Hardware Enforcement of Application Security Policies Using Tagged Memory

Nickolai Zeldovich*, Hari Kannan†, Michael Dalton†, and Christos Kozyrakis†
*MIT †Stanford University

ABSTRACT

Computers are notoriously insecure, in part because application security policies do not map well onto traditional protection mechanisms such as Unix user accounts or hardware page tables. Recent work has shown that application policies can be expressed in terms of information flow restrictions and enforced in an OS kernel, providing a strong assurance of security. This paper shows that enforcement of these policies can be pushed largely into the processor itself, by using tagged memory support, which can provide stronger security guarantees by enforcing application security even if the OS kernel is compromised.

We present the *Loki* tagged memory architecture, along with a novel operating system structure that takes advantage of tagged memory to enforce application security policies in hardware. We built a full-system prototype of Loki by modifying a synthesizable SPARC core, mapping it to an FPGA board, and porting HiStar, a Unix-like operating system, to run on it. One result is that Loki allows HiStar, an OS already designed to have a small trusted kernel, to further reduce the amount of trusted code by a factor of two, and to enforce security despite kernel compromises. Using various workloads, we also demonstrate that HiStar running on Loki incurs a low performance overhead.

1 INTRODUCTION

A significant part of the computer security problem stems from the fact that security of large-scale applications usually depends on millions of lines of code behaving correctly, rendering security guarantees all but impossible. One way to improve security is to separate the enforcement of security policies into a small, trusted component, typically called the trusted computing base [19], which can then ensure security even if the other components are compromised. This usually means enforcing security policies at a lower level in the system, such as in the operating system or in hardware. Unfortunately, enforcing application security policies at a lower level is made difficult by the *semantic gap* between different layers of abstraction in a system. Since the interface traditionally provided by the OS kernel or by hardware is not expressive enough to capture the high-level semantics of application security policies, applications resort to building their own ad-hoc security mechanisms. Such mechanisms are often poorly designed and implemented, leading to an endless stream of compromises [22].

As an example, consider a web application such as Facebook or MySpace, where the web server stores personal profile information for millions of users. The application's security policy requires that one user's profile can be sent only to web browsers belonging to the friends of that user. Traditional low-level protection mechanisms, such as Unix's user accounts or hardware's page tables, are of little help in enforcing this policy, since they were designed with other policies in mind. In particular, Unix accounts can be used by a system administrator to manage different users on a single machine; Unix processes can be used to provide isolation; and page tables can help in protecting the kernel from application code. However, enforcing or even expressing our example website's high-level application security policy using these mechanisms is at best difficult and error-prone [17]. Instead, such policies are usually enforced throughout the application code, effectively making the entire application part of the trusted computing base.

A promising technique for bridging this semantic gap between security mechanisms at different abstraction layers is to think of security in terms of what can happen to data, instead of specifying the individual operations that can be invoked at any particular layer (such as system calls). For instance, recent work on operating systems [10, 18, 35, 36] has shown that many application security policies can be expressed as restrictions on the movement of data in a system, and that these security policies can then be enforced using an information flow control mechanism in the OS kernel.

This paper shows that hardware support for tagged memory allows enforcing data security policies at an even lower level—directly in the processor—thereby providing application security guarantees even if the kernel is compromised. To support this claim, we designed Loki, a hardware architecture that provides a word-level memory tagging mechanism, and ported the HiStar operating system [35] (which was designed to enforce application security policies in a small trusted kernel) to run on Loki. Loki's tagged memory simplifies security enforcement by associating security policies with data at the lowest level in the system—in physical memory. The

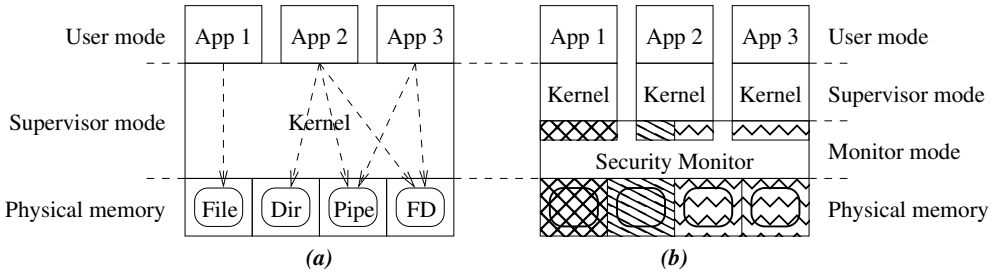

Figure 1: A comparison between (a) traditional operating system structure, and (b) this paper's proposed structure using a security monitor. Horizontal separation between application boxes in (a), and between stacks of applications and kernels in (b), indicates different protection domains. Dashed arrows in (a) indicate access rights of applications to pages of memory. Shading in (b) indicates tag values, with small shaded boxes underneath protection domains indicating the set of tags accessible to that protection domain.

resulting simplicity is evidenced by the fact that the port of HiStar to Loki has less than half the amount of trusted code than HiStar running on traditional CPUs. Finally, we show that tagged memory can achieve strong security guarantees at a minimal performance cost, by building and evaluating a full system prototype of Loki running HiStar.

While a tagged memory mechanism on its own can control read and write access to physical resources, it is not sufficient for enforcing strict information flow control. In particular, the lack of a name translation mechanism makes it difficult to avoid certain kinds of covert channels, as we will discuss later. To this end, this paper presents a novel OS structure that can enforce the same application security policy under two threat models. The first is a simpler *discretionary access control* model, which aims to control read and write access to memory, and is enforced largely in hardware. The second is a more complex *mandatory access control* model, which aims to control all ways in which data could be passed between processes, and is enforced in an OS kernel. The key difference between our OS structure and a traditional one is that the kernel is not trusted to enforce the discretionary parts of its mandatory access control model. Instead, it is the hardware's job to control read and write access to memory, and the kernel is only trusted to minimize covert channels.

The rest of the paper is structured as follows. The next section describes our overall system architecture and its security goals, as well as our experimental prototype. Section 3 describes the structure of our operating system in more detail, and Section 4 describes the tagged memory processor we developed as part of this work. Section 5 presents an evaluation of the security and performance of our prototype, Section 6 discusses related work, and Section 7 concludes.

2 System Architecture

This paper describes a combination of a new hardware architecture, called Loki, that enforces security policies in hardware by using tagged memory, together with a modified version of the HiStar operating system [35], called LoStar, that enforces discretionary access components of its information flow policies using Loki. The overall structure of this system is shown in Figure 1.

Traditional OS kernels, shown in Figure 1 (a), are tasked with both implementing abstractions seen by user-level code as well as controlling access to data stored in these abstractions. LoStar, shown in Figure 1 (b), separates these two functions by using hardware to control data access. In particular, the Loki hardware architecture associates *tags* with words of memory, and allows specifying protection domains in terms of the tags that can be accessed. LoStar manages these tags and protection domains from a small software component, called the *security monitor*, which runs underneath the kernel in a special processor privilege mode called *monitor mode*. The security monitor translates application security policies on data, specified in terms of *labels* on kernel objects in the HiStar operating system, into tags on the corresponding physical memory, which the hardware then enforces.

Most systems enforce security policies in hardware through a translation mechanism, such as paging or segmentation. However, enforcing security in a translation mechanism means that security policies are bound to virtual resources, and not to the actual physical memory storing the data being protected. As a result, the policy for a particular piece of data in memory is not well-defined in hardware, and instead depends on various invariants being implemented correctly in software, such as the absence of aliasing. Tagging physical memory helps bridge the semantic gap between the data and its security policy, and makes the security policy unambiguous even at a low level, while requiring a much smaller trusted code base.

As mentioned previously, tagged memory alone is not sufficient for enforcing strict information flow control, because dynamic allocation of resources with fixed names, such as physical memory, contains inherent covert channels. For example, a malicious process with access to a secret bit of data could signal that bit to a

colluding non-secret process on the same machine by allocating many physical memory pages and freeing only the odd- or even-numbered pages depending on the bit value. Operating systems like HiStar solve such problems by virtualizing resource names (e.g. using kernel object IDs) and making sure that these virtual names are never reused. However, the additional kernel complexity can lead to bugs far worse than the covert channels the added code was trying to fix. Moreover, implementing equivalent functionality in hardware would not be inherently any simpler than the OS kernel code it would be replacing, and would not necessarily improve security.

What hardware support for tagged memory can address, however, is the the tension between stronger security and increased complexity seen in an OS kernel. In particular, this paper introduces a new, intermediate level of security provided by hardware, which can enforce a subset of the kernel's security guarantees, as illustrated by our hybrid threat model in Figure 2. In the simplest case, we are concerned with two security levels, *high* and *low*, and the goal is ensuring that data from the high level cannot influence data in the low level. There are multiple interpretations of high and low. For instance, high might represent secret user data, in which case low would be world-readable, as in [2]. Alternatively, low could represent integrity-protected system configuration files, which should not be affected by high user inputs, as in [3].

The hybrid model provides different enforcement of our security goal under different assumptions. In particular, the weaker *discretionary access control* model, enforced by the tagging hardware and the security monitor, disallows both high processes from modifying low data and low processes from reading high data. However, if a malicious pair of high and low processes collude, they can exploit covert channels to subvert our security goal, as shown by the dashed arrow in Figure 2. The stronger *mandatory access control* model aims to prevent such covert communication, by providing a carefully designed kernel interface, like the one in HiStar, in a more complex OS kernel. The resulting hybrid model can enforce security largely in hardware in the case of only one malicious or compromised process, and relies on the more complex OS kernel when there are multiple malicious processes that are colluding.

The rest of this section will first describe LoStar from the point of view of different applications, illustrating the security guarantees provided by different parts of the operating system. We will then provide an overview of the Loki hardware architecture, and discuss how the LoStar operating system uses Loki's hardware mechanisms.

2.1 Application perspective

One example of an application in LoStar is the Unix environment itself. HiStar implements Unix in a user-space

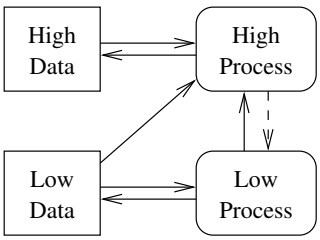

Figure 2: A comparison of the *discretionary access control* and *mandatory access control* threat models. Rectangles represent data, such as files, and rounded rectangles represent processes. Arrows indicate permitted information flow to or from a process. A dashed arrow indicates information flow permitted by the discretionary model but prohibited by the mandatory model.

library, which in turn uses HiStar's kernel labels to implement its protection, such as the isolation of a process's address space, file descriptor sharing, and file system access control. As a result, unmodified Unix applications running on LoStar do not need to explicitly specify labels for any of their objects. The Unix library automatically specifies labels that mimic the security policies an application would expect on a traditional Unix system. However, even the Unix library is not aware of the translation between labels and tags being done by the kernel and the security monitor. Instead, the kernel automatically passes the label for each kernel object to the underlying security monitor.

LoStar's security monitor, in turn, translates these labels into tags on the physical memory containing the respective data. As a result, Loki's tagged memory mechanism can directly enforce Unix's discretionary security policies without trusting the kernel. For example, a page of memory representing a file descriptor is tagged in a way that makes it accessible only to the processes that have been granted access to that file descriptor. Similarly, the private memory of a process's address space can be tagged to ensure that only threads within that particular process can access that memory. Finally, Unix user IDs are also mapped to labels, which are then translated into tags and enforced using the same hardware mechanism.

An example of an application that relies on both discretionary and mandatory access control is the HiStar web server [36]. Unlike other Unix applications, which rely on the Unix library to automatically specify all labels for them, the web server explicitly specifies a different label for each user's data, to ensure that user data remains private even when handled by malicious web applications. In this case, if an attacker cannot compromise the kernel, user data privacy is enforced even when users invoke malicious web applications on their data. On the other hand, if an attacker can compromise the kernel, malicious web applications can leak private data from one user to another, but only for users that invoke the malicious code. Users that don't invoke the malicious code

will still be secure, as the security monitor will not allow malicious kernel code to access arbitrary user data.

2.2 Hardware overview

The design of the Loki hardware architecture was driven by three main requirements. First, hardware should provide a large number of non-hierarchical protection domains, to be able to express application security policies that involve a large number of disjoint principals. Second, the hardware protection mechanism should protect low-level physical resources, such as physical memory or peripheral devices, in order to push enforcement of security policies to the lowest possible level. Finally, practical considerations require a fine-grained protection mechanism that can specify different permissions for different words of memory, in order to accommodate programming techniques like the use of contiguous data structures in C where different data structure members could have different security properties.

To address these requirements, Loki logically associates an opaque 32-bit *tag* with every 32-bit word of physical memory. Tag values correspond to a security policy on the data stored in locations with that particular tag. Protection domains in Loki are specified in terms of tags, and can be thought of as a mapping between tags and permission bits (read, write, and execute). Loki provides a software-filled *permissions cache* in the processor, holding permission bits for some set of tags accessed by the current protection domain, which is checked by the processor on every instruction fetch, load, and store.

A naive implementation of word-level tags could result in a 100% memory overhead for tag storage. To avoid this problem, Loki implements a multi-granular tagging scheme, which allows tagging an entire page of memory with a single 32-bit tag value. This optimization turns out to be quite effective, and will be described in more detail later in the paper.

Tag values and permission cache entries can only be updated in Loki while in a special processor privilege mode called *monitor mode*, which can be logically thought of as more privileged than the traditional supervisor processor mode. Hardware invokes tag handling code running in monitor mode on any tag permission check failure or permission cache miss by raising a *tag exception*. To avoid including page table handling code in the trusted computing base, the processor's MMU is disabled while executing in monitor mode.

2.3 OS overview

Kernel code in Loki continues to execute at the supervisor privilege level, with access to all existing privileged supervisor instructions. This includes access to traditionally privileged state, such as control registers, the MMU, page tables, and so on. However, kernel code does not have direct access to instructions that modify tags or permission cache entries. Instead, it invokes the security monitor to manage the tags and the permission cache, subject to security checks that we will describe later.

The kernel requires word-level tags for two main reasons. First, existing C data structures often combine data with different security requirements in contiguous memory. For example, the security label field in a kernel object should not be writable by kernel code, but the rest of the object's data can be made writable, subject to the policy specified by the security label. Word-level tagging avoids the need to split up such data structures into multiple parts according to security requirements. Second, word-level tags reduce the overhead of placing a small amount of data, such as a 32-bit pointer or a 64-bit object ID, in a unique protection domain.

Although Loki enforces memory access control, it does not guarantee liveness. All of the kernel protection domains in LoStar participate in a cooperative scheduling protocol, explicitly yielding the CPU to the next protection domain when appropriate. Buggy or malicious kernel code can perform a denial of service attack by refusing to yield, yielding only to other colluding malicious kernels, halting the processor, misconfiguring interrupts, or entering an infinite loop. Liveness guarantees can be enforced at the cost of a larger trusted monitor, which would need to manage timer interrupts, perform preemptive scheduling, and prevent processor state corruption.

3 Operating system design

To illustrate how Loki can be used to minimize the amount of trusted code, we modified HiStar, an operating system designed to minimize the amount of trusted code, to take advantage of tags to enforce its security guarantees in a smaller TCB. The rest of this section first motivates our choice of the HiStar operating system, then provides a brief overview of HiStar, and finally describes the modifications required to port HiStar to Loki in detail.

3.1 OS choice rationale

Enforcing application security policies at a low level requires addressing two main problems. First, applications must be able to express their security policies to the underlying system in a uniform manner, so that their policies can then be enforced, and second, application-level names, like filenames, must be securely bound to low-level protection domains, like memory tag values.

Traditional Unix-like operating systems are not a particularly good fit for addressing these two problems. Unix provides a large number of protection mechanisms, from process isolation to file descriptor sharing to user IDs, which have poorly defined semantics [5] and are

cumbersome to use in practice for building secure applications [17]. At the same time, mapping Unix filenames to the underlying object (inode) and its protection domain involves many layers of translation in kernel code. All of this kernel code must be fully trusted, since any mis-translation can subvert the intent of a privileged application by causing it to access an arbitrary file or device.

HiStar was an appealing choice for this work because it addressed both of these problems. First, HiStar used a single kernel mechanism—information flow control—to implement all protection in the system, from emulating Unix security to expressing application security policies. This meant that extending the enforcement of this single mechanism into hardware would automatically enforce all higher-level security policies implemented using HiStar's protection mechanism. Second, as we will discuss later on, HiStar reduces all naming to a single flat object ID space managed by the kernel. This means that a secure binding between names and protection domains can be implemented by just providing this simple namespace in the trusted security monitor.

3.2 HiStar overview

HiStar's information flow control mechanism revolves around three key concepts. The first is the notion of a *category*—an opaque 61-bit ID managed by the kernel—which represents a particular kind of data in a system, and can restrict how that data can be accessed or modified. For example, a separate category is allocated for every process, to ensure that only threads in that process can access that process's address space. A separate category is also allocated for each file descriptor to control what processes are allowed to access it. Finally, Unix user accounts are also represented with categories that mirror the user's UID.

The second notion is that of a *label*, which is a set of categories. Every kernel object has a label associated with it, and the contents of an object is subject to the restrictions of every category in that object's label.

The final notion is that of thread *ownership* of categories, which defines threads that have access to data labeled in a particular fashion. For example, every thread typically has ownership of the category corresponding to its process, categories for any file descriptors it has access to, and the category of the Unix user on whose behalf the process is executing.

HiStar reduces the amount of trusted kernel code compared to traditional OSes by providing a simple, low-level kernel interface, consisting of six kernel object types: *segments*, *address spaces*, *devices*, *threads*, *gates*, and *containers*. Kernel objects are named by 61-bit object IDs that are unique over all time, and most application-level naming is reduced to the kernel's trusted object ID namespace. For example, Unix process IDs are object IDs of the container object representing the process. Pseudo-terminal (pty) IDs correspond to the object ID of a segment object storing that pseudo-terminal's control block. Even file and directory inodes correspond to object IDs of the segments and containers used to implement them, and the kernel's single-level store provides persistent disk storage.

The two kernel object types of particular interest in this paper are threads and gates. *Thread* objects are used to execute user-level code, and consist of a register set and the object ID of an address space object that defines the virtual address space for the corresponding process. A thread's label reflects the data that the thread could have potentially observed. Threads can dynamically adjust their label to observe secret data at runtime. By doing so, a thread gives up the right to modify any objects not also labeled with the secret data's category, thus transitively controlling information flow. However, a thread can only add restrictions to its label, not remove them. To ensure that threads cannot unilaterally read all secret data in the system by adjusting their labels, each thread has a *clearance*, which is a set of categories that a thread is allowed to add to its label. The thread's clearance serves to enforce a form of discretionary access control.

Gate objects provide a mechanism for protected control transfer, allowing a thread to switch to a particular entry point in another address space and protection domain. Gates can be thought of as an IPC mechanism, except that the client, instead of the server, provides the initial thread of execution. The gate's privileges are stored in the *label* and *clearance* associated with the gate.

The kernel provides a small number of operations (system calls) that can be performed on each type of kernel object by threads. For each operation, the kernel knows how information can flow as a result of the operation. Whenever a thread asks the kernel to perform an operation on another object, the kernel compares the thread's label to the label of the other object, and decides whether the labels allow the operation.

3.3 Minimizing trusted code

HiStar's design already provides a significantly smaller fully trusted kernel than a traditional Unix system, as shown in Figure 3 (a) and (b). Code implementing traditional Unix semantics is moved to an untrusted user-level library, while security policies, specified by either the Unix library or the application in terms of labels, are enforced by a much smaller kernel.

The Loki architecture allows us to further reduce HiStar's trusted code base, by enforcing a subset of HiStar's security guarantees with a small *security monitor* in a system called LoStar, as shown in Figure 3 (c). At a high level, the kernel in LoStar still enforces *information flow*

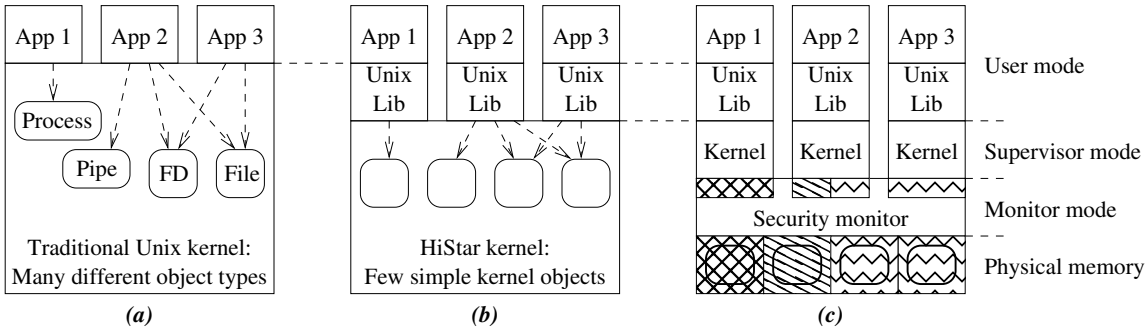

Figure 3: A comparison of operating system structure, showing (a) traditional Unix, (b) HiStar, and (c) LoStar. Vertical stacks correspond to different protection domains. The Unix library and kernel components in different protection domains in (c) are executing the same code with different privileges, similar to how Unix processes execute the same `libc` code with different privileges.

control: if a malicious application gains access to secret data, the kernel is responsible for ensuring that the malicious code cannot export this data outside of the system. However, the security monitor enforces a simpler *discretionary access control* policy, ensuring that objects can only be read or written by a thread whose label allows that operation, and ensuring that threads can only change their labels in approved ways (that is, not allowing a thread to arbitrarily lower its label or to raise it above its clearance). This effectively translates to enforcing Unix security policies that prevent one user's process from reading another user's files.

In LoStar, the kernel is no longer a single entity with a fixed set of privileges. Rather, there is a separate logical instance of the kernel associated with each running thread, and the kernel instance derives its privileges from the associated thread. (In HiStar, the kernel only has a notion of a thread, which has its own set of privileges, although multiple threads in the same Unix process, as implemented by the Unix library, often have the same privileges.) Moreover, all shared state in the system is managed by the security monitor, which means that one kernel instance cannot directly compromise another kernel instance by corrupting its data structures. As a result, the kernel can be viewed as just a library (much like libc, where the OS kernel maintains and protects all of libc's data), which means there is no longer any notion of a global kernel compromise. Instead, LoStar resembles a distributed system, in which many kernel instances cooperate in limited ways via the security monitor's mechanisms. The security monitor, in turn, protects different kernel instances from one another by write-protecting all kernel text and read-only data, and facilitates controlled sharing between kernels by allowing each kernel to explicitly specify labels to control read-write access to kernel objects.

HiStar's security policies, represented by labels on kernel objects, ultimately come from applications, often with the help of the Unix library. The security monitor maintains a mapping between these HiStar labels and Loki *tags*, and tags all physical pages of memory belonging to each kernel object with the tag corresponding to that object's label. This mapping is a fundamental part of the design, as hardware's fixed-width tag values cannot directly represent HiStar's variable-size labels. Our prototype uses the label's 32-bit physical address as the tag value. To ensure that tag values always refer to valid labels, the security monitor keeps a tag reference count for each label object, and avoids garbage collecting any referenced label. The monitor also keeps track of the Loki *permission cache* or P-cache for each thread's protection domain, and loads these permissions into the hardware cache on each context switch.

When the application or kernel code, running outside of monitor mode, accesses data with a particular tag for the first time, hardware raises a *tag exception*, which traps to the monitor. The monitor then looks up the HiStar label corresponding to the accessed tag value, compares it to the currently executing thread's HiStar label, and updates the permission cache accordingly.

Our design requires that applications be able to securely specify labels to the underlying kernel, which in turn relays them to the security monitor. While a compromised kernel instance could modify a label specified by an application to change the effective security policy, our design mitigates such attacks by treating different instances of the kernel as independent libraries that can only affect each other through the protected interface provided by the security monitor. As a result, our design trusts the kernel much in the same way that a traditional OS trusts libc to relay the application's security policy from the application to the kernel. More specifically, even if an attacker process were to compromise their instance of the kernel, this would not compromise other kernel instances, since the kernel code is write-protected by the security monitor, and all data structures shared by the kernels are maintained by the security monitor as well. Our design also achieves a form of "forward security": even if an attacker can compromise another process and its underlying kernel, and fully control its future

execution, it cannot change any labels that were already specified to the security monitor in the past.

3.4 Monitor functionality

Although most policies enforced by the LoStar security monitor translate directly into tags on physical memory, there are a few other guarantees that the monitor must provide which cannot be directly expressed with memory tagging. In particular, there are certain data structures, such as object labels, reference counts, and global hash tables, that should not be modified arbitrarily by untrusted kernel code. Instead, the security monitor protects these data structures by making the relevant fields read-only to application and kernel code, and providing a system-call-like interface for modifying these fields in a safe manner, as we will now describe.

First, the monitor protects object labels, which encode the security policies in our system, by using Loki's fine-grained tags. Each kernel object includes a pointer to a label object that describes the object's security policy. When a kernel object is allocated, the monitor sets the tag value for the object's label pointer, and for all words comprising the label object, to a special tag value that is readable but not writable by all kernel code. This allows kernel code to make its own access control decisions based on an object's label, but prevents potentially compromised kernel code from subverting the security by modifying labels.

Most of the state in a HiStar system resides in kernel objects, which have well-defined labels. However, the HiStar kernel also maintains one global data structure whose integrity is crucial for controlled sharing between mutually-distrustful kernels in different protection domains. This data structure is the object hash table, which maps object IDs to kernel objects, and it is implemented using chaining, so that each kernel object has a pointer to the next object in the same hash bucket.

The monitor ensures the integrity of the object ID to kernel object mapping by tagging the hash table structure, and the linked list pointer and object ID fields in every kernel object with a special *kernel-object* tag value. This tag value allows read but not write access for all protection domains. Loki's support for fine-grained word-level tagging simplifies the enforcement of kernel data structure integrity in this case. The integrity of the object ID mapping ensures that a user-level application, which uses object IDs to access its objects, will always access the correct object. An attacker that compromises the kernel running with different access rights will not be able to substitute other objects with the same object ID into the hash table.

The object type is also protected by the monitor, to ensure that one type of object cannot be mistaken for another. The contents of different types of kernel objects have different meanings, and the monitor associates certain privileges with thread and gate objects, based on their label. If the object's type was not protected, an attacker might be able to convince the security monitor to interpret the bit-level representation of a gate object as a thread with the privileges of the original gate object. Depending on the exact bit-level layout of these objects, this might result in the monitor executing arbitrary code in a thread with the gate's privileges. Enforcing the integrity of an object's type prevents these kinds of attacks.

The monitor also protects each object's reference count field, to ensure that a malicious kernel cannot deallocate a kernel object unless it controls every reference to that object. Every reference to a kernel object corresponds to a 64-bit memory location which holds the 61-bit ID of that kernel object. These references are protected by setting the low bit, called the *reference-holder* bit, of that memory location's tag to 1 (typically tags have the low bits set to 0, since tag values are page-aligned physical memory addresses of label objects). The monitor will only increment or decrement reference counts for an object if it also atomically sets or clears the *reference-holder* bit in the tag of a 64-bit word of memory storing that object ID. Thus, memory locations tagged as *reference-holders* are effectively capabilities to un-reference that object later on. To ensure these capabilities are not tampered with, the monitor only allows read access to tags with the *reference-holder* bit set, even if the tag with that bit cleared would have allowed write access.

Other linked lists of objects in the kernel, such as lists of waiting threads, are not protected by the security monitor in the same way as the global object hash table. Instead, kernels are free to arbitrarily manipulate all pointers in such linked lists. However, well-behaved kernel code can ensure that it is traversing a valid list of kernel objects by verifying that linked list pointers point to memory with the special *kernel-object* tag, and by verifying the object's type value at each step. Although malicious kernel code can corrupt the linked list and form an infinite loop, it cannot trick another kernel into accessing a kernel object of the wrong type, or any other piece of memory that is not a kernel object, in its traversal of a linked list. In the case of a list of waiting threads, this can result in lost or spurious wakeups, or more generally, denial of service, but not data corruption.

Finally, the monitor provides a narrow interface to perform a small number of operations on these integrity-protected data structures, which we describe in the next subsection. To provide this system-call-like interface, the monitor allocates a *monitor-call* tag that is not accessible to kernel code, and a special *monitor-call* memory word, tagged with the *monitor-call* tag value, which is used to invoke the monitor. When kernel code wants to

invoke a privileged monitor operation, it places its request in its registers and accesses the *monitor-call* word. This causes a tag exception, invoking monitor code. The monitor performs the requested operation, subject to security checks, and resumes kernel execution at the next instruction, skipping the memory access that caused the exception.

3.5 Monitor call API

The monitor call interface consists of a number of operations that cannot be safely implemented through memory tagging alone, which we will now describe. The first set of operations context-switch to a different protection domain:

- Switch to another thread, represented by a kernel thread object. LoStar implements cooperative scheduling between kernels. The monitor ensures the validity of the thread object, and loads the access rights associated with that thread object before executing its kernel code.
- Invoke the garbage collection code for a particular kernel object. The monitor conceptually keeps an idle protection domain associated with each kernel object, created when a kernel object is allocated, ready to garbage collect the kernel object once its reference count reaches zero. This protection domain has implicit rights over its respective kernel object and any reference counts that this object in turn holds.
- Call a function in a special protection domain, used for the page allocator. The monitor provides a fixed-depth stack for storing the caller's protection domain and execution state while the called protection domain executes (e.g. allocating or freeing a page of memory in the page allocator). The monitor provides the called protection domain with a fresh execution stack.
- Return from a cross-domain function call, passing the return values to the caller and restoring the caller's protection domain and execution state.

The monitor also provides operations to manipulate memory, such as pages and kernel objects:

- Change the tag for a range of memory, used to transfer memory between protection domains. Any protection domain that has read and write access to a range of memory can ask the monitor to assign any other non-reserved tag (that is, not *reference-holder*, *kernel-object*, and so on) to that range of memory. The page allocator is implemented as just another protection domain; allocating memory involves the allocator re-labeling one of its free pages with the caller's requested tag value, and freeing a tag involves re-labeling a page of memory with the allocator's tag value.
- Allocate a new kernel object with a particular type, label, and clearance. The monitor allocates an empty kernel object with a fresh object ID, places it on the object hash table, and returns the object pointer to the caller for further initialization. The monitor ensures that the label (and clearance, for threads and gates) of the new object is permitted for the currently executing thread.
- Atomically increment or decrement the refcount of a kernel object and, correspondingly, set-from-zero or clear-from-one the *reference-holder* bit in the tag of a 64-bit memory location storing the object ID of that kernel object. The monitor checks that the caller has read and write privileges over the tag of the memory location with the *reference-holder* bit cleared, and to avoid potential ID splicing attacks, disallows 64-bit memory locations that span pages.

Finally, the monitor provides operations to manipulate protection domains:

- Change the protection domain of the current thread by invoking a gate. The monitor verifies the validity of the supplied gate object, and checks that the caller is authorized to invoke the gate.
- Change the protection domain of the current thread by adjusting the label or clearance, as long as it is permitted by the thread's current label and clearance.
- Allocate a new category. The monitor grants ownership of the newly allocated category to the requesting thread.

The LoStar prototype incurs some performance overhead due to the introduction of a security monitor. In particular, any communication between two instances of the kernel running in different protection domains must now go through the security monitor, and the security monitor must be involved in the creation of new protection domains, as well as switches between protection domains. Section 5 will present a more detailed evaluation of the performance overheads incurred by the introduction of the security monitor.

3.6 Interrupts

LoStar must deal with two kinds of exceptions: traditional CPU exceptions, which include synchronous processor faults and asynchronous device interrupts, and Loki's tag exceptions, which will be described in more

detail in Section 4.3. The two exception mechanisms are independent, in that the CPU maintains two separate vector tables for the two kinds of exceptions, and only tag exceptions switch the processor into monitor mode.

Traditional synchronous CPU exceptions, such as page faults or divides by zero, are handled by the currently running kernel instance in LoStar, without switching into a different protection domain. Asynchronous device interrupts are also initially vectored to the currently running kernel instance, and can either be handled by the same kernel, or handed off to the device driver's protection domain. In the case of timer interrupts, the currently running kernel instance simply runs the scheduler, which picks another thread to execute and asks the monitor to switch to that thread's kernel instance. Network device interrupts, on the other hand, are handled by invoking the security monitor to pass the interrupt to the network device driver.

To simplify the security monitor, tag exceptions mask external interrupts when transitioning into monitor mode. However, the SPARC CPU can invoke synchronous register window overflow or underflow exceptions at almost any function call or return. As a result, the tag exception handler must install its own traditional CPU exception handlers before proceeding to execute C code in monitor mode. Since the traditional CPU exception mechanism does not transition the processor either in or out of monitor mode on its own, the security monitor's traditional exception handlers need not be any different than their non-monitor-mode counterparts.

3.7 Devices

One limitation of our prototype is that most device handling code is part of the trusted security monitor. Moreover, because the traditional interrupt mechanism does not switch the processor into monitor mode, device liveness relies on individual kernel instances handing off device interrupts to the driver in the security monitor. (However, if untrusted kernel code cannot clear the interrupt condition, the interrupt will be serviced as soon as the CPU starts executing a well-behaved kernel instance.) We believe that there are two approaches to reducing the amount of trusted device driver code, corresponding to two kinds of devices, as follows.

For devices that only handle data with a single label, such as a network card, a mechanism for controlling both device DMA and access to device registers would be sufficient for moving the device driver into a separate protection domain outside the fully-trusted monitor. The DMA control mechanism could use either memory tagging to define the set of tags accessible to each device, or an IOMMU mechanism like Intel's VT-d [1], although properly implementing protection through translation requires avoiding peer-to-peer bus transactions and other potential pitfalls [24].

The more difficult case is devices that handle differently-labeled data, such as the disk. While the disk device driver would likely remain in trusted code, we expect that support for lightweight tagging of on-disk data would allow moving some of the file system implementation into untrusted code. For example, a small amount of flash available in hybrid disk drives today could be used to store sector-level tag values, and track the label of data as it moves between RAM and disk.

4 MICROARCHITECTURE

Loki enables building secure systems by providing fine-grained, software-controlled permission checks and tag exceptions. This section discusses several key aspects of the Loki design and microarchitecture. Figure 4 shows the overall structure of the Loki pipeline.

4.1 Memory tagging

Loki provides memory tagging support by logically associating an opaque 32-bit tag with every 32-bit word of physical memory. Associating tags with physical memory, as opposed to virtual addresses, avoids potential aliasing and translation issues in the security monitor. These tags are cacheable, similar to data, and have identical locality.

Naively associating a 32-bit tag value with each 32-bit physical memory location would not only double the amount of physical memory, but also impact runtime performance. Setting tag values for large ranges of memory would be prohibitively expensive if it required manually updating a separate tag for each word of memory. Since tags tend to exhibit high spatial locality [29], our design adopts a *multi-granular* tag storage approach in which page-level tags are stored in a linear array in physical memory, called the *page-tag array*, allocated by the monitor code. This array is indexed by the physical page number to obtain the 32-bit tag for that page. These tags are cached in a structure similar to a TLB for performance. Note that this is different from previous work where page-level tags are stored in the TLBs and page tables [29]. Since we do not make any assumptions about the correctness of the MMU code, we must maintain our tags in a separate structure. The monitor can specify fine-grained tags for a page of memory on demand, by allocating a shadow memory page to hold a 32-bit tag for every 32-bit word of data in the original page, and putting the physical address of the shadow page in the appropriate entry in the linear array, along with a bit to indicate an indirect entry. The benefit of this approach is that DRAM need not be modified to store tags, and the tag storage overhead is proportional to the use of fine-grained tags.

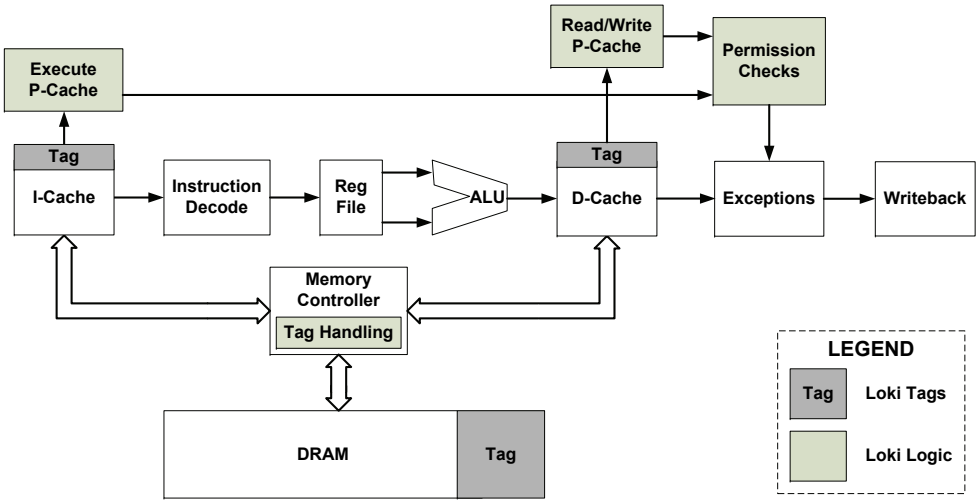

Figure 4: The Loki pipeline, based on a traditional pipelined SPARC processor.

4.2 Permissions cache

Fine-grained permission checks are enforced in hardware using a *permission cache*, or P-cache. The P-cache stores a set of tag values, along with a 3-bit vector of permissions (read, write, and execute) for each of those tag values, which represent the privileges of the currently executing code. Each memory access (load, store, or instruction fetch) checks that the accessed memory location's tag value is present in the P-cache and that the appropriate permission bit is set.

The P-cache is indexed by the least significant bits of the tag. A P-cache entry stores the upper bits of the tag and its 3-bit permission vector. The monitor handles P-cache misses by filling it in as required, similar in spirit to a software-managed TLB. All known TLB optimization techniques apply to the P-cache design as well, such as multi-level caches, separate caches for instruction and data accesses, hardware assisted fills, and so on.

The size of the P-cache, and the width of the tags used, are two important hardware parameters in the Loki architecture that impact the design and performance of software. The size of the P-cache affects system performance, and effectively limits the working set size of application and kernel code in terms of how many different tags are being accessed at the same time. Applications that access more tags than the P-cache can hold will incur frequent exceptions invoking the monitor code to refill the P-cache. However, the total number of security policies specified in hardware is not limited by the size of the P-cache, but by the width of the tag. In our experience, 32-bit tags provide both a sufficient number of tag values, and sufficient flexibility in the design of the tag value representation scheme. Finally, as we will show later in the evaluation of our prototype, even a small number of P-cache entries is sufficient to achieve good performance for a wide variety of workloads.

4.3 Tag exceptions

When a tag permission check fails, control must be transferred to the security monitor, which will either update the permission cache based on the tag of the accessed memory location, or terminate the offending protection domain. Ideally, the exception mechanism will be such that the trusted security handler can be as simple as possible, to minimize TCB size. Traditional trap and interrupt handling facilities do not conform with this, as they rely on the integrity of the MMU state, such as page tables, and privileged registers that may be modified by potentially malicious kernel code.

To address this limitation, Loki introduces a tag exception mechanism that is independent of the traditional CPU exception mechanism. On a tag exception, Loki saves exception information to a few dedicated hardware registers, disables the MMU, switches to the monitor privilege level, and jumps to the tag exception handler in the trusted monitor. The MMU must be disabled because untrusted kernel code has full control over MMU registers and page tables. For simplicity, Loki also disables external device interrupts when handling a tag exception. The predefined address for the monitor is available in a special register introduced by Loki, which can only be updated while in monitor mode, to preclude malicious code from hijacking monitor mode. As all code in the monitor is trusted, tag permission checks are disabled in monitor mode. The monitor also has direct access to a set of registers that contain information about the tag exception, such as the faulting tag.

5 PROTOTYPE EVALUATION

The main goal of this paper was to show that tagged memory support can significantly reduce the amount of trusted code in a system. To that end, this section reports on our prototype implementation of Loki hardware and the complexity and security of our LoStar software prototype. We then show that our prototype performs acceptably by evaluating its performance, and justify our hardware parameter choices by measuring the patterns and locality of tag usage.

In modifying HiStar to take advantage of Loki, we added approximately 1,300 lines of C and assembly code to the kernel, and modified another 300 lines of C code, but the resulting TCB is reduced by 6,400 lines of code—*more than a factor of two*. While Loki greatly reduces the amount of trusted code, we have no formal proof of the system's security. Instead, our current prototype relies manual inspection of both its design and implementation to minimize the risk of a vulnerability.

5.1 Loki prototype

To evaluate our design of Loki, we developed a prototype system based on the SPARC architecture. Our prototype is based on the Leon SPARC V8 processor, a 32-bit open-source synthesizable core developed by Gaisler Research [11]. We modified the pipeline to perform our security operations, and mapped the design to an FPGA board, resulting in a fully functional SPARC system that runs HiStar. This gives us the ability to run real-world applications and gauge the effectiveness of our security primitives.

Leon uses a single-issue, 7-stage pipeline. We modified its RTL code to add support for coarse and fine-grained tags, added the P-cache, introduced the security registers defined by Loki, and added the instructions that manipulate special registers and provide direct access to tags in the monitor mode. We added 6 instructions to the SPARC ISA to read/write memory tags, read/write security registers, write to the permission cache, and return from a tag exception. We also added 7 security registers that store the exception PC, exception nPC, cause of exception, tag of the faulting memory location, monitor mode flag, address of the tag exception handler in the monitor, and the address of the base of the page-tag array. Figure 4 shows the prototype we built.

We built a permission cache using the design discussed in Section 4.2. This cache has 32 entries and is 2-way set associative. During instruction fetch, the tag of the instruction's memory word is read in along with the instruction from the I-cache. This tag is used to check the Execute permission bit. Memory operations—loads and stores—index this cache a second time, using the memory word's tag. This is used to check the Read and Write permission bits. As a result, the permission cache is accessed at least once by every instruction, and twice by some instructions. This requires either two ports into the cache or separate execute and read/write P-caches to allow for simultaneous lookups. Figure 4 shows a simplified version of this design for clarity.

Parameter	Specification
Pipeline depth	7 stages
Register windows	8
Instruction cache	16 KB, 2-way set associative
Data cache	32 KB, 2-way set associative
Instruction TLB	8 entries, fully-associative
Data TLB	8 entries, fully-associative
Memory bus width	64 bits
Prototype Board	Xilinx University Program (XUP)
FPGA device	XC2VP30
Memory	512 MB SDRAM DIMM
Network I/O	100 Mbps Ethernet MAC
Clock frequency	65 MHz

Figure 5: The architectural and design parameters for our prototype of the Loki architecture.

Component	Block RAMs	4-input LUTs
Base Leon	43	14,502
Loki Logic	2	2,756
Loki Total	45	17,258
Increase over base	**5%**	**19%**

Figure 6: Complexity of our prototype FPGA implementation of Loki in terms of FPGA block RAMs and 4-input LUTs.

As mentioned in Section 4.1, we implement a multi-granular tag scheme with a page-tag array that stores the page-level tags for all the pages in the system. These tags are cached for performance in an 8-entry cache that resembles a TLB. Fine-grained tags can be allocated on demand at word granularity. We reserve a portion of main memory for storing these tags and modified the memory controller to properly access both data and tags on cached and uncached requests. We also modified the instruction and data caches to accommodate these tag bits.

We synthesized our design on the Xilinx University Program (XUP) board which contains a Xilinx XC2VP30 FPGA. Figure 5 summarizes the basic board and design statistics, and Figure 6 quantifies the changes made for the Loki prototype by detailing the utilization of FPGA resources. Note that the area overhead of Loki's logic will be lower in modern superscalar designs that are significantly more complex than the Leon. Since Leon uses a write-through, no-write-allocate data cache, we had to modify its design to perform a read-modify-write access on the tag bits in the case of a write miss. This change and its small impact on application performance would not have been necessary with a write-back cache. There was no other impact on the processor performance, as the permission table accesses and tag processing occur in parallel and are independent from data processing in all pipeline stages.

Lines of code	HiStar	LoStar
Kernel code	11,600 (trusted)	12,700 (untrusted)
Bootstrapping code	1,300	1,300
Security monitor code	N/A	5,200 (trusted)
TCB size: *trusted code*	**11,600**	**5,200**

Figure 7: Complexity of the original *trusted* HiStar kernel, the *untrusted* LoStar kernel, and the *trusted* LoStar security monitor. The size of the LoStar kernel includes the security monitor, since the kernel uses some common code shared with the security monitor. The bootstrapping code, used during boot to initialize the kernel and the security monitor, is not counted as part of the TCB because it is not part of the attack surface in our threat model.

5.2 Trusted code base

To evaluate how well the Loki architecture allows an operating system to reduce the amount of trusted code, we compare the sizes of the original, fully trusted HiStar kernel for the Leon SPARC system, and the modified LoStar kernel that includes a security monitor, in Figure 7. To approximate the size and complexity of the trusted code base, we report the total number of lines of code. The kernel and the monitor are largely written in C, although each of them also uses a few hundred lines of assembly for handling hardware traps. LoStar reduces the amount of trusted code in comparison with HiStar by more than a factor of two. The code that LoStar removed from the TCB is evenly split between three main categories: the system call interface, page table handling, and resource management (the security monitor tags pages of memory but does not directly manage them).

5.3 Performance

To understand the performance characteristics of our design, we compare the relative performance of a set of applications running on unmodified HiStar on a Leon processor and on our modified LoStar system on a Leon processor with Loki support. The application binaries are the same in both cases, since the kernel interface remains the same. We also measure the performance of LoStar while using only word-granularity tags, to illustrate the need for page-level tag support in hardware.

Figure 8 shows the performance of a number of benchmarks. Overall, most benchmarks achieve similar performance under HiStar and LoStar (overhead for LoStar ranges from 0% to 4%), but support for page-level tags is critical for good performance, due to the extensive use of page-level memory tagging. For example, the page allocator must change the tag values for all of the words in an entire page of memory in order to give a particular protection domain access to a newly-allocated page. Conversely, to revoke access to a page from a protection domain when the page is freed, the page allocator must reset all tag values back to a special tag value that no other protection domain can access. Explicitly setting tags for each of the words in a page incurs a significant performance penalty (up to 55%), and being able to adjust the tag of a page with a single memory write greatly improves performance.

Compute-intensive applications, represented by the *primes* and *gzip* workloads, achieve the same performance in both cases (0% overhead). Even system-intensive applications that do not switch protection domains, such as the system call and file system benchmarks, incur negligible overhead (0-2%), since they rarely invoke the security monitor. Applications that frequently switch between protection domains incur a slightly higher overhead, because all protection domain context switches must be done through the security monitor, as illustrated by the *IPC ping-pong* workload (2% overhead). However, LoStar achieves good network I/O performance, despite a user-level TCP/IP stack that causes significant context switching, as can be seen in the *wget* workload (4% overhead). Finally, creation of a new protection domain, illustrated by the *fork/exec* workload, involves re-labeling a large number of pages, as can be seen from the high performance overhead (55%) without page-level tags. However, the use of page-level tags reduces that overhead down to just 1%.

5.4 Tag usage and storage

To evaluate our hardware design parameters, we measured the tag usage patterns of the different workloads. In particular, we wanted to determine the number of pages that require fine-grained word-level tags versus the number of pages where all of the words in the page have the same tag value, and the working set size of tags—that is, how many different tags are used at once by different workloads. Figure 9 summarizes our results for the workloads from the previous sub-section.

The results show that all of the different workloads under consideration make moderate use of fine-grained tags. The primary use of fine-grained tags comes from protecting the metadata of each kernel object. For example, workloads with a large number of small files, each of which corresponds to a separate kernel object, require significantly more pages with fine-grained tags compared to a workload that uses a small number of large files. Since Loki implements fine-grained tagging for a page by allocating a shadow page to store a 32-bit tag for each 32-bit word of the original page, tag storage overhead for such pages is 100%. On the other hand, pages storing user data (which includes file contents) have page-level tags, which incur a much lower tag storage overhead of $4/4096 \approx 0.1\%$. As a result, overall tag storage overhead is largely influenced by the average size of kernel objects cached in memory for a given workload. We expect that it's possible to further reduce tag storage overhead for fine-grained tags by using a more compact in-memory representation, like the one used by Mondriaan Memory Protection [33], or to rearrange ker-

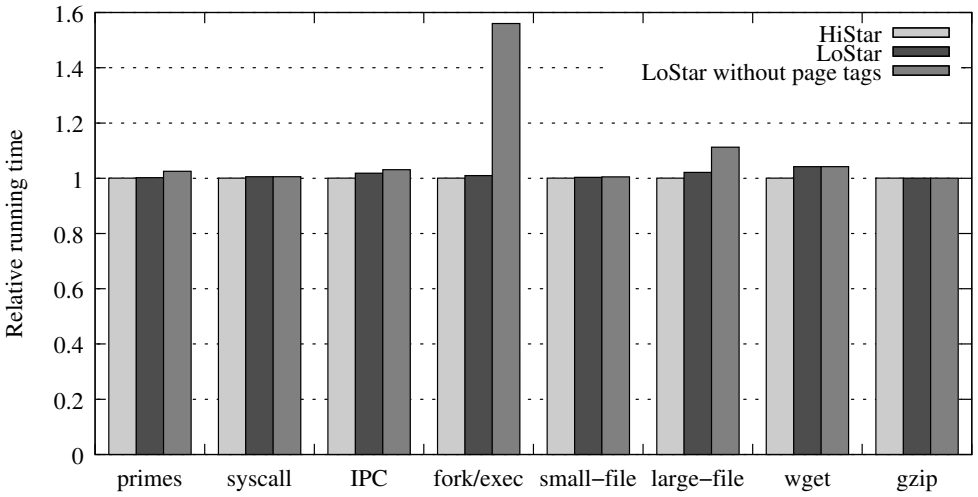

Figure 8: Relative running time (wall clock time) of benchmarks running on unmodified HiStar, on LoStar, and on a version of LoStar without page-level tag support, normalized to the running time on HiStar. The *primes* workload computes the prime numbers from 1 to 100,000. The *syscall* workload executes a system call that gets the ID of the current thread. The *IPC ping-pong* workload sends a short message back and forth between two processes over a pipe. The *fork/exec* workload spawns a new process using `fork` and `exec`. The *small-file* workload creates, reads, and deletes 1000 512-byte files. The *large-file* workload performs random 4KB reads and writes within a single 4MB file. The *wget* workload measures the time to download a large file from a web server over the local area network. Finally, the *gzip* workload compresses a 1MB binary file.

Workload	primes	syscall	IPC	fork/exec	small files	large files	wget	gzip
Fraction of memory pages with word-granularity tags	40%	49%	54%	65%	58%	3%	18%	16%
Maximum number of concurrently accessed tag values	12	11	18	24	13	13	30	12

Figure 9: Tag usage under different workloads running on LoStar.

nel data structures to keep data with similar tags together, although doing so would likely increase complexity either in hardware or software.

Finally, all workloads shown in Figure 9 exhibit reasonable tag locality, requiring only a small number of tags at time. This supports our design decision to use a small fixed-size hardware permission cache.

6 Related work

Since this paper describes a combination of hardware and software, we will discuss related work in these two areas in turn.

6.1 Hardware

Many hardware protection architectures have been previously proposed. Multics [25] introduced hierarchical protection rings which were used to isolate trusted code in a coarse-grained manner. *x*86 processors also have 4 privilege levels, but the page table mechanism can only distinguish between two effective levels. However, application security policies are often not hierarchical, and Loki's 32-bit tag space provides a way of representing a large number of such policies in hardware.

The Intel i432 and Cambridge CAP systems, among others [20], augment the way applications name memory with a capability, which allows enforcing non-hierarchical security policies by controlling access to capabilities, at the cost of changing the way software uses pointers. Loki associates security policies with physical memory, instead of introducing a name translation mechanism to perform security checks. As a result, the security policy for any piece of data in Loki is always unambiguously defined, regardless of any aliasing that may be present in higher-level translation mechanisms.

The protection lookaside buffer (PLB) [16] provides a similarly non-hierarchical access control mechanism for a global address space (although only at page-level granularity). While the PLB caches permissions for virtual addresses, Loki's permissions cache stores permissions in terms of tag values, which is much more compact, as Section 5.4 suggests.

The IBM system i [13] associates a one-bit tag with physical memory to indicate whether the value represents a pointer or not. Similarly, the Intel i960 [14] provides a one-bit tag to protect kernel memory. Loki's tagged memory architecture is more general, providing a large number of protection domains.

Mondriaan Memory Protection (MMP) [33] provides lightweight, fine-grained (down to individual memory words) protection domains for isolating buggy code. However, MMP was not designed to reduce the amount

of trusted code in a system. Since the MMP supervisor relies on the integrity of the MMU and page tables, MMP cannot enforce security guarantees once the kernel is compromised. Loki extends the idea of lightweight protection domains to physical resources, such as physical memory, to achieve benefits similar to MMP's protection domains with stronger guarantees and a much smaller TCB. Moreover, this paper describes how a fine-grained memory protection mechanism can be used to extend the enforcement of application security policies all the way down into hardware.

The Loki design was initially inspired by the Raksha hardware architecture [9]. However, the two systems have significant design differences. Raksha maintains four independent one-bit tag values (corresponding to four security policies) for each CPU register and each word in physical memory, and propagates tag values according to customizable tag propagation rules. Loki, on the other hand, maintains a single 32-bit tag value for each word of physical memory (allowing the security monitor to define how multiple security policies interact), does not tag CPU registers, and does not propagate tag values. Raksha's propagation of tag values was necessary for fine-grained taint tracking in unmodified applications, but it could not enforce write-protection of physical memory. Conversely, Loki's explicit specification of tag values works well for a system like HiStar, where all state in the system already has a well-defined security label that controls both read and write access.

There has also been significant work on hardware support for other types of security mechanisms, such as dynamic information flow tracking, to prevent attacks such as buffer overflows [6, 8, 9, 29]. Hardware designs for preventing information leaks in user applications have also been proposed [28, 32], although these designs do not attempt to reduce the TCB size. None of these designs provide a sufficiently large number of protection domains needed to capture different application security policies. Moreover, enforcement of information flow control in hardware has inherent covert channels relating to the re-labeling of physical memory locations. HiStar's system call interface avoids this by providing a virtually unlimited space of kernel object IDs that are never relabeled.

6.2 Software

Many operating systems, including KeyKOS [4], EROS [27], and HiStar [35], provide strong isolation of application code using a small, fully trusted kernel. However, existing hardware architectures fundamentally require that the fully trusted kernel include code to manage page tables, device drivers, and so on, in order to provide different protection domains for user-level code. LoStar can enforce certain security guarantees using a significantly smaller trusted code base, by directly specifying security policies for physical resources in hardware. This allows the fully trusted code base to exclude complex code such as page table management and device drivers. Even for an operating system such as HiStar, where the kernel is already small, Loki allows significantly reducing the trusted code size.

A number of systems attempt to provide some guarantees even in the case of buggy or malicious kernel code. Separation kernels [23] and virtual machine monitors [15] provide strong isolation between multiple processes on a single machine. SecVisor [26] ensures kernel code integrity in a small hypervisor. Proxos [31] allows sensitive applications to partition trust in operating system abstractions by using an untrusted kernel for certain peripheral functionality. Flicker [21] enables tamper-proof code execution without trusting the underlying operating system. Nooks [30] and Mondrix [34] isolate potentially buggy device driver code in the Linux kernel. These systems enforce relatively static security policies that do not directly map onto application security goals. As a result, applications running on top of these systems must provide their own security enforcement mechanisms. In contrast, LoStar maps application security policies onto the underlying hardware protection mechanisms, providing strong enforcement of application security.

Singularity [12] avoids the need for hardware protection mechanisms by relying on type safety instead. However, we believe that Singularity could also benefit from associating security policies with data, perhaps using types.

The VMM security kernel [15] provides strong isolation across multiple virtual machines with limited sharing. Using the Loki architecture, the VMM security kernel could be implemented using significantly less trusted code, by directly specifying security policies for physical hardware resources used by the different virtual machines. Unlike LoStar, the VMM security kernel provides very limited sharing. A virtual machine monitor could adopt an interface similar to that provided by Loki to enforce security policies on behalf of applications running inside a virtual machine.

Overshadow [7] aims to protect application data in an untrusted OS by using a virtual machine monitor. One of the most complex aspects of Overshadow is providing a secure binding between application names (such as Unix pathnames) and protection domains. LoStar addresses this problem by relying on HiStar's design which reduces all naming to a single 61-bit kernel object namespace. As a result, LoStar needs only to ensure the integrity of a single flat namespace in the trusted security monitor, which is simpler than a hierarchical file system.

7 CONCLUSION

This paper showed how hardware support for tagged memory can be used to enforce application security policies. We presented Loki, a hardware tagged memory architecture that provides fine-grained, software-managed access control for physical memory. We also showed how HiStar, an existing operating system, can take advantage of Loki by directly mapping application security policies to the hardware protection mechanism. This allows the amount of trusted code in the HiStar kernel to be reduced by over a factor of two. We built a full-system prototype of Loki by modifying a synthesizable SPARC core, mapping it to an FPGA board, and porting HiStar to run on it. The prototype demonstrates that our design can provide strong security guarantees while achieving good performance for a variety of workloads in a familiar Unix environment.

ACKNOWLEDGMENTS

We thank Silas Boyd-Wickizer for porting HiStar to the SPARC processor. We also thank Bryan Ford, David Mazières, Michael Walfish, the anonymous reviewers, and our shepherd, Galen Hunt, for their feedback. This work was funded by NSF Cybertrust award CNS-0716806, by NSF award CCF-0701607, by joint NSF Cybertrust and DARPA grant CNS-0430425, by NSF through the TRUST Science and Technology Center, and by Stanford Graduate Fellowships supported by Cisco Systems and Sequoia Capital.

REFERENCES

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Weigert. Intel Virtualization Technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, August 2006.

[2] D. E. Bell and L. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, March 1976.

[3] K. J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE Corp., Bedford, MA, April 1977.

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

[5] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proc. of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[6] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. of the International Conference on Dependable Systems and Networks*, Yokohama, Japan, June 2005.

[7] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. of the 13th ASPLOS*, Seattle, WA, March 2008.

[8] J. R. Crandall and F. T. Chong. MINOS: Control data attack prevention orthogonal to memory model. In *Proc. of the 37th Intl. Symposium on Microarchitecture*, Portland, OR, December 2004.

[9] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proc. of the 34th ISCA*, San Diego, CA, June 2007.

[10] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proc. of the 20th SOSP*, pages 17–30, Brighton, UK, October 2005.

[11] Gaisler Research. LEON3 SPARC Processor. `http://www.gaisler.com`.

[12] G. C. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft, Redmond, WA, October 2005.

[13] IBM Corporation. IBM system i. `http://www-03.ibm.com/systems/i`.

[14] Intel Corporation. Intel i960 processors. `http://developer.intel.com/design/i960/`.

[15] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A VMM security kernel for the VAX architecture. In *Proc. of the 1990 IEEE Symposium on Security and Privacy*, pages 2–19, Oakland, CA, May 1990.

[16] E. Koldinger, J. Chase, and S. Eggers. Architectural support for single address space operating systems. Technical Report 92-03-10, University of Washington, Department of Computer Science and Engineering, March 1992.

[17] M. Krohn. Building secure high-performance web services with OKWS. In *Proc. of the 2004 USENIX*, June–July 2004.

[18] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. of the 21st SOSP*, Stevenson, WA, October 2007.

[19] B. Lampson, M. Abadi, M. Burrows, and E. P. Wobber. Authentication in distributed systems: Theory and practice. *ACM TOCS*, 10(4):265–310, 1992.

[20] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[21] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. How low can you go? Recommendations for hardware-supported minimal TCB code execution. In *Proc. of the 13th ASPLOS*, Seattle, WA, March 2008.

[22] President's Information Technology Advisory Committee (PITAC). Cybersecurity: A crisis of prioritization. `http://www.nitrd.gov/pitac/reports/20050301_cybersecurity/cybersecurity.pdf`, February 2005.

[23] J. M. Rushby. Design and verification of secure systems. In *Proc. of the 8th SOSP*, pages 12–21, Pacific Grove, CA, December 1981.

[24] J. Rutkowska and R. Wojtczuk. Preventing and detecting Xen hypervisor subversions. `http://invisiblethingslab.com/bh08/part2-full.pdf`, August 2008.

[25] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Comm. of the ACM*, 15(3):157–170, 1972.

[26] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proc. of the 21st SOSP*, pages 335–350, October 2007.

[27] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proc. of the 17th SOSP*, December 1999.

[28] W. Shi, H.-H. Lee, G. Gu, L. Falk, T. Mudge, and M. Ghosh. InfoShield: A security architecture for protecting information usage in memory. In *Proc. of the 12th HPCA*, Austin, TX, February 2006.

[29] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. of the 11th ASPLOS*, Boston, MA, October 2004.

[30] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM TOCS*, 23(1), 2005.

[31] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proc. of the 7th OSDI*, pages 279–292, Seattle, WA, November 2006.

[32] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proc. of the 37th International Symposium on Microarchitecture*, Portland, OR, December 2004.

[33] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *Proc. of the 10th ASPLOS*, San Jose, CA, October 2002.

[34] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proc. of the 20th SOSP*, pages 31–44, October 2005.

[35] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th OSDI*, pages 263–278, Seattle, WA, November 2006.

[36] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proc. of the 5th NSDI*, pages 293–308, San Francisco, CA, April 2008.