Logical Attestation: An Authorization Architecture for Trustworthy Computing

Emin Gün Sirer Willem de Bruijn[†] Patrick Reynolds[‡]
Alan Shieh Kevin Walsh Dan Williams Fred B. Schneider
Computer Science Department, Cornell University [†]Google, Inc [‡]BlueStripe Software
{egs,wdb,reynolds,ashieh,kwalsh,djwill,fbs}@cs.cornell.edu

ABSTRACT

This paper describes the design and implementation of a new operating system authorization architecture to support trustworthy computing. Called *logical attestation*, this architecture provides a sound framework for reasoning about run time behavior of applications. Logical attestation is based on attributable, unforgeable statements about program properties, expressed in a logic. These statements are suitable for mechanical processing, proof construction, and verification; they can serve as credentials, support authorization based on expressive authorization policies, and enable remote principals to trust software components without restricting the local user's choice of binary implementations.

We have implemented logical attestation in a new operating system called the Nexus. The Nexus executes natively on x86 platforms equipped with secure coprocessors. It supports both native Linux applications and uses logical attestation to support new trustworthy-computing applications. When deployed on a trustworthy cloud-computing stack, logical attestation is efficient, achieves high-performance, and can run applications that provide qualitative guarantees not possible with existing modes of attestation.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Security and Protection

General Terms

Trusted Platform Module, Logic, Credentials-Based Authorization

1. Introduction

Secure coprocessors, such as industry standard Trusted Platform Modules (TPMs), are becoming ubiquitous. This hardware can provide a foundation for software systems that offer strong guarantees about run time behavior. Yet, there is a big semantic gap between the primitives provided by TPMs and what assurance secure applications actually require. The key primitive provided by secure coprocessors is *hash-based attestation*, whereby the platform generates a certificate that captures the binary launch-time hash of all components comprising the software stack. To identify trustworthy software configurations through their hashes necessitates software whitelisting, and that can restrict users to a limited set of applications due to platform lock-down [52]. Further, software certificates that divulge hashes compromise privacy [40]. Finally, launch-time program hashes do not adequately characterize programs whose behavior depends on inputs or external data. Much of the public backlash against trusted computing can, in fact, be traced to limitations of hash-based attestation.

Hash-based attestation forces all trust decisions to be *axiomatic*, because principals are trusted by fiat. Access control lists that enumerate principals by name, digital signatures to certify that a particular piece of code was vetted by a particular vendor, and authorization based on program hashes are all instances of the axiomatic basis for trust.

An alternative method of establishing trust is to employ an *analysis* that predicts whether certain behaviors by a program are possible. Proof carrying code [35], in which a program is accompanied by a proof that its execution satisfies certain properties, instantiates this analytical basis for trust. Similarly, systems that employ typecheckers and domain-specific languages, in which code snippets are loaded and executed only if the code is deemed safe, are employing analysis for establishing trust.

Finally, a *synthetic* basis for trust is involved when a program is transformed prior to execution and the transformation produces an artifact that can be trusted in ways that the original could not. Sandboxing [16], SFI [54], inlined reference monitors [11, 50], and other program rewriting techniques employ a synthetic basis for trust.

Today's operating systems provide disparate, ad hoc mechanisms to implement these three bases of trust. A unifying authorization architecture that can support all under the same rubric has not been undertaken. Moreover, establishing trust in practical settings often relies on a combination of these bases. For instance, a JVM enforces type correctness through both a static typechecker (an analytic basis) and code generation that adds run-time checks (a synthetic basis). The challenge, then, is to build an *authorization infrastructure*, by which we mean a system for generating, managing, and checking the credentials of principals in a computer system, that incorporates all three bases for trust. Our experience in designing such a unifying authorization architecture, implementing it in an operating system, and building system services to enable its use is the subject of this paper.

We propose a new authorization architecture, called *logical attestation*, that supports all three bases for trust. In logical attestation, a *labeling function* is used to generate an attributed statement called a *label* and expressed in a constructive logic of beliefs. Labels are unforgeable, machine-parseable statements of the form "*LF* `says` *S*" that capture information relevant to trust decisions. A

SOSP '11, October 23-26, 2011, Cascais, Portugal.

bitstring that encodes a label is known as a *credential*. Since labeling functions can be provided by third parties and labels are logical statements, a rich set of properties can be available for logical attestation. These properties can incorporate references to dynamic system state, including the current time, current resource availability, and even history. Labels used in proofs demonstrate, through logical inference, reasons why a principal should be trusted; they are consumed by *guards* that verify proofs to make authorization decisions.

We have implemented a new operating system, called Nexus, designed around logical attestation. Nexus executes on x86 platforms equipped with a TPM, supports much of the Posix API, and natively executes many Linux applications. To our knowledge, Nexus is the first operating system to implement logic-based authorization with dynamic system state, the first to implement operating system capabilities [7] based on statements issued by a TPM, and first to support all three bases for trust in a single unified framework. Logical attestation enables novel authorization functionality, as we illustrate, and provides strong and useful guarantees today's systems cannot provide.

We illustrate the power of our new authorization architecture by implementing a cloud computing application, called Fauxbook, that implements guarantees about safety, confidentiality, and resource control. Fauxbook provides a familiar social networking experience, where users publicly post and exchange status messages. Even Fauxbook developers are blocked, by our authorization architecture, from examining or data-mining the information Fauxbook handles. Moreover, logical attestation enables the cloud-infrastructure operator to guarantees certain forms of resource availability to Fauxbook developers. Experiments show that the cost of authentication with logical attestation in Fauxbook is on the order of 1ms, and it can be reduced to 20 cycles with proof caching, an optimization we describe later.

The rest of this paper is structured as follows. The next section describes the elements of logical attestation. Section 3 discusses operating system services required to support expressive, flexible logical attestation. Section 4 describes Fauxbook. Section 5 evaluates Nexus in terms of authorization efficiency, Section 6 reviews related work, and Section 7 summarizes and concludes.

2. Logical Attestation

Logical attestation is based on the generation, communication and use of attributable property descriptions represented as logical formulas. It builds on much past work that uses logical inference for authorization, known as credentials-based authorization (CBA) [57, 2, 26, 1].

The key idea in credentials-based authorization is that each request is accompanied by credentials, which are statements that can be attributed to principals. Accesses to resources are protected by a guard, a reference monitor that enforces a resource-specific authorization policy. The guard allows a request to proceed if credentials are available that imply a *goal statement* embodying the requisite authorization policy.

Credentials-based authorization provides for better expressiveness than traditional access control mechanisms. For instance, whereas Unix file systems perform access control based only on owner/-group/other permissions, a CBA system might enable a file containing an expense report to be restricted to, say, "*users who have successfully completed accounting training*," where a user can acquire such a credential by successfully completing an online course. Users of such a system need not contact an administrator in order to be placed in a special user-group. So, CBA enables decision-making authority to be removed from the guard (which now consists solely of a general-purpose proof-checker), and relocated to (potentially remote) unprivileged credential-granting entities, better suited for the task. CBA's flexibility provides clients with the ability to select a convenient way of discharging an access control policy. For instance, a CBA policy that limits access to "*a user whose identity is vetted by any two of: a stored password service, a retinal scan service, and an identity certificate stored on a USB dongle*" provides the client with the freedom to pick the most convenient method for gaining access [23]. Note that CBA credentials are self-documenting—they include all of the evidence used to reach a conclusion, a feature well-suited for logging and auditing.

Yet implementing credentials-based authorization in a real system poses significant challenges. First, there is the semantic gap between credentials and the actual state an operating system embodies. For instance, the seemingly innocuous credential "*Filesystem says User A is consuming less than 80% of her quota*" illustrates two fundamental problems: (1) the statement may become invalid even as the credential continues to be used in authorization decisions, and (2) a badly-implemented filesystem could issue credentials attesting to conflicting statements (e.g., in the case of statements issued before and after the the user exceeds 80% of her quota) that together imply false. Past work in CBA has tried to bridge this semantic gap between logic and OS state, either by limiting credentials to conveying irrevocable truths, or by replicating real world facts in logic variables (which creates inconsistencies stemming from the duplication of state).

The second set of challenges relates to the generation, manipulation, and management of credentials. To reap the benefits of CBA, an operating system must provide mechanisms for capturing relevant credentials. Specifically, the OS must support general-purpose mechanisms for performing analysis as well as synthesis. It must enable an application that possesses a particular property to acquire relevant credentials that can be forwarded and trusted by remote parties.

The final set of challenges relate to performance. The performance overhead of the mechanisms required for checking credentials and validating proofs of goals from credentials must not be prohibitive. And the performance impact of supporting the analysis and synthesis mechanisms must be small.

The rest of this section describes the mechanisms and abstractions supported by the Nexus operating system to address these challenges.

2.1 Logic Labels and NAL

Logical attestation bases all authorization decisions on *labels*. A label is a logical formula P `says` S that attributes some statement S to a principal P. Labels are expressed in Nexus Authorization Logic (NAL). The design of the logic is discussed in detail elsewhere [45]; here, we summarize why NAL is suitable for use in an OS setting.

First, to preserve justification in all authorization decisions, NAL is a constructive logic—a logic that restricts deduction to formulas that are derived solely from facts observed by a witness. In such a logic, tautologies such as double negation elimination ($\neg\neg p \Rightarrow p$) are not axioms. Where classical logics preserve only the truth of statements, proofs in constructive logics leave an audit trail for their inferences, making them well suited to reasoning about authorization.

Second, NAL is a logic of belief. Its formulas attribute facts and deductions to individual principals. Each NAL principal has a *worldview*, a set of formulas that principal believes to hold. The NAL formula P `says` S is interpreted to mean: S is in the worldview of P. All deduction in NAL is local. So we can derive from A `says` *false* the statement A `says` G for any G, but A `says` *false* cannot be used to derive B `says` G in NAL if B and A are unrelated

principals. This local inference property limits the damage an untrustworthy principal can cause. It also enables each Nexus application independently to specify which entities it trusts; the system does not require a superuser, a shared set of privileged principals, or an absolute universal frame of reference.

Finally, NAL supports group principals and subprincipals, as well as a `speaksfor` operator for characterizing delegation between principals. If *A* `speaksfor` *B* holds and *A* `says` *S*, then *B* `says` *S* for all statements *S*. Semantically, if *A* `speaksfor` *B* holds, then the worldview of *A* is a subset of the worldview of *B*. A subprincipal $A.\tau$ of *A*, by definition, satisfies *A* `speaksfor` $A.\tau$. This allows NAL to characterize dependencies between OS abstractions. For example, processes implemented by a kernel are all subprincipals of the kernel, which itself is a subprincipal of the hardware platform it executes on. So, strictly speaking, we should be writing *HW.kernel.process23* as the principal to which a statement by *process 23* would be attributed. For clarity, we elide the the prefix of dependencies in a principal's name whenever that prefix would be clear from the context. The NAL `speaksfor` operator optionally supports an "`on`" modifier that can restrict the scope of the delegation. For example, *Server* `says` *NTP* `speaksfor` *Server* `on` *TimeNow* delegates to *NTP* authority on statements for *Server* involving the time, but does not attribute to *Server* any other utterances by *NTP*.

The rest of this section traces label creation and usage for a time-sensitive content scenario. Here, a file on local disk is assumed to contain sensitive information that should be accessed before a fixed date. The contents of this file must not be overtly leaked over channels to the disk or network.

2.2 Label Creation

Labels are created in Nexus by invoking the **say** system call. This system call takes a string argument that encodes a NAL statement. Nexus imposes no semantic restrictions on the terms and predicates appearing in a statement. For instance, in the label *TypeChecker* `says` *isTypeSafe(PGM)*, *isTypeSafe* is a predicate introduced by the *TypeChecker*, whose meaning is presumed to be understood by a principal that imports this statement into its worldview. This flexibility enables third parties to define types of credentials that may not have been envisioned by the OS designers. Moreover, because all reasoning is local to a principal, separate applications need not subscribe to a common nomenclature or semantics for labels; for instance, the predicate *isTypeSafe* might be used by both a JVM and CLR, but denote different properties.

In our time-sensitive file example, the process that wants to read the file must acquire some credential certifying that its execution will not leak the sensitive information to the disk or network. One potential approach may use labels like:

Company `says` *isTrustworthy(***Client***)*
∧ *Nexus* `says` */proc/ipd/12* `speaksfor` **Client**

Here, **Client** is the well-known SHA1 hash of a program that some third party *Company* has certified to exhibit the properties sought. The second label, provided by the Nexus, indicates that the named process `speaksfor` its launch-time hash.

An alternative set of labels to accomplish the same task but without the disadvantages of axiomatic trust is:

Nexus `says` */proc/ipd/30* `speaksfor` **IPCAnalyzer**
∧ */proc/ipd/30* `says` ¬*hasPath(/proc/ipd/12,* **Filesystem***)*
∧ */proc/ipd/30* `says` ¬*hasPath(/proc/ipd/12,* **Nameserver***)*

Here, a separate program IPCAnalyzer, running as process 30, used an analytic basis to enumerate the transitive IPC connection graph using the Nexus introspection interface (for simplicity, we have replaced process identifiers with the strings **Filesystem** and **Nameserver**—hashes, signatures, or other labels may be used to relate these process identifiers to known principals). Since the disk and network drivers in Nexus operate in user space and rely on IPC for communication, a transitive IPC connection graph that has no links to these drivers demonstrates that there is no existing channel to the disk or network.

2.3 Labelstores

A simple way to implement labels is to use digital signatures. A process that controls a key stored in the TPM can direct the TPM to create such a signed credential. Alternatively, a process that stores a key in its address space can digitally sign statements; this is sensible only when the key resides in an address space (called an *isolated protection domain* (IPD) in Nexus terminology) that is not shared. Such credentials are sometimes used in Nexus, but because cryptographic operations are expensive, whether performed by TPM hardware or in software, the kernel provides a new abstraction that helps eliminate their overhead.

The *labelstore* is implemented by the Nexus for storing labels generated by user programs. A user process can issue labels by invoking the **say** system call, passing in a NAL statement *S*, and naming the labelstore into which the statement should be placed. The returned *handle* can be used to request manipulation of the label. The speaker *P* associated with the label is, by default, the name of the process that invoked the **say** system call. Once in a labelstore, labels can be transferred between labelstores, externalized into a standard cryptographically-signed certificate format (X.509), imported from that format into a labelstore, and deleted.

Since labels are generated directly by invoking the **say** system call, there exists a secure channel from the user program to the operating system during label creation. The presence of this channel obviates the need for cryptographic operations. In our time-sensitive file example, the labels mentioned above are emitted directly into the labelstore and stored as strings, without any costly cryptographic signatures.

2.4 Label Communication

A label *P says S* that is cryptographically signed is not vulnerable to misattribution and misquotation, but is computationally expensive to generate and verify. So the Nexus allows principals to exchange labels efficiently over secure system channels. Specifically, the kernel exposes IPC channels, and it authoritatively binds IPC ports to owning processes by producing a label *Nexus* `says` *IPC.x* `speaksfor` */proc/ipd/process.y*.

Externalized labels must convey context about principals named in labels. All Nexus principals are subprincipals of the TPM's secret key EK, associated permanently with that TPM at the time of manufacture. On first boot, the Nexus kernel uses the TPM to generate a Nexus key NK that is bound to the current contents of the TPM's platform configuration registers (PCRs) at boot time. NK serves as a principal associated with that version of the Nexus. And an attacker that boots a modified version of the kernel in order to access this NK will be unable to gain access to the private key due to the PCR mismatch. The kernel also generates a Nexus boot key (NBK) that identifies the unique boot instantiation of that Nexus installation. All processes are sub-principals of NK concatenated with the hash of the public component of the NBK, are named in full in X.509, signed with the NK, and are accompanied by another X.509 certificate attesting to NK using the TPM. So, when a label is externalized into X.509 format, the exported statements are, informally, of the form *"TPM says kernel says labelstore says processid says S."* In cases where TPMs are accompanied by cer-

tificates from the TPM manufacturer attesting to the chip and the PC hardware manufacturer attesting to the chip's integration on the motherboard, Nexus can furnish these to establish trust in the complete hardware platform; support for such certificates is currently vendor-dependent.

2.5 Goal Formulas

Nexus enables a *goal formula* to be associated with any operation on any system resource. The goal formula specifies what must be discharged for a client to be authorized to perform the given operation. A **setgoal** system call specifies the resource (e.g. process, thread, memory map, page, IPC port, files and directories), the operation on that resource, a goal formula and, optionally, the IPC channel to a designated guard. Following a successful **setgoal** call, all subsequent operations on that resource are vectored to the designated guard, which checks client-supplied labels against the specified goal formula.

Goal formulas, like labels, are expressed in NAL. A goal formula documents its trust assumptions by specifying `speaksfor` relationships in a preamble. For instance, a goal formula of the form "*Owner* `says` *TimeNow* < *Mar19*" requires the client to obtain a credential from a time server trusted by the file owner that attests that the deadline has not yet passed.[1] Specifically, this goal formula can be discharged by acquiring credentials:

Filesystem `says` *NTP* `speaksfor` *Filesystem* `on` *TimeNow*
∧ *NTP* `says` *TimeNow* < *Mar19*

Setting a goal formula is itself an operation that must be authorized. A goal formula that is separate (and often, distinct) from the one used for object access is involved. Typically, a goal formula for a shared object (e.g. a nameserver or a mail spool directory) will permit a wide range of principals to perform regular operations on the object, while restricting the **setgoal** call to an entity privileged enough to own or maintain that resource.[2]

For our time-sensitive file example, a suitable goal formula that embodies the desired policy is:

Owner `says` *TimeNow* < *Mar19*
∧ $\mathcal{X}$ `says` *openFile(filename)*
∧ *SafetyCertifier* `says` *safe*($\mathcal{X}$)

Here, calligraphic font is used for a class of identifiers that are instantiated for guard evaluation.

Typically, Nexus goal formulas involve the owner of a resource explicitly permitting an operation. Labels issued to the entity seeking access are used in conjunction with auxiliary labels issued by the resource owner to demonstrate that the conditions for access are satisfied. For instance, the *SafetyCertifier* above might examine the labelstore and issue additional labels of the form:

SafetyCertifier `says` *safe*($\mathcal{X}$)

for each IPD $\mathcal{X}$ when the following labels are also found in the labelstore:

Nexus `says` $\mathcal{Z}$ `speaksfor` **IPCAnalyzer**
∧ $\mathcal{Z}$ `says` ¬*hasPath*($\mathcal{X}$, **Filesystem**)
∧ $\mathcal{Z}$ `says` ¬*hasPath*($\mathcal{X}$, **Nameserver**)

[1] We discuss the details of handling credentials that refer to mutable state in Section 2.7.

[2] It is technically possible, in our current implementation, for a bad applicaton to set goal formulas on a resource that prohibit every entity, including itself, from ever interacting with that resource. This feature is the inevitable consequence of the lack of a superuser. If desired, one can modify the kernel to always permit operations by a designated superuser process or one of its delegates.

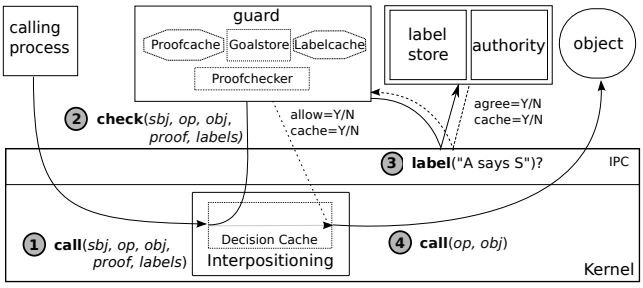

Figure 1: Logical Attestation in Nexus: To perform an operation on an object, an access control subject (1) passes a proof and set of labels that satisfy the associated goal formula. The kernel calls a guard process (2), that evaluates the proof and verifies authenticity of labels, referring (3) to external labelstores and authorities for unknown labels. The call is permitted to proceed (4) if the proof discharges the goal.

This is equivalent to the definition:

safe($\mathcal{X}$) ≜ ¬*hasPath*($\mathcal{X}$, **Filesystem**) ∧
¬*hasPath*($\mathcal{X}$, **Nameserver**)

and an alternative would have been simply to include this definition in the guard.

2.6 Guards

Authorization decisions are determined by performing a logical inference that derives a goal formula from a set of credentials.

Since proof derivation in logics like NAL is undecidable, the Nexus places the onus on the client to construct a proof and present it when invoking an operation on an object. The guard need only check the proof and authenticity of credentials, both of which are tractable problems and, as we will demonstrate later, cheap. Figure 1 depicts all of the steps in the authorization process.

Kernel resources implemented by the Nexus, including threads, IPDs, IPC channels, network sockets, files and directories, are managed by a kernel-designated guard. The choice of default policy for such resources is tricky, since one has to ensure that a nascent object for which a goal policy has not yet been established is protected from access by unauthorized parties. The kernel-designated guard implements a simple default policy to solve this bootstrapping problem: it interprets the absence of a goal formula as the policy *resource-manager.object* `says` *operation*, which is only satisfiable by the object or its superprincipal, the resource manager that created the object. To pass object ownership to a third party, the resource manager issues a label *resource-manager* `says` *third-party* `speaksfor` *object*. For instance, when */proc/ipd/6* creates a file called */dir/file*, the fileserver *FS* creates the file on behalf of */proc/ipd/6* and deposits the label *FS* `says` */proc/ipd/6* `speaksfor` *FS./dir/file* in the labelstore for that process.

2.7 State and Authorities

A trustworthy principal should refrain from producing statements in transferable form if these statements might subsequently become invalid. For instance, a statement by *NTP* that assures the bearer of the current time would quickly expire, causing others to conclude that *NTP* is an untrustworthy principal. Nevertheless, realistic authorization policies often refer to non-monotonic dynamic state, such as user input and resource utilization. Therefore, logical attestation supports an *authority* abstraction for querying dynamic state without incurring the problems of invalidated credentials.

Nexus authorities attest to the veracity of a label only when asked, and they never issue credentials that are both transferable and can

become invalidated. Specifically, an authority is implemented by a process listening on an attested IPC port i. That process authoritatively answers whether it currently believes statement *IPC.i* `says` *S* holds; its answer can only be observed by the principal posing the query. The default labels Nexus provides for IPC channels thus enable such a statement to be attributed to the authority process. So, over an attested IPC channel, the Nexus implements the following protocol: a guard that wants to validate a label sends the label to the port. The process listening on the port returns a binary answer that is authoritative (by virtue of the IPC channel), thus conveying the validity but not in a way that can be stored or further communicated. For example, in our time-sensitive file application, a trustworthy system clock service would refuse to sign labels, but would subscribe to a small set of arithmetic statements related to time, such as *NTP* `says` *TimeNow* $\leq$ *March 19*. The guard process can establish the veracity of such a claim by querying the system clock service on each time-dependent check.

By partitioning trusted statements into indefinitely cacheable labels and untransferable yes/no responses from authorities, we obviate the need in Nexus for an additional, system-provided revocation infrastructure. For instance, a software developer *A* wishing to implement her own revocation check for a statement *S* can, instead of issuing the label *A* `says` *S*, issue *A* `says` *Valid(S)* $\Rightarrow$ *S*. This design enables third-parties to implement the revocation service as an authority to the statement *A* `says` *Valid(S)*.

2.8 The Decision Cache

Since guard invocations are expensive, the overhead entailed by credential checks needs to be reduced whenever possible. To this end, the Nexus implements a cache in the kernel that stores previously observed guard decisions, called the *decision cache*.

The goal of the decision cache is to avoid expensive proof checking and validation operations when they are unnecessary. To support the decision cache, the guard-kernel interface is amended with a bit to signify whether a validation is cacheable. NAL's structure makes it easy to mechanically and conservatively determine those proofs that do not have references to dynamic system state and, thus, are safe to cache.

The decision cache is implemented as a hashtable indexed by the access control tuple of subject, operation, and object. Because the cache is a performance optimization, its contents can be marked invalid and the cache can be resized at runtime.

Authorization decisions are invalidated as a system executes. When a process updates a goal or proof, the kernel must invalidate corresponding entries in its decision cache. The kernel therefore interposes on the guard control IPC to monitor updates. On a proof update, the kernel clears a single entry in the decision cache. A **setgoal** operation, on the other hand, might affect many entries that, due to hashing, may spread across the memory implementing the decision cache. To avoid clearing the whole decision cache on each such update, the hash function we use was designed to hash all entries with the same operation and object into the same subregion. Subregion size is a configurable parameter that trades-off invalidation cost to collision rate. Only when the kernel has no cached decision, does it consult an IPC port lookup table and make an upcall to a guard process.

The decision cache has a significant impact on performance. As we discuss later, the decision cache can reduce proof checking latency on a minimal system call from 2100% of unguarded invocation latency down to 3%.

2.9 Guard Cache

To amortize proof-checking cost across principals and invocations, guards internally cache as much proof-checking as possible. Caching valid credentials cannot cause a vulnerability, because labels are valid indefinitely. Even when proofs depend partially on dynamic state, they often have pieces that can be replaced by lemmas whose outcome may be cached safely. So, a cache in the guard can enable some parts of a proof to be checked quickly, reducing authorization to a few checks and subsequent consultation with designated authorities.

Since all information in the guard cache constitutes soft state and can be re-checked when needed, evictions from the guard cache cannot impact the correctness of access control decisions. To provide some measure of performance isolation between principals, the default Nexus guard, in response to a new request from a given principal, preferentially evicts cache entries from that same principal. To limit exhaustion attacks due to incessant spawning of new processes and thus principals, quotas are attached to the principal that is the root of an entire process tree.

3. Operating System Services

To effectively build secure applications using logical attestation, additional OS mechanisms are necessary.

3.1 Introspection

For supporting an analytic basis for trust, Nexus implements an extensible namespace through which principals can query the state of the kernel. Similar to Plan 9's `/proc` filesystem [38], this grey-box information service allows components to publish application-defined *key=value* bindings. Logically, each node in the introspection service is the same as a label *process.i* `says` *key = value*. These key-value pairs indicate kernel state information. Each process and the kernel are linked against an in-memory fileserver. The fileserver has default mechanisms for rendering data contained in hashtables, queues, and other kernel datastructures. Using these tools, the Nexus kernel exposes a live view of its mutable state, including lookup tables for processes, IPC ports, and guard ports. Applications, such as the Python interpreter, similarly publish their state information, such as the list of currently loaded modules and executing files.

Introspection of metadata offers a portable alternative to property attestation by unique hash. For instance, a labeling function can verify, by analyzing information exported through introspection, that a language runtime is not executing unsafe code, that a device driver has its I/O mediated by a reference monitor, or that a keyboard driver detected physical keypresses. All of these, in fact, are used by the example applications in Section 4.

Two properties of NAL's term language are essential to meaningful label generation. First, the uniform naming scheme provided by the introspection service enables labeling functions to identify entities in the system in a portable manner. Second, the filesystem interface presents standard, well known, mechanisms for term access control and change notification. Associating goal formulas to information exported through the `/proc` filesystem enables the kernel to impose access control on sensitive kernel data.

3.2 Interpositioning

Not all properties are fully analyzable prior to execution. But even if it might be a priori undecidable to determine whether an application will open a particular file or execute a particular code path, it could still be trivial to monitor and detect such behavior dynamically. In this case, a synthetic basis for trust is achieved by actively interposing on all I/O of an untrusted process and transforming that I/O into safe actions, in effect rendering the untrusted process trustworthy.

Nexus provides an interpositioning service by which a reference monitor can be configured to intercept IPC operations originated by a particular process. Specifically, the **interpose** system call provides a way for a particular process to bind itself to a given IPC channel. As with every Nexus system call, an **interpose** call only succeeds if the reference monitor can satisfy some goal formula, typically by presenting a credential obtained from the process to be monitored. Thus, interposition is subject to consent, but a reference monitor, once installed, has access to each monitored IPC call. The reference monitor can inspect and modify IPC arguments and results, and at its discretion, block the IPC. Since all system calls in Nexus go through the IPC interface, a reference monitor can inspect, modify and block *all* interaction of a process with its surrounding environment, to the extent it is permitted to do so.

Interposition is implemented by using an redirector table in the kernel. Upon an IPC invocation, the kernel consults the redirector and reroutes the call to the interceptor, passing the subject, operation, and object. The interceptor can access the arguments passed to the intercepted call by issuing further system calls, and it may also modify them if it has credentials to do so. On completion, the monitor notifies the kernel about whether the call should be permitted to continue. If the call does continue as normal, then the kernel will later make an upcall when the return for that IPC occurs, so that the interceptor can modify response parameters.

Interpositioning is a composable operation. Multiple processes can be interpositioned on a given IPC channel, and the interposition system call itself can be monitored by an interposition agent.

3.3 Attested Storage

Data confidentiality and integrity can be important in many high-integrity applications. Password authenticators, capability managers, and file systems often need to retain confidential data in a manner that prohibits unauthorized access, avoids replay attacks, and detects tampering. Although hardware attacks against the TPM is beyond the capabilities of most attackers, attacking the storage system while a given machine is powered down can be as trivial as duplicating and replaying a disk image. Yet it is infeasible to store all sensitive information on a TPM, because the TPM provides only a small amount of secure on-chip storage. To guard against attacks on the storage system, a trustworthy operating system needs to offer integrity and confidentiality protection, even across reboots. Nexus does this by providing an abstraction called *Secure Storage Regions* (SSRs) that enable the limited TPM storage resources to be multiplexed in a way that provides integrity- and confidentiality-protected, replay-proof, persistent storage.

SSRs create the illusion of an unlimited amount of secure storage that is backed by the TPM. This can be achieved with TPM v1.1, which provides only two 20-byte registers (called Data Integrity Registers, or DIRs) for storage, or with TPM v1.2, which provides only a finite amount of secure NVRAM. Each SSR is an integrity-protected and optionally encrypted data store on a secondary storage device. Applications can **create**, **read**, **write** and **destroy** their SSRs. SSRs can be used by applications to store arbitrary data that demands integrity, and optionally, confidentiality guarantees, such as authentication tokens, keys, cookies, and other persistent and sensitive information. And guards can use SSRs to store the state of security automata [44], which may include counters, expiration dates, and summary of past behaviors.

The integrity of an SSR is protected by a hash. When the number of SSRs is small relative to the amount of storage on the TPM, their hashes can be stored by the TPM. Thereafter, attempts to replay old values of SSRs, for instance, by re-imaging a disk, would fail, because the hash of the (replayed) SSR would not match the (modified, current) hash stored in the TPM.

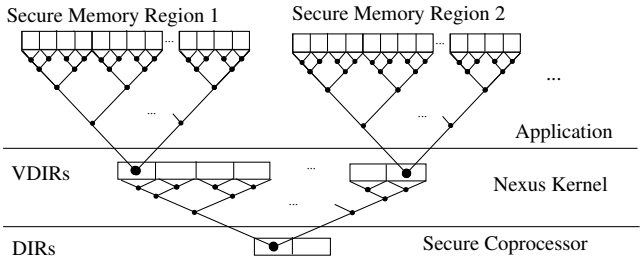

Figure 2: The SSR interface enables applications to provide integrity- and confidentiality-protection for data at rest.

In order to provide these same guarantees for an arbitrary number of SSRs, the Nexus utilizes a kernel-managed Merkle hash tree [33, 34] to store hashes of all SSRs. A Merkle hash tree divides a file into small blocks whose hashes form the leaves of a binary tree and, thus, somewhat decouples the hashing cost from the size of the file. Inner nodes in the tree store hashes computed by concatenating and hashing the values of their child nodes, resulting in a single root hash that protects the entire file. SSRs are implemented at user-level; they use a Nexus kernel abstraction, called Virtual Data Integrity Registers (VDIRs), to hold hashes. The Nexus kernel stores VDIR contents in a hash tree kept in memory and stored on secondary storage (between boots), with the root hash of the hash tree stored in the TPM. Any attempt to change or replay the contents of this tree by modifying the contents while it is dormant on a secondary storage device will be caught during boot through a mismatch of the root hash. Similarly, replay or modification attacks to an application's SSRs produces a mismatch against the hashes stored in the kernel-managed Merkle hash tree.

Writes to the storage system and to the TPM are not atomic, and a power failure may interrupt the system between or during these operations, so care must be taken when updating the kernel-managed Merkle hash tree. The Nexus uses an update protocol that can withstand asynchronous system shutdown and that requires no more than the two 160-bit hardware registers as provided by the TPM v1.1 standard. These TPM data integrity registers (DIRs), which we will call DIR_{cur} and DIR_{new} for clarity, are set up such that they cannot be accessed unless the state of certain platform configuration registers (PCRs) match a sequence that corresponds to the Nexus. The protocol additionally employs two state files `/proc/state/current` and `/proc/state/new` on disk to store the contents of the kernel hash tree.

When an application performs a write to a VDIR, the Nexus effectively creates an in-memory copy of the kernel hash tree and updates it to reflect the modification to that VDIR. It then follows a four step process to flush the contents to disk; namely: (1) write the new kernel hash tree to disk under `/proc/state/new`, (2) write the new root hash into DIR_{new}, (3) write the new root hash into DIR_{cur}, (4) write the kernel hash tree to `/proc/state/current`. A success indication is returned to the user application only after all four steps complete without failure.

On boot, the Nexus reads both state files, computes their hashes, and checks them against the contents of the two DIR registers. If only one of the DIR entries matches the corresponding file contents, the contents of the corresponding file are read and used to initialize all VDIRs; if both match, then `/proc/state/new` contains the latest state; and if neither matches, indicating that the on-disk storage was modified while the kernel was dormant, the Nexus boot is aborted. This sequence ensures that the current copy of the VDIR contents always resides on disk and can be located even in

the presence of machine failures that may leave the files or the DIR registers in an undefined state. If it is desirable to protect against failures of the secondary storage device that may affect the files at rest, then more copies could be made at steps (2) and (3).

Nexus provides confidentiality guarantees for SSRs using kernel abstractions called Virtual Keys (VKEYs). Whereas VDIRs provide a mechanism to virtualize the limited data integrity storage on the TPM, VKEYs virtualize the limited encryption key storage. The VKEY interface provides methods for creating, destroying, externalizing, and internalizing key material, in addition to standard cryptographic operations suited for the type of key. VKEYs are stored in protected memory in the kernel. During externalization, a VKEY can optionally be encrypted with another VKEY to which a program has access; the Nexus uses the TPM to generate a default Nexus key, and sets it up to be accessible only to a kernel whose platform configuration registers (PCRs) match those for the Nexus. Specifically, SSRs use a symmetric counter-mode AES block cipher on the data blocks that comprise an SSR. Counter-mode encryption allows regions of files to be encrypted independently, decoupling operation time from the size of the file. Unlike most block-chaining modes, a counter-mode ciphertext block does not depend on the its predecessor, obviating the need to recalculate all successor ciphertexts after an update to some plaintext block. This mechanism allows Nexus to retrieve and verify only the relevant blocks from the filesystem, and it enables demand paging for reading data blocks in an SSR.

Since all operations on VKEYs and VDIRs can be protected using logical attestation, complex policies about the uses of cryptographic keys and stored data are easy to express. Group signatures, for instance, can be implemented by creating a VKEY and setting an appropriate goal formula on the **sign** operation that can be discharged by members of the group. Further, by associating a different goal formula with the **externalize** operation, an application can separate the group of programs that can sign for the group from those that perform key management and transfer keys for the group. Similarly, goal formulas can be used to limit access to VDIRs and the corresponding SSRs. For instance, an SSR that holds sensitive data subject to policy controls such as Sorbanes-Oxley, HIPAA, and the like, can be restricted for access solely to those applications that have been certified, analyzed, or synthesized by appropriate authorities to uphold that policy.

3.4 Nexus Boot

Integrating the additional kernel abstractions and mechanisms described above with the TPM requires only modest modifications to the boot sequence. On power-up, the TPM initializes its platform configuration registers (PCR) to known values, and the system BIOS extends PCRs with a firmware hash and the firmware extends PCRs with the boot loader hash. A trusted boot loader extends PCRs with a hash over the Nexus kernel image. This provides a measurement over the entire kernel image; it forms a static root of trust for the Nexus kernel. We adopted this simple approach for establishing the root of trust because it makes minimal assumptions about the processor hardware; recent hardware extensions to support a dynamic root of trust, which can reduce TCB size even further by dynamically providing a safe execution environment for snippets of code, could also have been employed.

After the kernel is loaded, the Nexus boot sequence attempts to initialize the TPM to a known good state and to recover the kernel's internal state. If this is the first Nexus boot, then the Nexus forces the TPM to generate a new Storage Root Key (SRK) associated with the current PCR state by taking ownership of the coprocessor. If this is not the first boot, then it uses the protocol described in the preceding section to decrypt the VDIR and VKEY contents.

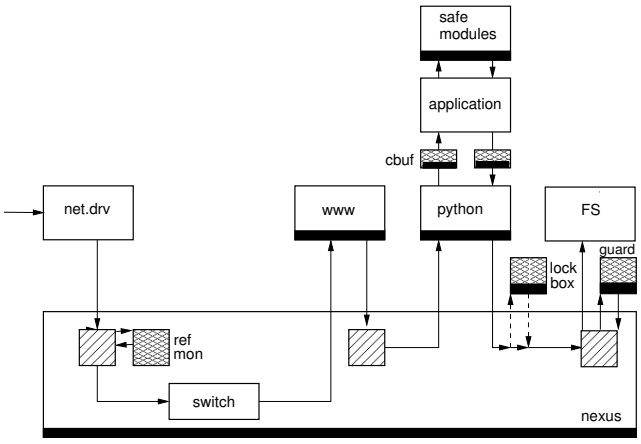

Figure 3: The Fauxbook multi-tier webserver. Cross-hatched boxes denote enforcement mechanisms, shaded boxes identify decision caches and thick borders indicate hash attestation.

This is performed through a TPM unseal operation that depends on the SRK and the PCR values. Consequently, the Nexus state can only be restored by the kernel that initially took ownership; attempts to boot a modified kernel to acquire access to the SRK and, consequently, decrypt the VDIR and VKEY contents, are prevents because a modified kernel that permits such unsafe operations will lead to different PCR values.

An early version of the Nexus kernel investigated mechanisms for acquiring a privacy-preserving kernel key from a Nexus Privacy Authority that can be used in lieu of TPM-based keys, and therefore mask the precise identity of the TPM. Techniques for implementing such privacy authorities (also known as trust brokers) are well-established and could be employed in settings where the identity of the TPM cannot be revealed.

4. Applications

The combination of operating system level interpositioning, introspection, and logical attestation enables the Nexus to support applications that enforce guarantees about state and history, and prove these guarantees to third parties.

4.1 Fauxbook

To illustrate these capabilities, we implemented a privacy-preserving social network application called Fauxbook. Unlike many, Fauxbook enforces user-defined policies on data dissemination. Fauxbook is a three-tier web service built from a standard Lighttpd webserver, Python application server with SQLite, and Posix filesystem. Figure 3 depicts how a web request flows through the system.

Logical attestation enables Fauxbook to provide novel guarantees. There are three kinds of entities in cloud computing environments, each demanding different guarantees: cloud providers who operate the cloud infrastructure and house tenants; developers who deploy applications on the cloud infrastructure and act as tenants to cloud providers; and users of the deployed applications. To the cloud provider, it guarantees that developers remain confined to a sandbox, obviating the need to use virtual machines or other mechanisms for isolation. To developers, Fauxbook guarantees that the underlying cloud provider actually provides levels of resources contracted. And to ordinary users, Fauxbook ensures that data shared with friends in the Fauxbook social network is protected from inspection, datamining, and display to unauthorized third-parties, including to Fauxbook developers themselves. Faux-

book implements these guarantees by combining axiomatic, analytical, and synthetic bases for trust.

Resource Attestation. Fauxbook employs logical attestation on kernel resources in order to guarantee that cloud providers deliver agreed-upon levels of service to the Fauxbook application. Oversubscription is a long-standing problem in shared environments, like the cloud. Clients might contract for some desired level of service from the underlying platform, but conventional systems provide few mechanisms for enforcing such service-level agreements (SLAs). Where SLAs involve externally-observable performance metrics, such as latency or throughput, enforcement typically requires continuous end-to-end measurement. And there is a class of SLAs, pertaining to the availability of reserved resources, such as backup links in the network, that are difficult or impossible to measure from an application's vantage point; such service must be evaluated using exogenous, ad hoc mechanisms, such as reputation and feedback measures on web forums. All of these approaches are costly, difficult, and incomplete.

If the cloud provider executes the cloud platform on top of the Nexus, a labeling function can examine the internal state of resource allocators in the kernel. Labels can then vouch for reservations of service. For instance, we implemented a proportional-share CPU scheduler that maintains a list of all active clients, which it exports through the introspection interface. A file in each tenant's directory stores the weight assigned to that tenant, while goal statements ensure that file is not readable by other tenants. Therefore, a labeling function that measures the resource reservations from each of the hosts on which the tenant code is deployed ensures that the tenant receives an agreed-upon fraction of the CPU.

Safety Guarantees. Fauxbook uses logical attestation to guarantee that tenant code will remain confined to a sandbox, thus obviating the need for other, potentially expensive, isolation mechanisms. A labeling function uses analysis and synthesis to ensure that mutually distrusting tenant applications can be executed safely within the same address space. Specifically, the labeling function performs static analysis to ensure that tenant applications are legal Python and that tenants import only a limited set of Python libraries. This restriction, by itself, is not sufficient to achieve the desired level of isolation, because Python provides rich reflection mechanisms that could be used to access the import function provided by the language. That import function could in turn allow a rogue application to invoke arbitrary (and potentially unsafe) code. To defend against this attack, a second labeling function rewrites every reflection-related call such that it will not invoke the import function. The two labeling functions in combination ensure that the resulting tenant application can only invoke a constrained set of legal Python instructions and libraries.

Confidentiality Guarantees. Fauxbook employs logical attestation to guarantee users that their data is shared only with people they have authorized through the social networking service. Even Fauxbook developers are blocked from accessing users' shared content; developers are not able to examine, data-mine, or misappropriate user data, even though, as developers, they are intimately involved in manipulating and servicing such data. These guarantees were motivated by well-publicized incidents where developers of social networking systems abused their power [12].

The observation we leverage to protect users from developers is that, in this particular application, actions are, in a sense, data independent. Collecting personal information comprising status updates, photos and videos from users, storing this information for later use, and collating it for display to other users, does not require the ability to examine that data—it just requires copying and displaying it. Thus, Fauxbook treats user information as indistinguishable from opaque blobs, and the Fauxbook code is restricted to store, manipulate, and merge such blobs without seeing their content.

The central data structure in any social networking application is a social network graph. Nodes represent users, and edges correspond to friend relationships. Fauxbook guarantees that this graph (1) only contains edges explicitly authorized by the user, and (2) data flows from one node to another only if there is an edge between the two nodes. More complex authorization policies, perhaps involving friends-of-friends or various subsets of nodes belonging to different circles of friends, are also possible, but they do not raise substantially new security or implementation issues, so we concentrate on this simpler authorization policy for clarity.

Fauxbook attests to these two guarantees about the social network graph by presenting to its users a set of labels concerning properties of the software components that comprise the application. These properties are gleaned through analysis based on introspection and through synthesis based on interpositioning. Coupled with axiomatic trust in the Nexus kernel, the labels together demonstrate to Fauxbook users that their data will be handled in a way that achieves the properties discussed above.

Credentials conveying externalized forms of these labels currently reside at a public url in X.509 form (e.g. similar to where the privacy policy for a web application would be located) and can be queried by a user prior to signing-up with Fauxbook. An alternative implementation would involve transferring these certificates to the client during SSL connection setup; in addition to the traditional SSL certificate which binds a name to an IP address, these certificates would provide the groundwork for the client to be convinced that the software connected to the socket has certain properties that together uphold the desired privacy policy. In both cases, the guarantees about the behavior of tenant applications stem from trust in the infrastructure operated by the cloud provider, which in turn proscribes the behavior of the applications deployed in the cloud.

The operation of Fauxbook depends on several software components. A user-level device driver manages a network interface card. A `lighttpd` web server receives HTTP requests from users and dispatches them to URL handlers supported by a web framework, in addition to extracting the HTTP stream into TCP/IP packets (since networking is done in user level in the Nexus). The web framework provides the execution environment for web applications deployed in the cloud by developers. It provides libraries for user management and authenticating sessions, as well as the dispatch loop for generic applications. Finally, the Fauxbook application provides the logic required to implement a simple social network.

The cloud environment precludes certain simplifying assumptions. First, we cannot assume that application code is monolithic, public, or unchanging. If it were, then certifying its behavior would be a simple task of certifying its binary hash and making its code public. We instead expect that applications deployed in the cloud cannot have their source code made public and will change too frequently for their behavior to be manually certified. Second, we cannot assume a web framework having functionality that is specific to Fauxbook or any other particular application. A web framework typically is operated by the cloud provider and designed to support any generic application. Therefore, it cannot be tightly coupled with application code provided by third-party developers. Finally, we cannot assume that users possess unique cryptographic keys, because they don't.

The privacy guarantees of Fauxbook derive from the properties of each of the components involved in managing user data. Be-

low, we describe each of these components and how they guarantee these properties.

The network device driver needs to ensure that user data is delivered solely to its intended recipient. A driver that copies information from packets could potentially also exfiltrate authentication information, such as submitted passwords and returned authenticating cookies, to third parties who then could impersonate the user, as well as directly copying personal information from the user for use by others. Our device driver can demonstrate that it is unable to perform these actions. Like most user-level drivers, the Nexus NIC device drivers operate by allocating memory pages, granting these to the NIC, setting up DMA registers to point to these pages, and handling device interrupts. Unlike other device drivers, Nexus device drivers operate under control of a device driver reference monitor (DDRM) [56] that can constrain access to the device and to memory. So the driver provides the aforementioned assurance by demonstrating that it is operating in a DDRM with no read or write privileges for any of the pages the driver manages. In fact, the driver can perform the DMA setup and other device functionality without access to page contents, so it does not actually need that access. In addition, it operates under a second reference monitor that blocks all but a small set of systems calls governing I/O ports, memory, and IPC. In particular, the reference monitor only allows sending and receiving packets to and from a particular IPC channel connected to the web server process. In sum, the network driver provides labels, based on synthesis and provided by the reference monitor, certifying that the reference monitor only forwards unmodified data between network device and the web server, and that it cannot modify message contents, either by copying between sessions or by forcing transmission to untrustworthy hosts.

The trustworthiness of the web server rests on both axiomatic and synthetic bases for trust. The web server forwards packets from the device driver to the web framework, and vice versa. Unlike the device driver, the web server requires read/write access to data, because it must translate IP packets into HTTP requests and, subsequently, into FastCGI messages. This task requires only IPC-related system calls in addition to polling, synchronization, and memory allocation. To prove that it will not leak information to other entities, the web server relinquishes the right to execute all other system calls after initialization. And it provides labels that demonstrate that it lacks the ability to communicate with other processes besides the device driver and the web framework, and that it is bound by hash to a well known binary version of analyzable open source software.

The web framework provides guarantees (1) that it will provide libraries for creating, deleting, and authenticating users, (2) that user authentication information is stored in a file to which the web framework has exclusive access, (3) that it will dispatch the correct handler for each web application, and, most importantly, (4) it will constrain each such application to not leak user information except as authorized by the users. Since the web framework code is relatively static, the first three guarantees can be obtained through hash-based attestation. The fourth guarantee forms the critical link to the overall security properties of Fauxbook, since it means that, even though the application code is provided by Fauxbook developers and is tasked with storing and assembling the web pages a user sees when visiting the Fauxbook social networking site, the very same code is unable to parse and examine the contents of the data, such as status updates and images, submitted by users.

The web framework enforces guarantees by constraining Fauxbook application code to access user data through a restricted interface called *cobuf*, for *constrained buffers*. Cobufs enable untrusted applications to manipulate user-supplied data without allowing that data to be examined. A cobuf comprises a byte array that stores data and an identifier that identifies the principal owning that information; the result is an attributed buffer that may be used only for content-oblivious string manipulations. Applications running on the web framework can store, retrieve, concatenate, and slice cobufs but lack the ability to act on cobuf contents. Akin in functionality to homomorphic encryption, cobufs permit operations on data without revealing that data, but the functionality is achieved using low-overhead language-based access control techniques rather than by expensive crytography.

By design, cobufs are useful only for data-independent applications, yet much of the functionality of a social networking application is data-independent. Because the cobuf interface does not support data dependent branches, it is not Turing-complete—certain functionality, such as vote tallying, which is inherently dependent on the data values submitted by clients cannot be implemented using cobufs. But, in some cases, it may be possible to create new cobuf-like objects that perform the requisite computation while bounding the amount of information that may leak in the worst case. The design of such extensions to the cobuf interface is beyond the scope of this paper.

A modification of the Python loader analyzes code during loading and ensures that Fauxbook code cannot use Python reflection mechanisms for peeking at object fields. Every cobuf is tagged with an owner identifier that is assigned on a session basis following a successful user authentication, and cobuf contents may only be collated if the recipient cobuf's owner speaks for the owner of the cobuf from which the data is copied.

Cobufs are used to protect the integrity of the underlying social network graph. Because the owner identifier is attached in the web server layer, Fauxbook application code cannot forge cobufs on behalf of a user. This prohibits the application from adding impostors that leak sensitive data. A legitimate, user-initiated friend addition into the Fauxbook social network invokes a method in the user authentication library that generates the requisite link in the social graph corresponding to that `speaksfor` relationship.

Fauxbook stores user data in the Nexus filesystem. Goal formulas associated with each file constrain user access in accordance with the social graph. Moreover, Fauxbook files reside in a directory that can be accessed only by a process with the expected web framework process and Python code hashes. Additionally, each operation on each file in this directory has a policy: private, public, or friends. Private data of user a Alice is only accessible if an authority embedded in the web server attests to label *name.webserver* `says` *user=alice*. Alice can only read the files of her friend Bob if an embedded authority attests to the label *name.python* `says` *alice in bob.friends*. To verify whether this holds, the authority introspects on the contents of a publically readable friend file. This operation is trustworthy, because only the web framework can update the value of the current user and only the web framework acting on behalf of Bob can modify his friend file.

Taken together, the statements embedded in the labels described above attest to an environment in which the Fauxbook code cannot directly inspect personal information provided by its users. And even though the Nexus has no built-in notion of social networks or web users, the flexibility of the logical attestation framework was able to provide such privacy guarantees, demonstrating the generality and power of the logical attestation framework.

4.2 Other Applications

We have built other applications based on logical attestation. We outline their operation to provide a broader view on how labels can be used to prove desired characteristics to remote parties.

Movie Player. Platform lock-down is a long-standing and widely reviled problem with binary hash-based attestation. Yet, without any form of attestation, content owners are justifiably wary of distributing high-value digital content, such as movies, that can easily be copied and redistributed widely. With conventional approaches to attestation, a content owner wanting to ensure that her movies are not illegally copied would create a whitelist of known-to-be-trustworthy media players, demand a binary hash attestation, and stream content only if the player is on a list of trusted players. As a result, the user either needs to use a precertified player and operating system or forgo watching the movie.

Logical attestation makes it possible to offer much greater choice to users yet still satisfy the content owner's needs. Specifically, the user, instead of furnishing a binary attestation certificate, exports a label that says an IPC channel-connectivity analyzer has determined that user's program (whose hash need not be divulged in the certificate or elsewhere) lacks the ability to write to the disk or the network. We have built and implemented such a general-purpose IPC connectivity analyzer, though one could also imagine the content owner furnishing that labeling function to the user. In either case, the label provides a basis for a content provider to decide whether the user can be trusted with the data, but the user now has the flexibility to pick any player that can be analyzed and shown to satisfy the desired security policy.

Java Object Store. Unlike filesystem access control lists, logical attestation makes it possible to restrict access to a class of programs that possess an intrinsically-described capability. This can have far-reaching impact, including for performance optimization. For instance, Java virtual machines have long suffered object deserialization overhead. Because the Java runtime is typesafe and because data coming from the external world cannot be trusted, a number of type invariants needs to be checked dynamically during deserialization.

Logical attestation can be used to obviate such checking in cases where the object to be deserialized was generated by another typesafe Java virtual machine. Take, for instance, a Java object store, where the objects are stored on disk and later downloaded onto a separate computer. If the downloader can be assured that the entity producing that database was another Java virtual machine satisfying the same typesafety invariants, then the slow parts of sanity checking every byte of data can be skipped when reinstating an object. This is an instance of a class of applications based on transitive integrity verification [49].

Not-A-Bot. A recent proposal aims to combat email spam with certificates attached to each message that indicate whether that message originated from a human or from an automated script [17]. Using logical attestation, we have prototyped this approach. An email program was modified to obtain a certificate from the keyboard driver, where that certificate attests to the number of keypresses it received. Such a TPM-backed certificate then serves as an input to a SPAM classification algorithm.

TruDocs and CertiPics. Recent public scandals, such as those involving published pictures that were altered in significant ways and intelligence reports quoted in a manner that distorted their meaning, demonstrate the need to ensure important data is handled in accordance with appropriate guidelines. And there are many settings where such guidelines not only exist but, also, have been specified in a manner amenable to mechanistic enforcement [45]. To demonstrate that the logical attestation machinery is sufficient to implement such applications, we built TruDocs and CertiPics, two document handling systems that ensure that modifications comply with desired policies. There are many different ways in which the different bases for trust can be used to implement these applications. We describe some implementation choices, which cover the spectrum.

CertiPics is an image editing suite whose goal is to ensure that images to be used in publications conform to standards. CertiPics consists of a user interface that executes without privilege and a set of image processing elements, such as crop, color transform, resize, clone, and other manipulations supported by the portable bitmap suite of tools, that execute on the Nexus. In addition to generating a desired image from a given source, CertiPics concurrently generates a certified, unforgeable log of the transformations performed. This log, coupled with the source and final images, make it possible for analyzers to determine if a disallowed modification, such as cloning, was applied to the image.

TruDocs is a document display system whose goal is to ensure that a given excerpt conveys the beliefs intended in the original document. Implemented as a set of extensions to the OpenOffice suite, this application issues a certificate attesting that an excerpt speaks for the original document if the excerpt satisfies a given *use policy*. Supported use policies can admit changing typecase, replacing certain text fragments with ellipses, and inserting editorial comments in square brackets, while limiting the length and total number of excerpts.

Protocol Verifiers. The BGP protocol is widely deployed, critical to Internet routing, and suffers from vulnerability to misbehaving participants. A naive approach to ensuring that a given BGP speaker is trustworthy would involve equipping all BGP speakers with TPMs and attesting to their launch-time hashes; an instance of axiomatic trust. Such an effort would not only incur tremendous hardware costs to replace legacy hardware but would also entail a tremendous software certification effort to determine which, if any, of the many different versions of BGP software is correct.

Applying the logical attestation approach to this problem, using synthetic trust, yields a far simpler approach. We have designed a BGP protocol verifier by coupling a BGP parser with a set of minimal BGP safety rules that identify route fabrication and false origination attacks [39]. The verifier straddles a legacy BGP speaker whose inputs and outputs it monitors by acting as a proxy between the legacy speaker and other speakers. It ensures that every outgoing BGP advertisement or withdrawal conforms to the BGP safety rules; for instance, by ensuring that a host cannot advertise an n hop route to a destination for which the shortest advertisement that it itself received is m, for $n < m$. While the details of this application, such as how such verifiers can be connected into an overlay, how they react to non-conforming BGP messages, and how they admit local policy preferences, is beyond the scope of this paper, this application is an exemplar for the use of synthetic trust in a network setting.

5. Evaluation

This section reports measurements of costs for logical attestation. We first quantify costs imposed by the base Nexus microkernel, by logical attestation operations, and by system services. Next, we measure the cumulative cost this architecture imposes on our Fauxbook application.

Benchmarks presented in this section represent the median of at least 100 measurements when we observed a maximal deviation of upper and lower quartiles below 2.5% each, unless otherwise stated. Application benchmarks have the same bound, showing the median of at least 10 runs, each of at least 1000 requests. All performance data was obtained on a Dell Optiplex 745 containing a 2.13 GHz Intel E6400 CPU with 2MB of L2 cache and 2 GB of

	Nexus Bare	Nexus	Linux
null	352	808	*n/a*
null (block)	*n/a*	624	*n/a*
getppid	360	824	688
gettimeofday	640	1112	978
yield	736	1128	1328
open		8752	3240
close		4672	1816
read		3600	1808
write		11792	3900

Table 1: System call overhead, in cycles, comparing Nexus with Linux. System calls are cheaper in Nexus when interpositioning is disabled; when enabled, low-level operations have comparable performance, while high-level filesystem operations have higher overheads because their implementation employs multiple user-level servers.

component	lines	component	lines
kernel core	9904	headers	5020
IPC	1217	label mgmt	621
interpositioning	67	introspection	981
VDIR/VKEY	1165	networking	1357
generic guard†	4157	malloc	158 / 3322
filesystem†	1810	debug†	356
Xen†	9678	kernel drivers†	27238
posix library†	3953	user drivers†	24830
TCB	20490		

Table 2: Lines of Code. Items marked † are optional. Nexus has a small TCB, enabled by factoring device drivers out of the kernel.

main memory running in 32bit uniprocessor mode. This system has an Intel 82540EM network adapter and Atmel v1.2 compatible TPM. Linux results were obtained with a Ubuntu 10.10 installation with the default 2.6.35-23 kernel.

5.1 Microbenchmarks

The Nexus system architecture implements services in user-space whenever possible, in order to reduce the size of the trusted computing base (TCB). Such a microkernel design inevitably adds overhead due to longer communication paths. System call interpositioning imposes further overhead for parameter marshaling at every kernel-mode switch.

To establish the cost of these design decisions, we compare invocation cost of common operations both with and without interpositioning directly to Linux. To establish size of the TCB, we calculate the number of lines of code contributing to each essential Nexus component.

Kernel Operations. Table 1 shows system call costs for a modified Nexus without interpositioning, standard Nexus, and Linux. Invocation of an empty `null` call gives an upper bound of interpositioning overhead, at $808 - 352 = 456$ extra cycles. An interposed call that is blocked returns earlier than a completed call, after only 624 cycles in total. Nexus executes the Posix calls `getppid`, `gettimeofday` and `yield` faster than Linux (1.5-2x) when interpositioning is disabled. When enabled, performance is comparable (0.8-1.2x). We conclude that parameter marshaling required for interpositioning does not significantly impact basic call overhead. File operations are between 2 and 3x more expensive, on the other hand, at least in part due to the communication imposed by the client-server microkernel architecture.

TCB Size. Table 2 reports the size of the TCB, using David Wheeler's `sloc` counter.

At less than 25 thousand lines, the kernel is larger than some microkernels but smaller than many device drivers. Device driver reference monitors (DDRMs) [56] enable the Nexus device drivers, comprising keyboard, mouse, pci, network (e1000, pcnet32, and tg3), sound (i810 and es1370), storage (ide) and tpm (atmel 1.1, nsc 1.1, and emulator 1.2) drivers, to execute in user space. Shown in the table as `user drivers`, they operate without privilege and are constrained throughout execution by a device driver safety policy designed to protect the isolation guarantees provided by Nexus IPDs, even in the presence of misbehaving and malicious drivers. We maintain implementations of a VESA video driver, a TG3 network driver, and a PCI driver in the kernel (`kernel drivers`) for debugging and for performance comparison purposes.

The filesystem functionality in Nexus is spread over three components. Basic namespace services are provided by the kernel core, while a RAM-based store (`filesystem`) provides transient data storage. In addition, the networking module provides TFTP and NFS-based file and directory access. Even though these networking protocols do not provide security guarantees and the data resides on physically remote disks, the SSR implementation ensures that application data as well as the Nexus kernel's private VDIRs and VKEYs contents are tamper- and replay-proof.

The generic guard used by default for kernel resources is less than 5000 lines of code. For comparison, the Broadcom network driver alone measures 16920 lines (version tg3-3.110g). Note that the generic guard implementation is optional; applications can use alternative guards for their resources.

We have biult a few additional components to support legacy applications. A Posix library provides a familiar API for Nexus applications operating directly on the Nexus. Similar to past work on VM-based secure systems [9, 14], Xen support enables the Nexus to execute monolithic legacy systems in isolated virtual machines. A compact malloc library provides in-kernel memory management. Auxiliary libraries that are used by Nexus applications but are not specific to the Nexus kernel, such as μClibc and OpenSSL, are not shown.

5.2 Logical Attestation

Logical attestation can be prohibitively expensive without caching and system-backed credentials. We quantify the run-time cost of guard invocation and the effects of caching, measure scalability with proof complexity, and compare control operations overhead of system-backed and cryptographically implemented labels.

Invocation. Authorization cost depends on how many checks have to be performed. Figure 4 compares these costs for a bare system call invocation. To establish a baseline, we give the cost of evaluating a trivial proof consisting of a single assumption. From left to right, the figure presents runtime for the case where (a) authorization is disabled (`system call`), (b) a default ALLOW goal is set (`no goal`), (c) a real goal is set, but no proof was supplied by the subject (`no proof`), (d) the supplied proof is not sound (`not sound`), (e) the proof is sound and all premises are supported (`pass`), (f) the proof lacks a credential (`no cred`), (g) the proof depends on an embedded authority (`embed auth`), and (h) the proof depends on an external authority (`auth`). Solid bars depict execution time with the kernel decision cache enabled; dashed bars with it disabled. The upcall into the guard increases cost of authorizing

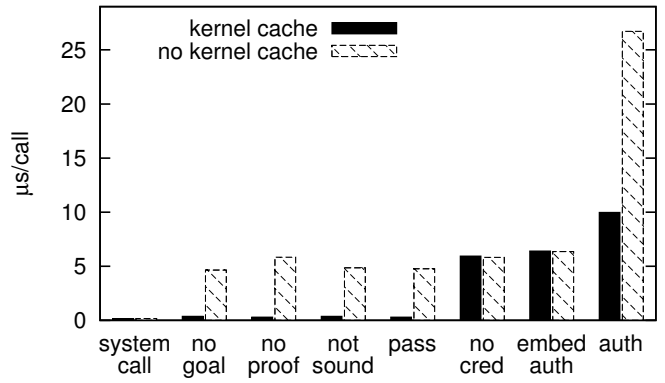

Figure 4: Authorization Cost. Cached decisions add around 456 cycles, keeping total runtime well below one μsecond. Upcalls into the guard are 16-20x as expensive.

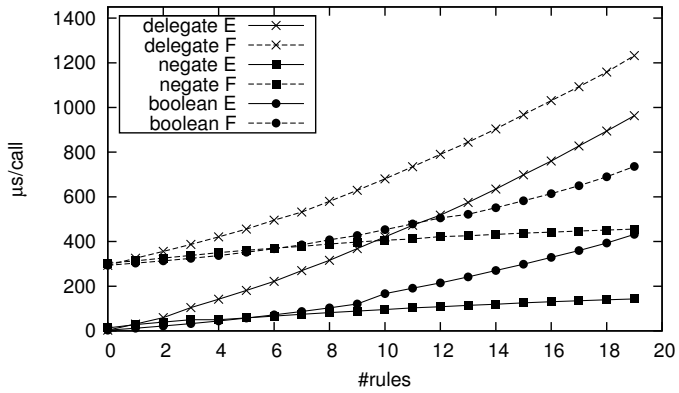

Figure 5: Proof Evaluation Cost. With caching, access control cost is reduced to tens of cycles.

an operation from 624 to 12424 cycles. In general, guard operations are between 16 and 20x as expensive as kernel decisions. Cases (a)–(e) indicate the effectiveness of decision caching. The jump between (e) and (f) clearly delineates the set of cacheable proofs. Operations that cannot be cached are credential matching (20% overhead over `pass`) and invocation of an embedded (31%) or external authority (106%). The last case is exceptionally expensive with caching disabled, due to the cumulative effect of interposing on all system calls made by the authority process.

Proof Evaluation. When decisions cannot be cached, time spent in proof evaluation depends on proof size. Figure 5 shows proof checking time for proofs of increasing length, measured as the number of inference rules applied. It presents execution time for application of the simplest NAL rule, double negation introduction, together with two common NAL deduction rules: `speaksfor` delegation and disjunction elimination. The solid lines show the isolated cost of proof checking, while the dashed lines show total execution time, which also incorporates the time to check labels and look up authorities. Both curves show the same trend, with a different constant cost that reflects the overhead of IPCs and scheduling the processes that implement the label store and authority. Overall, the proof checker executes all proofs shorter than 15 steps in less than 1ms. All practical proofs that we have written in our applications involve less than 15 steps.

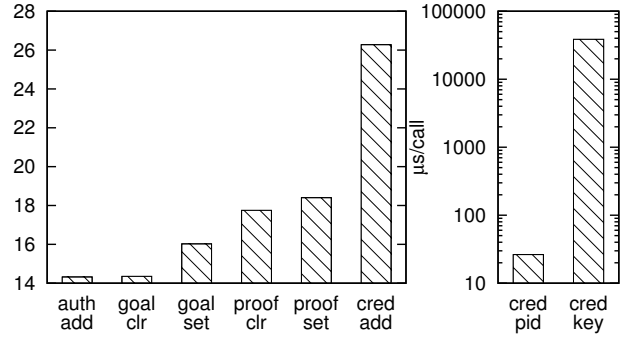

Figure 6: Authorization Control Overhead. Avoidance of cryptography reduces cost by 3 orders of magnitude.

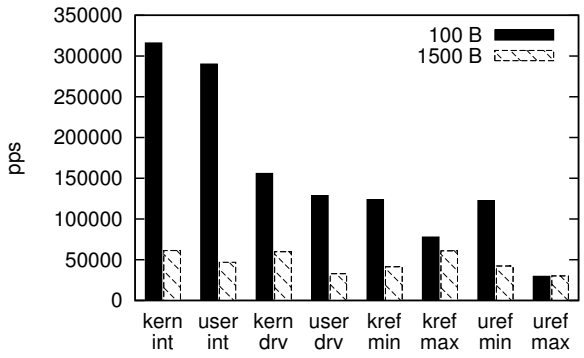

Figure 7: Overhead of interpositioning. Caching decisions decrease packet processing rate by less than 6%.

Control Operations. Logical attestation requires management of goals, proofs, and labels. Nexus is optimized for efficient invocation, even if it meant increased cost for these less-frequently executed operations. Figure 6 summarizes control-operation execution times. System-backed and cryptographic operations are displayed at difference scales, because of the huge gap in execution times. The left-hand figure shows cost of operations without cryptography at linear scale: authority registration (`auth add`), goal deletion and insertion (`goal clr/set`), proof deletion and insertion (`proof clr/set`), and credential insertion (`cred add`). The credential insertion operation is twice as expensive as the next slowest, because each label has to be parsed to verify that the caller is allowed to make the statement. The right-hand figure compares this same system-backed credential insertion call (`cred pid`) to insertion of a cryptographically signed label, using a logarithmic scale. Verification of the signed label is three orders of magnitude more expensive than the same operation for its system-backed equivalent, substantiating our view that cryptographic operations should be avoided if possible.

5.3 Operating System Services

Introspection and interpositioning are essential to using our logical attestation framework. Since introspection, whose performance is comparable to the file I/O numbers shown in Table 1, typically does not lie on the critical path, its performance overhead is unlikely to affect benchmarks. Interpositioning, on the other hand, introduces dynamic checks that could prove prohibitive, especially

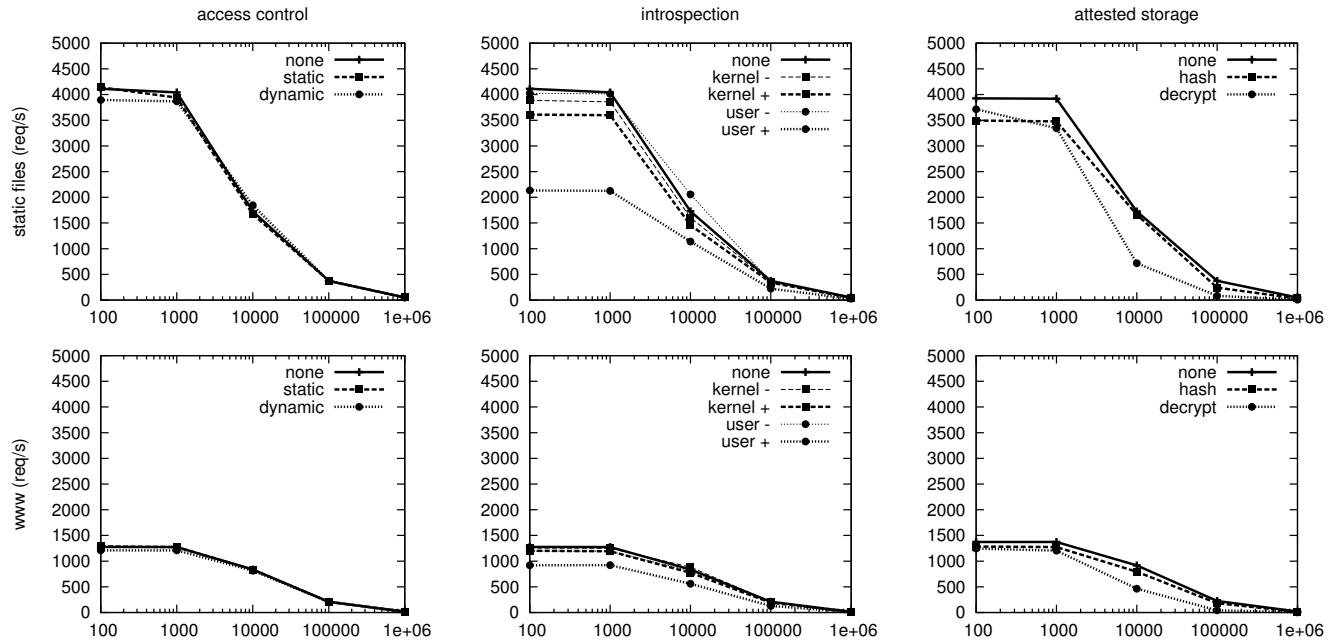

Figure 8: Application evaluation: impact of access control (col. 1), reference monitors (col. 2) and attested storage (col. 3) on a webserver serving static files (row 1) and dynamic Python content (row 2). Filesize varies from 100B to 1MB, the x-axis is plotted in logarithmic scale.

on highly active communication paths.

Interpositioning. To calculate an upper bound on overhead from dynamic checks, we install progressively more demanding reference monitors and measure maximally obtained throughput for a trivial application: a UDP echo server written in 27 lines of C. Figure 7 plots throughput in packets per second for increasing levels of protection, for both small and large size packets. For this test, quartiles lie within 5% of the median, except for the first measurement, which lies within 18%.

Figure 7 illustrates the causes of interpositioning overhead by progressively enabling more of the interpositioning machinery. The first two cases (`kern-int` and `user-int`) show throughput when the driver directly responds to requests from within the interrupt handler, bypassing all interpositioning. While a kernel driver achieves 17% higher rate than its user-level equivalent, executing untrusted application code within the device driver interrupt handler is not practical without substantial additional measures. Cases `kern-drv` and `user-drv` show the more realistic scenario of an independent server application that communicates with the driver through kernel IPC. The 2x drop in throughput is due to IPC, scheduling, routing, and the user-level TCP/IP stack. In cases `kref` and `uref`, reference monitors, located in the kernel and at user-level, respectively, are installed on the userspace driver to ensure compliance with a device driver safety policy. The measurements for `min` and `max` show the throughput with and without caching, respectively. While cache misses during reference monitoring can reduce throughput by 50%, caching can reduce monitoring overhead to as little as 4% for small packets. When user-space reference monitors are employed, the impact on throughput is as high as 77%, but the decision cache can reduce the overhead to less than 5%.

5.4 Application

In the worst case, overhead of logical attestation can be three orders of magnitude over functional cost if credentials are implemented by using cryptographic labels. The Nexus architecture is based on the assumption that caching can reduce such overhead to negligible levels. To test this hypothesis, we measure the effects of the Nexus mechanisms on a demanding workload: throughput of the Fauxbook web application. We measure the impact of access control, interpositioning, and attested storage on throughput—both in the case of static file serving and dynamic Python execution. Figure 8 shows these three sources of cost, from left to right. Each figure plots HTTP requests per second versus filesize at logarithmic scale. The top row of graphs presents throughput for a static fileserver, the bottom row displays the same numbers for a dynamic server running Python.

The curves in graphs compare throughput under three types of access control: `none` performs no authorization checks, `static` evaluates a cacheable proof, and `dynamic` invokes an external authority. Figure 4 showed considerable cost at the micro level. At the application level, worst-case overhead is 6% for a minimal page with guard invocation. Comparing the two rows shows that overhead is consistently less pronounced for the multi-tier server with Python than for the simpler fileserver. Interpositioning cost, displayed in the middle column, significantly decreases throughput without caching. Worst case throughput is roughly 50% with the user-space reference monitor. With caching, this overhead is only 6%.

Hashing and encryption are expensive. Encryption decreases throughput by up to 85%; hashing up to 38%. Small file hashing suffers from a large 1 kB blocksize, requiring padding for the smallest files. Eventually, efficiency consistently decreases with larger file sizes as the per byte cost of hashing increases, while per request costs stay constant: the worst case occurs at the largest size. When files are accessed through Python, the overheads remain similar, with a worst case of 85% at the largest filesize.

6. Related Work

Hash-based attestation was initially proposed in security architectures for secure bootstrapping [15, 22, 3]. Logical attestation differs from this work in that credentials can capture properties that are not derived from the launch-time hash of a program. Like logical attestation, semantic remote attestation [18] and property-based attestation [40] suggest encoding meaningful application-level properties. Unlike logical attestation, their protocol formats are application-specific or unspecified, and they do not provide a corresponding implementation in an operating system. Nexus is the first OS to offer semantically rich attestation as a system service and demonstrate an efficient implementation.

TCGLinux [42] proposes to add attestation capabilities to a conventional Linux system. This approach is problematic due to Linux's large TCB and lack of strong isolation. Software-based attestation [47, 46] derives its correctness from assumptions about execution time, requiring intricate knowledge of target hardware. Hardware-based approaches do not have this constraint. BIND [49] offers fine-grained attestation on an untrusted OS through monitoring and sandboxing by a trusted software module. Flicker [32] extends the approach to legacy systems. Logical attestation is not inconsistent with such isolation, independent of the OS. The system-level implementation in Nexus avoids costly repeated state serialization, however, rendering per-call interpositioning feasible and offering assurance to all system objects.

To secure legacy systems, trusted hardware has been combined with virtualization, in the form of trusted hypervisors [41, 14], mediated TPM access [10], and full TPM virtualization [4]. Multiplexing the TPM hardware interface in software does not solve the problems stemming from limitations of the binary hash-based attestation interface provided by the hardware. Some approaches refine or replace the hardware interface: property-based virtualization [40] supports variations on the TPM Quote function, Not-a-Bot [17] presents human presence attestations, and TrustVisor exposes the isolated process interface of Flicker [31]. Even higher-level VMMs protection is inherently limited by the system interface, whereas operating system protection extends to all primitives, including files and users.

Microkernels offer another path to TCB minimization. seL4 demonstrated formal verification of correctness of microkernels [24]. L4 [21] has been proposed as a a small TCB [37, 20, 19]. Nizza [20] extends it with trusted wrappers, similar to how TrustVisor adapts VMMs. Nexus is a similarly small OS, but Nexus integrates a comprehensive authorization architecture. EROS [48] is another capability-based microkernel OS for mutually distrusting users. Like Nexus, it caches state in the kernel. EROS uses kernel-protected numerical capabilities that are not suitable for remote attestation and policies are not as flexible as arbitrary proofs. Flask [51] introduced an authorization architecture that supports these features and caches security decisions in the kernel. In this model, Nexus implements and evaluates a complete, expressive, policy mechanism.

The NGSCB [9] splits the system software into a large untrusted and smaller trusted compartment. XOMOS [29] treats the OS as an untrusted compartment from which trusted processes are shielded using hypothesized CPU extensions. Logical attestation replaces this duality with a delegated trust model that extends to multiparty environments. Wedge [5] splits applications into least-privilege compartments to isolate internal secrets, similar to the authenticated interpreter in the Nexus cloud stack, though this does not extend to multiple processes and system objects, such as files. HiStar [58] assures systemwide data confidentiality and integrity by enforcing information flow constraints on Asbestos [8] labels.

TPMs also support implementation of trustworthy abstractions that are independent of the operating system. Monotonic counters [43, 27] and append-only logs [6] are examples of trustworthy computing abstractions rooted in specialized hardware. Storage services on untrusted servers use TPMs [28, 55], possibly through virtual counters [53] and other cryptographic operations [30, 13]. An operating system such as the Nexus can offer the collective primitives and assurances in a single system, but requires more pervasive software changes than any individual system.

A recent survey reviews the state of the art in trusted computing in more depth than we can here [36]. Proof based authorization is not limited to trusted computing. Lampson argued for combining logics, reference monitors, and chains of trust to construct secure systems [25]. NAL borrows ideas and notation from Taos [57] and CDD [1] (among others [45]). Proof carrying authorization [2] introduces distributed decision based on logic proof evaluation. Alpaca [26] generalizes the approach to include common credential formats and cryptographic primitives.

For any credential issued by an operating system to be trustworthy, the system implementation must be free of bugs. The construction of a verifiable OS kernel is an area of active research [24], while efforts to reduce the kernel footprint [56] are synergistic.

7. Conclusions

Logical attestation offers strong operating system authorization by deriving trustworthiness from unforgeable system properties. It incorporates software analysis and containment as sources for trust decisions and captures the results in what we call labels: meaningful, unforgeable, and attributable statements written in a high-level logic. Labels serve as facts in proof-based authorization to offer rational decision making and incontestable audit trails. A comprehensive operating system architecture combines efficient and secure means for label generation, communication, and consumption with system services for introspection, interpositioning, and secure persistent storage.

We have implemented logical attestation in Nexus, a trustworthy operating system that combines a small TCB with a strong root of trust in the form of a TPM secure coprocessor. Evaluation shows that strong isolation and trust are compatible with high performance: multilevel caching amortizes authorization cost to offer system operations on level footing with Linux; cryptography avoidance reduces distributed decision making cost by three orders of magnitude. Even with full isolation, Nexus mechanisms impose modest overhead on policy-rich cloud computing applications.

Acknowledgments

The authors would like to thank the anonymous reviewers and our shepherd, Adrian Perrig, for their insightful feedback. This work was supported in part by ONR grant N00014-09-1-0652, AFOSR grant F9550-06-0019, NSF grants 0430161, 0964409, CNS-1111698 and CCF-0424422 (TRUST), and a gift from Microsoft Corporation.

8. References

[1] Martín Abadi. Variations in Access Control Logic. In *Proc. of the Conference on Deontic Logic in Computer Science*, Luxembourg, July 2008.

[2] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Proc. of the Conference on Computer and Communications Security*, Singapore, November 1999.

[3] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A Secure and Reliable Bootstrap Architecture. In

Proc. of the IEEE Symposium on Security and Privacy, Oakland, CA, May 1997.

[4] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *Proc. of the USENIX Security Symposium*, Vancouver, Canada, August 2006.

[5] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proc. of the Symposium on Networked System Design and Implementation*, San Francisco, CA, April 2008.

[6] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. of the Symposium on Operating System Principles*, Stevenson, WA, October 2007.

[7] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Comm. of the ACM*, 9:143–155, March 1966.

[8] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, M. Frans Kaashoek, and Robert T. Morris. Labels and Event Processes in the Asbestos Operating System. In *Proc. of the Symposium on Operating Systems Principles*, Brighton, UK, October 2005.

[9] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A Trusted Open Platform. *Computer*, 36(7):55–62, July 2003.

[10] Paul England and Jork Loeser. Para-Virtualized TPM Sharing. In *Proc. of the International Conference on Trusted Computing and Trust in Information Technologies*, Villach, Austria, 2008.

[11] Ùlfar Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[12] Electronic Frontier Foundation. Facebook's New Privacy Changes: The Good, The Bad, and The Ugly. http://www.eff.org/deeplinks/2009/12/facebooks-new-privacy-changes-good-bad-and-ugly.

[13] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and Secure Distributed Read-Only File System. In *Proc. of the Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[14] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proc. of the Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[15] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The Digital Distributed System Security Architecture. In *Proc. of the National Computer Security Conference*, Baltimore, MD, October 1989.

[16] Li Gong. Java Security: Present and Near Future. *IEEE Micro*, 17(3):14–19, May/June 1997.

[17] Ramakrishna Gummadi, Hari Balakrishnan, Petros Maniatis, and Sylvia Ratnasamy. Not-a-Bot: Improving Service Availability in the Face of Botnet Attacks. In *Proc. of the Symposium on Networked System Design and Implementation*, Boston, MA, April 2009.

[18] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic Remote attestation: A Virtual Machine Directed Approach to Trusted Computing. In *Proc. of the USENIX Virtual Machine Research and Technology Symposium*, San Jose, CA, May 2004.

[19] Hermann Härtig. Security Architectures Revisited. In *Proc. of the SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[20] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza Secure-System Architecture. In *Proc. of the International Conference on Collaborative Computing*, San Jose, CA, December 2005.

[21] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The Performance of μ-Kernel-Based Systems. In *Proc. of the Symposium on Operating Systems Principles*, Saint Malo, France, October 1997.

[22] Hermann Härtig and Oliver Kowalski and Winfried Kühnhauser. The BirliX Security Architecture. *Journal of Computer Security*, 2(1):5–21, 1993.

[23] William K. Josephson, Emin Gün Sirer, and Fred B. Schneider. Peer-to-Peer Authentication With a Distributed Single Sign-On Service. In *Proc. of the Workshop on Peer-to-Peer Systems*, San Diego, CA, February 2004.

[24] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. of the Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[25] Butler Lampson. Computer Security in the Real World. *IEEE Computer*, 37(6), 2004.

[26] Chris Lesniewski-Laas, Bryan Ford, Jacob Strauss, Robert Morris, and Frans M. Kaashoek. Alpaca: Extensible Authorization for Distributed Services. In *Proc. of the Conference on Computer and Communications Security*, Alexandria, VA, October 2007.

[27] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proc. of the Symposium on Networked System Design and Implementation*, Boston, MA, April 2009.

[28] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure Untrusted Data Repository. In *Proc. of the Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[29] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proc. of the Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[30] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proc. of the Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[31] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010.

[32] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. of the European Conference on Computer Systems*, Glasgow, Scotland, April 2008.

[33] Ralph C. Merkle. Protocols for Public Key Cryptosystems.

In *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1980.

[34] Ralph C. Merkle. A Certified Digital Signature. In *Proc. of the International Cryptology Conference*, Santa Barbara, CA, August 1989.

[35] George C. Necula. Proof-Carrying Code. In *Proc. of the Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 1997.

[36] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping Trust in Commodity Computers. In *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010.

[37] Birgit Pfitzmann, James Riordan, Christian Stüble, Michael Waidner, and Arnd Weber. The PERSEUS System Architecture. Technical Report RZ3335 (93381), IBM Research Division, Zurich, April 2001.

[38] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[39] Patrick Reynolds, Oliver Kennedy, Emin Gün Sirer, and Fred B. Schneider. Securing Bgp Using External Security Monitors. Technical Report TR2006-2065, Cornell University, Computing and Information Science, Ithaca, New York, December 2006.

[40] Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. Property-Based TPM Virtualization. In *Proc. of the International Conference on Information Security*, Hyderabad, India, December 2008.

[41] Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert van Doorn, John Linwood Griffin, and Stefan Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. Technical Report RC23511 (W0502-006), IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, February 2005.

[42] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proc. of the USENIX Security Symposium*, San Diego, CA, August 2004.

[43] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual Monotonic Counters and Count-limited Objects Using a TPM without a Trusted OS. In *Proc. of the Workshop on Scalable Trusted Computing*, Fairfax, VA, November 2006.

[44] Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 1(3), February 2000.

[45] Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus Authorization Logic: Design Rationale and Applications. *ACM Transactions on Information and System Security*, 14(1), May 2011.

[46] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying Integrity and Guaranteeing Execution of Code on Legacy Platforms. In *Proc. of the Symposium on Operating Systems Principles*, Brighton, UK, October 2005.

[47] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SWATT: Software-based Attestation for Embedded Devices. In *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[48] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. In *Proc. of the Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[49] Elaine Shi, Adrian Perrig, and Leendert van Doorn. BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In *Proc. of the Symposium on Security and Privacy*, 2005.

[50] Emin Gün Sirer, Robert Grimm, Arthur J. Gregory, and Brian N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *Proc. of the Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[51] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proc. of the USENIX Security Symposium*, Washington, DC, August 1999.

[52] Richard Stallman. Can You Trust Your Computer? Available at `http://www.gnu.org/philosophy/can-you-trust.html`.

[53] Marten van Dijk, Jonathan Rhodes, Luis F. G. Sarmenta, and Srinivas Devadas. Offline Untrusted Storage with Immediate Detection of Forking and Replay Attacks. In *Proc. of the Workshop on Scalable Trusted Computing*, Alexandria, VA, November 2007.

[54] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[55] Carsten Weinhold and Hermann Härtig. VPFS: Building a Virtual Private File System with a Small Trusted Computing Base. In *Proc. of the European Conference on Computer Systems*, Glasgow, Scotland, April 2008.

[56] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *Proc. of the Symposium on Operating System Design and Implementation*, San Diego, CA, December 2008.

[57] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *Transactions on Computer Systems*, 12(1), 1994.

[58] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *Proc. of the Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.