protecting Critical Applications on Mobile Platforms. *Intel Technology Journal 13* (2009), 16–35.

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

[30] Seshadri, A., Luk, M., Qu, N., and Perrig, A. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (2007), ACM, pp. 335–350.

[31] Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2005), ACM, pp. 1–15.

[32] Shacham, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 552–561.

[33] Singaravelu, L., Pu, C., Härtig, H., and Helmuth, C. Reducing tcb complexity for security-sensitive applications: three case studies. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (New York, NY, USA, 2006), ACM, pp. 161–174.

[34] Strackx, R., Piessens, F., and Preneel, B. Efficient Isolation of Trusted Subsystems in Embedded Systems. *Security and Privacy in Communication Networks* (2010), 344–361.

[35] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security* (2009), ACM, pp. 1–8.

[36] Ta-Min, R., Litty, L., and Lie, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 279–292.

[37] Thekkath, D. L. C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M. Architectural support for copy and tamper resistant software. *SIGOPS Oper. Syst. Rev. 34* (November 2000), 168–177.

[38] Williams, P., and Boivie, R. Cpu support for secure executables. *Trust and Trustworthy Computing* (2011), 172–187.

[39] Younan, Y., Joosen, W., and Piessens, F. Code injection in c and c++ : A survey of vulnerabilities and countermeasures. Tech. rep., Department of Computer Science, KULeuven, 2004.

[40] Zhou, Z., Gligor, V., Newsome, J., and McCune, J. Building verifiable trusted path on commodity x86 computers. In *IEEE Symposium on Security and Privacy* (2012).

APPENDIX

A. THE VAULT

The Vault is an SPM that stores sensitive information on behalf of other SPMs. It offers two services. First, an SPM can ask the Vault to store persistent secret data. The Vault will append the identity of the requesting SPM (its layout and cryptographic hash of the public section), encrypt and sign the data and store it using the (untrusted) services of the legacy operating system.

Second, *only* an SPM that previously stored secret data can retrieve it again. After mutual authentication, the Vault retrieves the encrypted data from the legacy operating system, checks its integrity, decrypts it and sends it over a secure channel to the requesting SPM.

The Vault is treated specially by Fides: it is created when Fides is booted, and it receives its own secret data directly from the secure storage space on the TPM.

Besides offering secure storage, the Vault also ensures state continuity [24]. In particular, protection against two possible attacks is provided. First, in a rollback attack, an attacker passes a stale version of an SPM's stored data from disk to the Vault. Depending on the module's functionality, this may result in a security vulnerability, such as the reuse of cryptographic keys.

Second, the Vault should also provide crash resilience. As a compromised legacy kernel may allow an attacker to cause the system to crash, persistent storage of fresh data could be prevented based on subtle timing differences. This essentially enables an attacker to prevent the system from making progress.

Our prototype implementation does not yet implement state continuity guarantees, but the techniques proposed by Parno et al. [24] are also applicable to Fides: each SPM could request persistent storage of service requests before they are handled. The Vault in turn, keeps a request history for each module. The Vault thus acts as an intermediate module that stores state information of other modules. This reduces the number of modules that require storage for state-continuity on the TPM chip to one, limiting the wear on NVRAM.

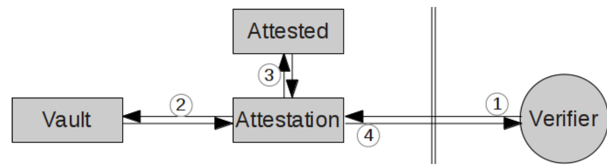


Figure 8: Using Fides' access control model and fast local communication, attestation can be supported easily and transparently to *any* attested module.

B. REMOTE ATTESTATION

Fides' access control model and local communication mechanism can also be leveraged to attest correct execution of modules with two key characteristics. First, meaningful attestation can be provided to the remote party, called the verifier. Only a small TCB consisting of the Fides architecture, an attestation module, the Vault and the attested module(s), are included in the measurement. Second, the attestation is transparent: the correct execution of any module can be attested without *any* modification. This improves reusability of modules.

Attestation in Fides is based on a two-layered approach where each layer attests its correct execution. Due to page constraints, only a sketch of the mechanism is presented. It relies however on μTPMs presented by McCune et al. [20]. Interested readers are referred there.

At the lowest level, the TPM chip ensures the correct loading of Fides and boots trust on the next layer. To achieve this, PCR registers 17, 18 and 19 are extended with a measurement of Fides, the security report of an attestation module and the Attestation Identity Key (AIK_{SPM}) respectively.

At the second level, *attestation modules* provide an attestation service and implement PCR extend and quote functionality similar to a hardware TPM chip. This prevents hardware PCR registers from being cluttered. As several identical attestation modules can also be loaded in the system, the number of SPMs that can be attested at the same time is virtually unlimited.

Figure 8 displays how the correct execution of an *Attested* SPM can be proven. First, the verifier provides two nonces n_1 and n_2 and an attestation module is created. Next, the attestation module extends its measurement with the Vault and requests its AIK_{SPM} key. Similarly the attested module is measured and contacted in step 3. Finally, the attestation module extends its measurement with the received result and signs it together with n_2. A similar request is sent to the lower level with n_1, but is only granted when the request is made from a module compliant with the measured security report in PCR 18. In step 4, both quotes are sent to the verifier.

In case the attested SPM calls other SPMs, the verifier is able to rely on the authenticated communication mechanism to ensure that no untrusted SPMs are used in the computation of the result. Alternatively, attestation-aware SPMs could notify the attestation module which SPMs are used.