ThreadLock: Native Principal Isolation Through Memory Protection Keys

William Blair
wdblair@bu.edu
Boston University
Boston, MA, USA

William Robertson
wkr@ccs.neu.edu
Northeastern University
Boston, MA, USA

Manuel Egele
megele@bu.edu
Boston University
Boston, MA, USA

ABSTRACT

Inter-process isolation has been deployed in operating systems for decades, but secure intra-process isolation remains an active research topic. Achieving secure intra-process isolation within an operating system process is notoriously difficult. However, viable solutions that securely consolidate workloads into the same process have the potential to be extremely valuable.

In this work, we present native principal isolation, a technique to restrict threads' access to process memory by enforcing intra-process security policies defined over a program's application binary interface (ABI). A separate memory protection mechanism then enforces these policies. We present ThreadLock, a system that enforces native principal isolation policies using memory protection keys (MPKs) present on recent Intel CPUs. We demonstrate that ThreadLock efficiently restricts access to both thread-local data and sensitive information present in real workloads. We show how ThreadLock protects data within 3 real world applications, including the Apache web server, Redis in-memory data store, and MySQL relational database management system (RDBMS) with little performance overhead (+1.06% in the worst case). Furthermore, we show ThreadLock stops real world attacks against these popular programs. Our results show that native principal isolation is expressive enough to define effective intra-process security policies for real programs and that these policies may be enforced using MPKs without requiring any change to a program's source or binary.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Memory Protection Keys; Intra-Process Isolation; Policy Based Defenses; Hardware Security; Memory Safety

ACM Reference Format:
William Blair, William Robertson, and Manuel Egele. 2023. ThreadLock: Native Principal Isolation Through Memory Protection Keys. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579856.3595797

ASIA CCS '23, July 10–14, 2023, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-0098-9/23/07…$15.00
https://doi.org/10.1145/3579856.3595797

1 INTRODUCTION

Protecting mission critical services and consumer devices from evolving threats is a vital and unending task. Modern operating systems and hardware provide robust solutions for inter-process isolation, but achieving intra-process isolation is less supported by commodity systems and represents an ongoing research effort. Successful intra-process isolation techniques can provide significant value by limiting the security consequences of compromised program components within an individual process. Furthermore, consolidating multiple workloads securely within the same process has the potential to lower operating costs and make more efficient use of computing resources. Intra-process isolation can take different forms, including isolating individual code segments and memory to the degree supported by operating systems and hardware. In this work, we propose an intra-process isolation policy language for selectively restricting threads' access to sensitive data using recent memory protection hardware features.

Isolating sensitive data used by mission critical services, personal devices, and embedded systems has previously been accomplished using secure enclaves [35], control-flow enforcement [3], and hardware memory protection mechanisms [37]. Memory protection keys (MPK) available in Intel CPUs are a recent addition to Intel's memory protection mechanisms. MPKs allow a developer to restrict threads' access to individual protection domains in a process address space using a simple interface that introduces low performance overhead. Prior work has successfully used MPKs to secure sensitive data in applications through direct program transformations (i.e., annotations) or to implement intra-process sandboxes.

Prior work has employed this novel primitive to protect the just-in-time (JIT) Javascript interpreter within a web browser [39], a browser's high-performance memory allocators [16], language runtimes [29], and sensitive cryptographic data such as keys used by the transport layer security (TLS) protocol [28]. These approaches provide workloads tangible security benefits by preventing adversaries within a process from accessing or corrupting important application data-structures. Furthermore, developers can protect additional components by introducing annotations at relevant locations within a source tree.

To limit more capable adversaries or support running untrusted code, other approaches use MPKs to define full-fledged intra-process sandboxes [46]. In contrast to isolating data, intra-process sandboxes isolate individual program modules within a process and prevent adversaries from observing or modifying process data located outside of the sandbox. In this setting, programs may directly benefit from using MPKs without any changes to their source or binary, but the program's execution environment changes significantly to accommodate sandbox monitors. These monitors are necessary to

verify the activity of untrusted code. This can be done statically, by scanning instruction sequences in a binary for illegal instructions [46], or dynamically by defining a hardware monitor that scans for illegal instructions during execution [19]. Alternatively, strict system call gates can use nested filtering to prevent adversaries from evading a sandbox via confused deputy attacks [17, 41]. A more adaptive sandbox can rely on a `ptrace` monitor to scan for illegal instructions by incrementally installing hardware breakpoints at reachable destinations throughout execution [47].

While prior works have shown MPKs can provide security benefits to widely used programs, they often require changing the program's source code. This can be done through direct source modifications or through annotations on relevant components that cause compiler transformations to insert code at build time. Furthermore, MPKs typically protect extremely high value targets within a process, such as a web server's TLS private key [46], but valuable user data may be left unprotected. The protected software often has a variety of use cases, and conceiving every useful security policy at build time may not always be feasible.

Intra-process sandboxes that detect adversaries either through instruction scanning or runtime monitoring may not require changes to a program's source. However, these methods can require significant runtime introspection into the program's execution. Depending on an adversary's capabilities within the sandbox, the cost of introspection may impact the program's normal operation. As previous work has shown, this introspection may also be incomplete, and allow adversaries to break the sandbox by implementing confused deputy attacks against the operating system kernel [17]. In these attacks, an adversary can trick the kernel into acting on another sandbox by calling a seemingly benign system call, such as `madvise` [41]. Furthermore, an intra-process sandbox may be unnecessary for use cases that run trusted code and simply need to limit access to sensitive data within the process.

A compromise between rewriting programs to protect individual components with MPKs and creating full-fledged intra-process sandboxes can be obtained by enforcing intra-process security policies defined over a program's application binary interface (ABI). The ABI encompasses the entire environment necessary to run an individual program, which includes the external functions a program may invoke during execution. This interface permits defining lightweight security policies that protect sensitive user and application data. For example, if a developer observes that a thread within a general purpose program can handle sensitive data, they could author a policy to grant the thread access using MPKs. Accomplishing this with previous approaches either requires changing the source code of a given program or embedding expensive runtime monitors in production environments where changes that alter a process' capabilities (to support monitoring) or affect the underlying system may take significant amount of time to vet. In contrast, simply deploying an additional library dependency that enforces an intra-process security policy can be done without changing a program nor altering internal data-structures. Furthermore, policy enforcement is limited to intercepting external function calls, as opposed to monitoring the internal details of a whole process.

We propose native principal isolation, a technique for defining and enforcing intra-process security policies over native programs' ABIs. Native principal isolation policies restrict threads' access to sensitive data held by principals. In this work, we consider principals as individual threads, or entities identifiable from an ABI. For example, a memory allocator may be identified by an external allocation routine used by a program. Native principal isolation enforces these intra-process security policies using a memory protection mechanism.

We present ThreadLock, a system for enforcing native principal isolation policies using MPKs available on recent Intel CPUs. To evaluate ThreadLock's ability to implement intra-process isolation on real workloads, we author native principal isolation policies for the widely used Apache web server, Redis in-memory data store, and MySQL relational database management system (RDBMS). These programs are natural candidates for evaluating ThreadLock since they both support multi-threaded modes of operation, and, in the case of Redis and MySQL, exercise ThreadLock's ability to protect both thread-local and sensitive data shared across multiple threads. All of these applications are widely used, either to power 32.1% of public websites [12], to perform inter-process communication (IPC) in distributed system components, or as a web application's database server. We argue that these applications' wide production use and diverse domains make them suitable for evaluating ThreadLock's ability to enforce intra-process security policies. Previous approaches overcome the limited number of protection domains available in hardware by virtualizing MPKs. Virtualizing MPKs increases the number of available protection domains. This increases the number of principals a policy can use. ThreadLock could benefit from prior works that make MPKs a virtual resource in the kernel [38], split applications into virtual machine sandboxes [24], or carve out virtual domains in processes [51].

During our evaluation, we measure the performance overhead incurred by enforcing these policies while stress testing each individual program. Furthermore, we confirm that ThreadLock successfully stops real world attack vectors that are documented by assigned CVEs. We argue that these results show native principal isolation is expressive enough to define effective intra-process isolation policies for general purpose workloads and that ThreadLock enforces these policies efficiently.

In summary, we make the following contributions in this paper.

- We introduce native principal isolation as a generic technique for defining intra-process policies using a program's ABI and enforcing these policies using memory protection primitives (see Section 3). Native principal isolation requires no modification to a program's binary, source, or internal data-structures which makes it ideal for protecting real world programs.
- We introduce ThreadLock, a system that enforces native principal isolation policies using MPKs available on recent Intel CPUs.
- We present our prototype implementation of ThreadLock which we evaluate over 3 widely used programs, including the Apache web server, Redis in-memory data store, and MySQL RDBMS. We show that ThreadLock isolates memory holding sensitive application data (i.e., rendered application response data, data store contents, and user credentials). Furthermore, the performance overhead incurred by ThreadLock is limited and is often comparable to the baseline. Throughout our evaluation, we observed the performance overhead stay below 1.06%.

In the interest of open science, the source code for ThreadLock can be found online[1].

2 BACKGROUND AND THREAT MODEL

In this section, we provide background on memory protection keys (MPK). We demonstrate how MPKs provide a convenient intra-process isolation mechanism for protecting data. In addition, we discuss the threat model we assume in this work.

2.1 Memory Protection Keys

Memory protection keys (MPKs) are a recent feature added to Intel CPUs that allow operators to restrict an individual thread's access to a process address space. MPKs are supported by an additional 4-bit field to the page table entry within the memory management unit (MMU) available on recent Intel CPUs. Every page table entry that maps a virtual page to a physical page contains a protection key entry at bits 59-62. Every page assumes a value of 0 by default, and the kernel can assign a given page to any of the 16 protection key domains using privileged instructions. A protection key is a value that refers to one of these 16 domains, and a protection domain refers to all the pages associated with a given protection key.

On recent kernels, user processes may also alter a page's protection key using a variant of the `mprotect` system call. Each CPU thread in a user space process holds a protection key register for user pages (PKRU) that designates the thread's protection key permissions. The PKRU register represents a bitmap of permissions for each protection key, where each protection key k occupies two bits in the bitmap, one to designate access permissions and a second for write permissions. Whenever the CPU executes a load or store instruction, the CPU obtains the relevant page's protection key k from the MMU and consults the appropriate bit located in k's position in the PKRU register. When a load instruction is executed, then the "access" bit is checked. Likewise, the "write" bit is checked when the CPU executes a store instruction. In either case, if the bit is set, the instruction is denied, and the hardware raises an interrupt. This causes the operating system to send a segmentation violation signal to the thread's process which will, by default, stop the process. Otherwise, the instruction is allowed. User space programs may change a thread's permissions by writing to the PKRU register using a special `wrpkru` instruction. In addition, the `xrstor` instruction can escalate PKRU privileges with malicious processor state. While 16 keys are available for restricting access and writes, protection domain 0 is the default domain for all pages. Restricting access to protection domain 0 may inhibit programs' normal operation, since a thread would no longer be able to read from or write to any memory bound to the default domain. This restriction may benefit some use cases, but in this work all threads retain access to protection domain 0 to ensure the protected program's normal operation.

In Figure 1, thread X is configured to access and write the protection domain k's pages. However, MPKs prevent a malicious thread Y from doing the same. In this setting, a developer has configured PKRU for thread X to have full access to pages marked with protection key k by clearing the relevant bits reserved for k in X's PKRU register. Likewise, thread Y is denied access by setting each bit for protection key k in Y's PKRU register. This allows thread X to both access and write pages marked with protection key k. However, if an adversary were to cause thread Y to read or alter protection domain k's pages in any way, the MMU will observe that Y's PKRU forbids this interaction and will cause a segmentation violation in response. This prevents any disclosure or corruption of thread X's data by another thread running in the process. In Section 3, we use this simple primitive to define a policy language that allows developers and operators to isolate individual threads and data using intra-process security policies (see Section 3). ThreadLock takes such an intra-process security policy and synthesizes a shared library that enforces the policy on the target workload.

[1] https://github.com/BUseclab/threadlock

2.2 Policy Based Defenses

In this work, we propose that developers define intra-process security policies that allow ThreadLock to restrict access to sensitive data within a program. Previous works have also proposed protection mechanisms that require authoring security policies tailored to a program. Requiring developers to author security policies is in line with prior research. For example, using MPKs to restrict access to sensitive pointers requires stateful policies to define which pointers must be restricted, and access policies can vary between programs [27]. Section 6 describes additional security defenses that require developers to author security policies.

2.3 Design Assumptions and Threat Model

One limitation imposed by MPKs is that each page table entry can only be associated with a single protection key. Therefore, multi-threaded programs that allocate memory with page level granularity are best suited for ThreadLock. Multi-threaded programs that allocate memory with smaller granularity can also benefit from ThreadLock by treating allocation routines as trusted components. This enables more applications to use ThreadLock, but with the drawback of incurring less isolation between threads. In Section 5 we show examples of how popular multi-threaded programs like Redis and MySQL benefit from selectively granting threads access to sensitive data.

In this work, we assume the following threat model.

- The adversary's primary goal is to disclose or corrupt sensitive data. Sensitive data may be held within a victim thread or within a program component, such as a credential store. For example, an unprivileged user could escalate their privileges by obtaining an administrator's security token held within a separate thread.
- To this end, the adversary can statically inspect the protected program and all of its library dependencies.
- The adversary can influence the program through a remote socket. We do not assume the adversary has any control over the victim process' command line arguments or files on the system, beyond what he can influence through a socket.
- Finally, the adversary has access to a vulnerability in the program or any of its libraries that allow him to either disclose or corrupt data held by principals, but the adversary has no ability to execute arbitrary code.

We emphasize that the adversary lacks any ability to hijack control of the process protected by ThreadLock; his main objective is to disclose or corrupt sensitive application data. This implies that the adversary cannot gain the ability to issue either the `wrpkru` or `xrstor`

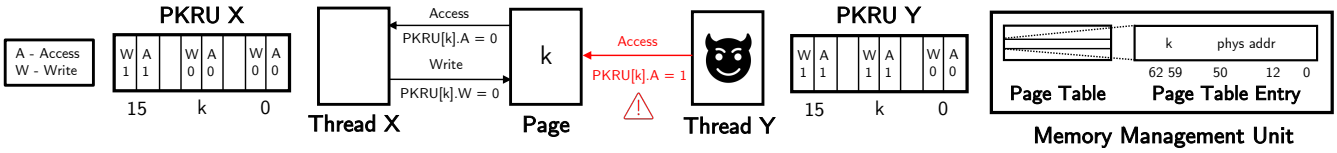

Figure 1: Memory protection keys (MPKs) isolating memory belonging to individual threads.

program	$P ::= \overline{\rho}$
policy	$\rho ::= pr\ prid\ \overline{stmt}$
symbol	sym
ABI function	$function$
address	ℓ
principal ID	$prid ::= sym \mid function \mid \ell$
principal	$pr ::= thread \mid abstract$
operation	$op ::= \textbf{tag} \mid \textbf{untag} \mid \textbf{grant} \mid \textbf{revoke}$
type	$\tau ::= \square \mid sym \mid op\ \tau \mid \tau \rightarrow \tau$
statement	$stmt ::= \langle function, \tau \rangle \mid \textbf{loop}\ \overline{stmt} \mid \langle op, prid \rangle$

Figure 2: The native principal isolation policy syntax.

instructions to escalate privileges and fetch victim data. Utilizing existing control-flow integrity (CFI) mechanisms [13] like those found within Intel control-flow enforcement technology (CET) [44] can help realize this restriction. In some cases, indirect branch tracking (IBT) provided by CET may be too course-grained, CET hardware is unavailable, or the operating system has not yet made full use of CET's features [10]. In these cases, efficient software-based CFI mechanisms can be used instead [5].

3 OVERVIEW

In this section, we introduce a novel intra-process security technique called native principal isolation which restricts threads' access to memory by enforcing policies over production binary artifacts. We present a syntax and semantics for defining native principal isolation policies using a program's ABI. Finally, we describe a motivating policy that isolates both thread-local data and sensitive global memory in a relational database (see Section 3.2).

3.1 Native Principal Isolation

Native principal isolation uses a memory protection mechanism to enforce intra-process isolation policies defined over a multi-threaded binary program's application binary interface (ABI). In this work, we consider policies restricted to the external symbols imported by a binary program which makes up a single component of the program's ABI [4]. The principals given in each policy are represented as either individual threads running in a workload or abstractions that share data amongst threads, such as a memory allocator. Native principal isolation transforms each policy into a finite automaton (FA) for enforcement. In this work, we assume a deterministic FA. The FA intercepts calls to functions in a program's ABI that are referenced by a policy. A function from the program's ABI is referred to as an ABI function. Each state in the FA represents a location in the policy, and edges represent policy statements. The syntax of native principal isolation policies is shown in Figure 2. In the following, we describe the semantics of this policy language.

Every ThreadLock policy consists of one or more policies assigned to specific principals. Principals and their associated policies are declared via *pr*. A principal may either be *abstract* or a *thread*. Abstract principals contain policies for specific ABI functions independent of a calling thread (e.g., a shared memory allocator's functions). Abstract principals are identified by a symbol *sym* which uniquely identifies the principal in the ThreadLock policy. A *thread* principal associates a policy with a specific thread entrypoint. The thread entrypoint can refer to a specific ABI function *function* or an address ℓ that points to a function in memory. In either case, ThreadLock uses the address of the entrypoint to activate the thread's policy after thread creation (see Section 4.3). Each principal policy consists of a sequence of statements $\overline{stmt}$. The sequence $\overline{stmt}$ describes when and how to restrict the thread's access to memory. Each statement may refer to an ABI function given by *function*. A called ABI function matches a statement if the statement contains an ABI function (e.g., $\langle function, \tau \rangle$) and *function* equals the called ABI function. Loops may be defined with the **loop** statement. Every **loop** iteratively enforces a sequence of statements $\overline{stmt}$ indefinitely by default. The loop can only stop once the program calls an ABI function that matches a statement that follows the loop.

When the program calls an ABI function, ThreadLock first checks whether the ABI function matches an abstract policy. Since abstract principals are not associated with a specific thread, their policies are naturally stateless. For this reason, ThreadLock always attempts to match any policy statements for abstract principals. If a called ABI function does not match an abstract policy, ThreadLock enforces the current thread's policy. After thread startup, ThreadLock maintains a "current" state for the policy. In addition, one or more outgoing edges from the "current" state denote candidate policy statements. ThreadLock compares the called ABI function to each of these candidate statements. If the ABI function matches the statement of an outgoing edge, ThreadLock evaluates this statement and advances the "current" state to the destination of the edge. Evaluating a statement performs any operations embedded within the statement. For example, given the statement $\langle function, \tau \rangle$, ThreadLock examines the type signature τ for any operations that either restrict or allow access to memory. If $op\ \tau$ is found within the function's type signature, then ThreadLock performs *op* on the value that corresponds to τ. The operation *op* may be applied to either a function argument or a return value. The recursive definition of τ allows ThreadLock to easily match operations to function arguments. If the operation refers to an argument, ThreadLock performs the operation on the argument before calling the ABI function. For return values, the operation is performed

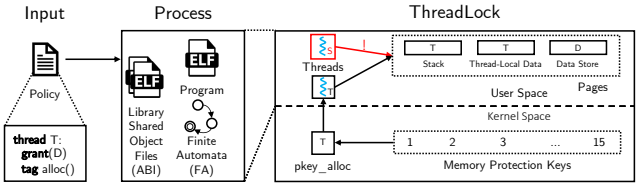

Figure 3: An overview of ThreadLock enforcing a native principal isolation policy over individual threads. Each thread is assigned its own protection domain.

after the ABI function returns. The only operations allowed on function arguments are **tag** and **untag**. When performing an operation, ThreadLock examines τ for an argument that denotes the operation's size (i.e., a built-in symbol called n). For example, **tag** mmap(□,n), causes ThreadLock to tag n bytes returned by mmap.

In some cases, a policy may contain an operation along with a principal ID $\langle op,prid \rangle$. These statements are useful for defining cooperative policies where threads temporarily escalate permissions in between ABI function calls. Such policies are cooperative since developers are expected to limit threads' access to sensitive shared resources. The operations allowed in these statements are **grant** and **revoke**. The **grant** operation provides the thread access to the domain associated with the principal given by *prid* and **revoke** removes access to the domain. These constructs are useful for designating privileged critical sections within a cooperative policy and designating untrusted threads as unprivileged. In this setting, a critical section is a policy segment where a thread escalates their privileges to sensitive memory. In order to simplify authoring policies, the wildcard symbol (□) allows a developer to refer to all threads outside of the policy, or to refer to irrelevant function arguments.

A memory protection mechanism is required to carry out the operations specified in a native principal isolation policy. This memory protection mechanism must also enforce tagging rules by preventing invalid accesses and writes to tagged memory. Once a memory page is tagged to a given principal, the memory protection feature must ensure that the principal maintains proper access to tagged memory. Note that the memory protection feature can be implemented in software, for example as a compiler pass that limits threads' view of memory [26]. Native principal isolation may also use protection features available in hardware in order to implement operations supported by our syntax. Such hardware features must be capable of restricting individual threads' access to memory. MPKs provide a convenient mechanism that allows programs to selectively grant threads read or write access to a principal's data. Figure 3 visualizes ThreadLock protecting an individual program with MPKs by enforcing a native principal isolation policy provided by a developer. In this example policy, the developer has designated a principal T, which represents a thread, to access all memory belonging to the principal D, which represents a data store. Furthermore, the thread T maintains exclusive access to its stack and thread-local data returned by the `alloc` function, which the policy binds to T. If a compromised principal S were to access or write to either T's stack, thread-local data, or D's data store, then MPKs would cause the operating system to terminate the program in response. Hardware implementations of MPKs may provide a limited number protection keys. To address this limitation, prior works have

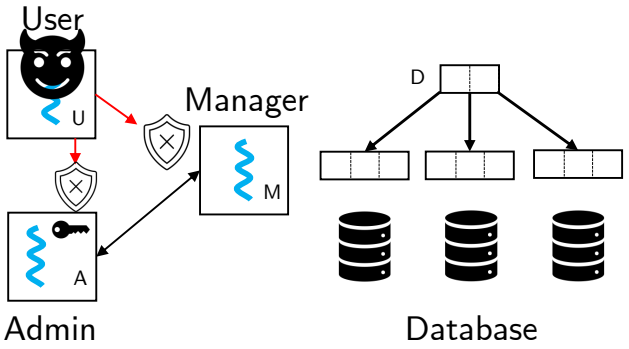

Figure 4: MPKs preventing a compromised user thread from escalating privileges in a database server.

proposed MPK virtualization approaches that can make more domains available to native principal isolation policies (see Section 7).

Native principal isolation policies can define narrow and cooperative scopes that accurately restrict access to sensitive data, which may be either thread-local or global to a process. Enforcing narrow scopes helps avoid the risk of over tagging memory. Over tagging memory to a principal may lead to false positive violations if multiple principals act on data that is mistakenly assigned to a single principal.

3.2 Database Access Control

As a motivating example, consider the database server visualized in Figure 4. In this setting, a pool of worker threads fulfill client queries by interacting with a manager thread that enumerates the B-trees that represent database indexes. Whenever a thread issues a query on behalf of a user, the manager checks the user's token to ensure that the user may perform the operation. ThreadLock reinforces the database's existing access control mechanisms by isolating data between different principals. In this example, a principal is either an individual thread, or the database D. In Figure 4, principals U and A are assigned to the unprivileged user and administrator connection threads, and principal M is assigned to the manager thread. That is, the principal associated with each thread is given in the lower right corner of each thread in Figure 4. Note that every principal is assigned its own protection domain from the chosen memory protection mechanism (see Section 4). Therefore, every thread created by ThreadLock is assigned its own protection domain. Furthermore, each abstract principal, such as a database, is also assigned a single protection domain. For example, Principal D is assigned to the database, and the pages that back the B-tree are bound to D. The policy language allows developers to designate that specific threads share a domain. This is helpful for grouping trusted threads together in a single domain. With trusted threads sharing a domain, more hardware protection keys can be assigned to untrusted threads that handle adversarial inputs. This is helpful for restricting untrusted threads in a MySQL database (see Section 5). Here, the trusted manager principal M is granted full access to D.

When an adversary accesses the server through a victim thread suppose they exploit a vulnerability that allows them to read from another thread's stack. This could allow an adversary to steal an administrator's authentication token. However, ThreadLock isolates

```
1  abstract database:
2    tag mmap(_, n, _)
3    munmap(untag p, n)
4  thread main:
5    revoke(_)
6  thread _:
7    grant(database)
8  thread connection:
9    loop:
10     read(_);
11     grant(database);
12     close(_);
13     revoke(database);
```

Figure 5: A native principal isolation policy that selectively grants threads access to an internal database.

individual thread stacks and pages allocated by principals. While enforcing this policy, the MPK hardware observes that principal U cannot access principal A, and stops any attempt at disclosure. If an adversary attempts to disclose any sensitive information stored by principal M, such as the contents of the database, the MPK hardware triggers the same response. This shows ThreadLock protecting both thread-local and sensitive shared data in a process.

Figure 5 shows the textual representation of this policy using the syntax defined in Figure 2. In this policy, Lines (1-3) designate the database as an "abstract" principal. The purpose of this declaration is to provision a protection key for internal database data-structures and to tag all memory returned by `mmap` with this protection key (Line 2). Note that _ is a textual representation of a "hole" (□) which is helpful for ignoring function arguments that are irrelevant to a policy. In this case, the "hole" allows us to ignore arguments passed to `mmap` since the policy is only concerned with the return value. Should the database unmap pages with `munmap`, ThreadLock binds pages to the default protection domain 0 (Line 3). Policy authors can use the `n` symbol to denote the length for the corresponding `tag` and `untag` operations. Next, the policy simply revokes the `main` thread's access and write privileges to all principals' memory (Lines 4-5). Note that _ allows a policy to grant or revoke access to all principals. This prevents a compromised "main" thread from stealing an authentication token held by another privileged thread. We can also use _ to grant privileges to threads that are not named elsewhere in the policy (Line 6). This allows us to effectively declare a set of trusted threads as a principal and share memory amongst them. This is necessary in programs where the number of running trusted threads exceeds the number of principals supported by a memory protection mechanism. In this policy, we grant all threads outside the policy access to the database (Line 7), and restrict untrusted connection threads' access (Lines 8-13). Note that loops are inherently stateful, so ";" encodes a sequence of function calls that must occur.

Since individual connection threads process queries in an infinite loop, we use the `loop` construct (Line 9) to specify that ThreadLock can expect to see two symbols alternate while handling connections (`read` and `close`) (Lines 10 and 12). These symbols designate the start and end of a connection, and we use them to determine when to grant and revoke access to the database, respectively (Lines 11 and 13). Any attempt to access or modify the database outside of this critical section or access another principal's data will result in an exception that kills the process before any data can be disclosed. Note the critical section is defined by the policy between granting and revoking access (Lines 11 and 13). A more in-depth evaluation of a related policy for the MySQL RDBMS is given in Section 5.

4 IMPLEMENTATION

In this section, we describe our prototype implementation of ThreadLock which consists of 677 lines of C/C++ code. First, we provide an overview of how our prototype may be deployed. We show how ThreadLock transforms a native principal isolation policy into an FA that, during program execution, enforces a policy. Prior to enforcement, ThreadLock must correctly configure the `PKRU` to restrict new threads' access to memory according to the policy. Finally, we describe how ThreadLock selectively isolates program memory using recently introduced system calls and instructions.

4.1 ThreadLock Deployment

The current prototype implementation of ThreadLock generates a shared object file that wraps individual functions called by the protected binary in order to isolate individual principals. This requires intercepting calls to functions that spawn threads, and to library functions specified by the policy. We use the `LD_PRELOAD` environment variable to intercept functions. We emphasize that in a real deployment of ThreadLock, an operator could avoid using `LD_PRELOAD` by embedding the shared object file emitted by ThreadLock into a system folder to be loaded by the linker. An alternative implementation of ThreadLock could enforce policies by inserting instrumentation through a compiler pass [32] or via binary rewriting [50].

4.2 Policy Synthesis and Enforcement

Given a native principal isolation policy that outlines how to protect a workload's principals, ThreadLock must synthesize C code to enforce the policy. To do so, ThreadLock simply transforms the policy provided by the developer into a series of stand alone C functions that intercept ABI functions given in the policy's statements. Recall the policy language in Section 3 permits a developer to isolate data across different threads identified by their `start_routine` function. A hook in the generated code around `pthread_create` ensures that a protection key is provisioned for each thread or shared amongst a set of threads. This also allows ThreadLock to apply portions of the policy relevant to thread initialization (see Section 4.3). If `start_routine` refers to a function given in the policy, the hook creates an FA to enforce the policy and associates the FA with the newly created thread. Additional details on policy enforcement may be found in the Appendix (see Section A.1).

4.3 Isolating Threads

The widely used POSIX threads (`pthreads`) library provides programs the ability to spawn individual threads that execute code concurrently. In the program generated by the synthesis stage, ThreadLock defines a stub that intercepts every call to `pthread_create`. Intercepting the call to `pthread_create` is necessary to enforce a policy within a new thread using a trampoline visualized in Figure 6. Upon entering the `pthread_create` stub, ThreadLock first allocates a small buffer that stores the `attr`, `start_routine`, and `arg` pointers that the program passed to the stub. Next, ThreadLock calls `pthread_create`, but provides an internal trampoline function

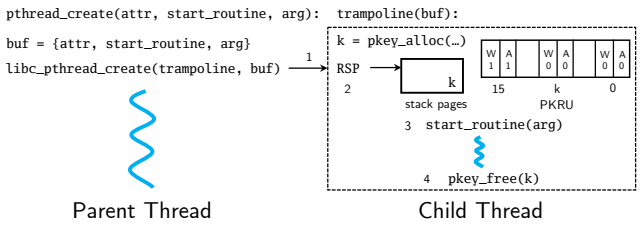

Figure 6: A trampoline that allows ThreadLock to isolate a thread using a protection key obtained from the operating system.

called `trampoline` as the new thread's `start_routine` function (Step 1).

ThreadLock passes the buffer containing the original `attr`,`start_routine`, and `arg` pointers as the argument to the trampoline. After the new thread begins, the trampoline obtains the three pointers given in the buffer, and then frees the buffer. The trampoline must now restrict the new thread using an MPK provisioned for the thread (Step 2). This implies that each thread is assigned its own distinct protection key. Note that a policy may cause multiple trusted threads to share the same protection key. This is beneficial when the number of threads outnumber the protection keys available in hardware. In Section 7 we present alternative solutions to the limited hardware protection keys.

The trampoline allocates a protection key k from the operating system using the `pkey_alloc` library function and system call. These are available in recent releases of the Linux kernel and the GNU C Library (`glibc`). ThreadLock binds the thread's stack pages to protection key k using the `pkey_mprotect` system call. This system call allows ThreadLock to associate all of the thread's stack pages to protection key k. After calling `pkey_mprotect` on the stack, any thread that is forbidden from accessing protection key k's pages will then crash when attempting to read from or write to the new thread's stack. ThreadLock stores the protection key k in thread-local memory using `pthread_setspecific`. The dual of this function, `pthread_getspecific`, allows ThreadLock to obtain protection key k later on while enforcing policies.

Next, ThreadLock initializes the PKRU register to permit access only to memory bound to the new thread's protection domain and the default domain 0. Recalling the semantics of the PKRU given in Section 2, we configure the PKRU in the trampoline using the following procedure. First, define a bitmask `mask = (0x3 << (2 * key) | 0x3)`. This bitmask contains values of 1 in the access and write positions for protection keys k and 0 within the PKRU. We can disallow both access and writes to other domains by computing the bitwise negation of this mask (`~mask`) and storing the negated mask in the PKRU register by using the `wrpkru` instruction. Recent Linux kernels also save and restore the value of the PKRU during context switching. Once ThreadLock sets the PKRU register for a thread it will be preserved unless altered by the policy.

Once the memory protection key restrictions are in place for the new thread, the trampoline must obtain the function that the `start_routine` parameter refers to in the program binary. This is necessary in order to determine which part of the program's policy applies to the newly created thread. In general, this cannot be determined statically since programs may assign the location of dynamically generated code to `start_routine`. In this work, we recover the function given by `start_routine` by consulting a binary's `.text` section (see Section A). Once the function is obtained, the trampoline checks whether a policy exists for the function. If a policy exists, the trampoline initializes an FA to track the execution of the relevant policy, if necessary, and store the FA in thread-local memory with `pthread_setspecific`.

Following this initialization procedure, the now restricted trampoline calls the original `start_routine` function with the original `arg` pointer as input (Step 3). Thereafter, any attempt to access memory outside of protection domain k or 0 will yield a segmentation violation and the operating system will, by default, terminate the process in response. Once the `start_routine` function completes, the trampoline deallocates the protection key k by calling the `pkey_free` function (Step 4). This is necessary to ensure that future threads can utilize protection key k after a thread exits. Additional details can be found in the Appendix (see Section A.1).

4.4 Policy Operations

While ThreadLock automatically hardens individual threads' stack pages, pages allocated by threads may benefit from protections offered by MPKs. Since threads may share access to memory pages, protecting these shared pages without incurring false positives is accomplished with cooperative policies. In general, ThreadLock cannot know a-priori whether a given page allocated for heap data will be accessible solely by a thread. For example, a thread may use a page, and then recycle the page to be used by another thread. Instead of re-designing existing memory allocators to accommodate MPKs' page level granularity, our policy language allows a developer to choose when a thread should access sensitive data through cooperative policy scopes. This assumes that sensitive pages are allocated by ABI functions imported by the protected program, and thus can be intercepted by ThreadLock. Cooperative scopes require implementing the operations presented in Section 3. During thread execution, the generated function stubs handle transitioning between individual policy states using an FA. Upon encountering a "tag" or "untag" operation in a policy, the FA needs a way to associate or dissociate a given page with the thread's protection key k. On recent Linux kernels, both of these operations can be accomplished by calling a variant of the `mprotect` system call, `pkey_mprotect`. Given a pointer to the memory pages to "tag", the FA will invoke `pkey_mprotect` on the memory chunk using protection key k. To "untag", the FA invokes `pkey_mprotect` using protection key 0, which effectively returns the pages to all threads. The number of bytes for each operation is taken from the function parameter labeled with the symbol `n` in the policy statement. For example, the statement `tag alloc(n)` causes `pkey_mprotect` to be called on `n` bytes starting from the pointer returned by `alloc`. Details on granting and revoking permissions within cooperative scopes can be found in the Appendix (see Section A).

5 EVALUATION

In this section, we seek to answer the following research questions.

RQ1: Can ThreadLock protect real world applications without compromising performance?

Program	Version	No. of Network Threads	Policy Description	Attack Vector	CVE
Apache	2.4.41	15	Isolate dynamically allocated thread memory	Memory Disclosure	CVE-2014-0160
Redis	7.0.4	5	Grant worker threads access to data store	Memory Corruption	CVE-2020-14147
MySQL	8.0.31	13	Isolate connection threads from internal memory	Privilege Escalation	CVE-2022-21617

Table 1: ThreadLock policies that protect against real world attack vectors on popular programs and the number of initial threads that handle network inputs.

RQ2: Can enforcing native principal isolation policies with ThreadLock provide security benefits to real world applications?

Answering both **RQ1** and **RQ2** are important to prevent ThreadLock from interfering with protected applications, and to ensure ThreadLock can mitigate real world attack vectors. To answer these questions, we define ThreadLock policies that restrict access to memory held by principals in the widely used Apache web server, Redis in-memory data store, and MySQL RDBMS. Section 5.1 describes the configuration used for our experiments. Section 5.2 describes our performance evaluation for each application. Overall, we show that ThreadLock's performance overhead is minimal for each workload included in our evaluation. Finally, Section 5.3 shows the security benefits of ThreadLock by mitigating exploits against vulnerabilities documented by CVEs in our evaluation artifacts.

5.1 Experimental Set Up

We evaluated ThreadLock on an Ubuntu 20.04LTS server with 8 Intel Xeon Platinum 8275CL CPUs and 32GB of RAM. Table 1 summarizes the ThreadLock policies used in our evaluation. We argue that these policies show ThreadLock can protect real-world multi-threaded workloads across different domains, and enforce diverse policies, including isolating individual threads' memory and selectively granting individual threads access to global data. For each application, we evaluated ThreadLock using the production binary files available on Ubuntu 20.04LTS.

We configured each application in our evaluation to limit the number of threads that handle network inputs. This is done to isolate threads that handle untrusted data using the limited protection keys available to each process. Reducing the number of threads may limit the number of threads available for handling requests. However, an operator can increase the number of threads available by increasing the number of application processes (e.g., increase the number of child processes started by Apache or spin up more Redis servers) but cap the number of threads in each process. Since process startup is slower than thread startup, this may incur some performance penalty. However, provisioning the necessary number of processes beforehand ensures threads can be managed within each process. In some cases, as with Redis and MySQL, the initial number of worker threads is less than the number of available protection keys. This is necessary to ensure background threads may be protected, and that the server may start up new worker threads during times of high load. In the following, we describe our ThreadLock policy for MySQL, which is similar to the example policy presented in Section 3.2.

MySQL Policy. In contrast to the other applications included in our evaluation, MySQL's default configuration spins up more than thirty threads on start up to manage the operation of the InnoDB database engine and to monitor the database's normal operation. This implies that a blanket ThreadLock policy that assigns a protection key to each thread will quickly exhaust all the protection keys available before isolating the connection threads that interact with remote adversaries. In order to protect these connection threads, we define a policy that assigns a protection key to each connection thread. The policy then provisions a shared internal protection key for all other "internal" threads. Each connection thread only has access to its stack and protection domain 0. Each "internal" thread has access to all internal and connection threads' stack memory. We found that it is necessary to provide "internal" threads access to connections' memory while querying information from the database. That is, denying "internal" threads' access to connection thread stacks quickly crashed the MySQL server with false positives while querying database tables with the `SELECT` command. Section 5.3 presents a concrete privilege escalation attack vector thwarted by this policy. Further details can be found in the Appendix (see Section A.3).

	Mean Time Per Request (ms)			
Clients	Apache		Apache (TL)	
32	315.32	±1.35	315.01	±1.44
64	633.45	±3.81	634.30	±2.77
128	1,267.22	±5.21	1,266.71	±4.49
256	2,535.34	±13.12	2,553.75	±28.67
512	5,054.10	±18.06	5,075.09	±26.62

Table 2: Mean and standard deviation of the mean time per request for clients issuing 10k requests to a baseline front-end Apache server and a server protected by ThreadLock (TL).

5.2 Performance Benchmarks

In this section, we evaluate the performance overhead of ThreadLock by stress testing our evaluation artifacts using standard benchmarks.

Apache. In order to measure the performance impact incurred by ThreadLock on Apache, we ran the Apache Benchmark (`ab`) test suite on both an unprotected baseline Apache server, and a server protected by ThreadLock. In this experiment, we set up an Apache web server to act as a front end server for a Wordpress 6 installation (the latest version). Next, we measured the server's response time as we increased the number of concurrent clients visiting the front page of the Wordpress site a total of 10,000 times. We repeat this experiment 10 times for each number of clients. Table 2 summarizes the average mean time per request and standard deviation in milliseconds (ms) for each configuration. Overall, applying ThreadLock to the server did not degrade the server's performance. In fact, the server's

performance is often comparable to the baseline. Furthermore, the overhead imposed by ThreadLock did not increase even as the number of clients communicating with the server increased and placed greater pressure on individual threads. In the worst case, we observed 1.06% overhead on the Apache server's mean time per request.

Redis. In this section, we measure the performance overhead while pushing a Redis instance protected by ThreadLock to its limits. To do so, we stress a Redis server with the `redisbench` utility which allows us to apply a configurable amount of load to the Redis server. The load placed on the server is given by the product of the total number of clients started by `redisbench`, a constant that represents the object size created by each client, and the total number of objects each client creates. In this experiment, we found that choosing a combination of these parameters to generate a 27GB data-store maximized the load on the server without exceeding the machine's memory. Increasing the total size of the data store beyond this threshold quickly caused the out of memory killer to terminate the Redis server, both in the baseline and ThreadLock configuration. In order to keep consistent load on the server, we decreased the amount of objects created by each client as the number of clients increased.

Table 3 compares this benchmark's mean runtime and standard deviation on both a baseline Redis server and one protected by ThreadLock. In this experiment, we ran the benchmark 10 times on each server. These results show that ThreadLock can protect a multi-threaded Redis server with negligible overhead. In this case, we observed 0.04% performance overhead in the benchmark's runtime on average while protecting Redis with ThreadLock.

	Mean Time Per Benchmark (s)	
Clients	Redis	Redis (TL))
64	16.39 ±0.06	16.41 ±0.11
128	25.87 ±0.17	26.02 ±0.12
256	41.10 ±0.98	41.14 ±0.94
512	73.78 ±0.47	73.53 ±0.68
1,024	118.29 ±0.98	118.26 ±0.68

Table 3: Mean and standard deviation of the `redisbench` benchmark's runtime while storing the maximum amount of data into a baseline Redis server and a server protected by ThreadLock (TL).

MySQL. To profile the performance overhead of ThreadLock on MySQL, we utilize the `sysbench` utility to create large amounts of records in a database. Next, we simulate realistic load on the server by having the maximum number of clients query data from the tables. In this experiment, we use Lua scripts provided by `sysbench` to create millions of records in the database, and then utilize separate scripts to repeatedly issue queries to the database. We argue that this shows ThreadLock supports the normal operation of the database. This includes creating the database, adding new records, retrieving existing records, and deleting the database.

Table 4 summarizes the outcome of this experiment. Overall, ThreadLock imposed negligible performance overhead, and allowed the server to perform its operations as normal. Furthermore, the server often behaves with runtime comparable to the baseline server.

	Mean Time Per Benchmark (s)	
Operation	MySQL	MySQL (TL))
Bulk Insert	10.15 ±0.03	10.13 ±0.03
Select Range	10.01 ±0.01	10.01 ±0.01

Table 4: Mean and standard deviation of the `sysbench` benchmark's runtime while inserting and selecting 1M rows in a MySQL database protected by ThreadLock (TL).

Since MySQL threads all have their `PKRU` values assigned during thread creation and do not change for the lifetime of the server, traffic submitted to the database avoids the expensive operation of writing to the `PKRU` register. This explains the low performance overhead of using ThreadLock while stress testing MySQL.

5.3 Attack Scenarios

In this section, we evaluate the security benefits of ThreadLock by describing attack scenarios mitigated in our evaluation artifacts.

Heartbleed Disclosure in Apache. To evaluate ThreadLock's ability to protect sensitive data within an Apache server, we reproduced a proof of concept (PoC) exploit against the Heartbleed vulnerability (CVE-2014-0160) [6]. The Heartbleed vulnerability allowed malicious clients to disclose up to 64kb of heap memory by tricking a web server using OpenSSL (1.0.1-1.0.1f) into performing a buffer over-read while handling heartbeat requests. Before running the exploit, we confirmed that threads' protected pages held sensitive data. We confirmed this by deterministically scanning all protected pages using our custom PINtool. Running the HeartBleed exploit against an unprotected Apache web server provides a baseline of what sensitive data an adversary can disclose using a heap buffer over-read primitive. If a Heartbleed style attack is unable to recover information protected by ThreadLock, that provides a small signal for ThreadLock's efficacy.

ThreadLock does not prevent adversaries from exploiting Heartbleed, since the compromised heartbeat thread can still recover data present in buffers unprotected by ThreadLock. In our setting, we were concerned whether an adversary can find any sensitive data in unprotected buffers. This would easily evade ThreadLock. After running a Heartbleed attack for over 12 hours, while simultaneously stress testing the server with traffic generated by the Apache Bench (`ab`) tool, we observed an adversary could only recover a limited amount of request data, such as HTTP verb, requested URL, and a fixed amount of meta-data. Contrast this partial request information to the complete requests, responses, and rendered application content held in Apache worker threads' stack and thread-local memory observed by our PINTool. These results indicate that the `glibc` heap does not contain Apache threads' sensitive data. Otherwise, we would expect a varying amount of sensitive data to appear after repeatedly disclosing 64kb of data from multiple locations on the heap. Therefore, an adversary would have to obtain sensitive data by accessing pages protected by ThreadLock. This includes thread stacks or thread-local memory, which are protected by MPKs. This demonstrates the utility of ThreadLock, since any attempt to disclose these protected pages from a heartbeat thread will fail. Additional mitigated heap-based attack vectors are described in the Appendix (see Section A.2).

Poisoning the Redis Data Store. An adversary that corrupts Redis' internal data-structures can mislead clients that rely on the data store. This can have serious consequences when Redis is used as a message broker for transmitting sensitive messages between services. For Redis configurations that rely on access control lists (ACLs) to prevent unprivileged users from writing to the data-store, these corruption attacks allow adversaries to circumvent existing security mechanisms provided by Redis. For example, a vulnerability documented by CVE-2020-14147 [8] and CVE-2015-8080 [7] permit an adversary to perform a stack-based buffer overflow by taking advantage of a type confusion error within the Lua interpreter embedded in Redis. This allows an adversary to write a value of their choice to any address located above a victim stack frame by simply evaluating a Lua expression via the `EVAL` command [1]. During our evaluation, we confirmed that the vulnerability in `EVAL` allows an adversary to reach data store pages from worker threads via a stack based buffer overflow, and that MPKs configured by a native principal isolation policy prevent a successful corruption. More details are given in the Appendix (see Section A).

Privilege Escalation in MySQL. A MySQL database often accommodates multiple clients with a variety of privileges. After a client logs in, the connection thread started for the client receives a pointer to a security context that stores the client's authentication token. When the client issues a query from the connection thread, the thread's security context determines the client's visibility into the database. For example, if an unprivileged client logs into the database, their security context will be unable to delete records stored in a table or query sensitive information stored in administrative tables. However, if an adversary can peak into memory held by another thread, they could steal a logged in administrator's security context in order to escalate privileges. We show how ThreadLock can prevent compromised threads from stealing account credentials by isolating access to thread stacks and internal memory. For example, consider the issue present in CVE-2022-21617 [9], where a vulnerability in the server's connection manager allows an attacker with low privileges the ability to read the contents of server memory. Successfully exploiting the connection manager allows the adversary to read memory from the "main" thread of execution. ThreadLock prevents this attack by configuring the `PKRU` register for "main" to permit access only to protection domain 0 (i.e., the default domain). Instead of implementing an exploit against the vulnerable connection handler, we simulate a successful exploit in line with prior security evaluations [39] by adding malicious behavior to the connection handler. Once our "compromised" connection handler reads from an active connection thread's memory, ThreadLock terminates the server process before an adversary can steal user credentials.

6 RELATED WORK

ThreadLock relates to prior work in three main categories, memory defenses, policy based defenses, and intra-process isolation.

6.1 Memory Defenses

MPKs provide programs more fine-grained control over the traditional permissions enforced by the memory management unit (MMU) on modern CPUs. Without the restrictions imposed by MPKs, every thread running in a process has the same view of memory provided by the MMU. That is, once a program changes a page's permissions to some combination of readable, writable, or executable, then the MMU enforces those permissions whenever any thread accesses, writes, or executes that page. This enables protections like $W \oplus X$ where a page is either writable, executable, but never both. This prevents adversaries from spraying instructions into executable code regions and helps neutralize stack based shellcodes and JIT spraying. While MPKs are a recent addition to Intel CPUs, MPKs have existed in other architectures such as the "storage protect keys" available on IBM AIX systems [11] or the planned memory tagging extension (MTE) for ARM CPUs [2]. Furthermore, research platforms like CHERI explore the feasibility of finer-grained memory access control at the cost of potentially doubling pointer sizes [14, 36].

An analogy can be drawn between restricting access to memory with protection keys to hiding memory using the address space layout randomization (ASLR) defense found in every commodity operating system [45]. ASLR maps program segments into random locations at runtime. Instead of isolating program components with hardware features, ASLR conceals the location of code segments from a curious adversary. Though randomizing the location of pages holding executable code and allocated memory makes it difficult for an adversary to exploit a process, historically ASLR implementations have encountered practical difficulties [43]. Examples include allowing brute force attacks from remote adversaries [15] and cache based attacks [23]. These limitations have spawned numerous improvements over the years in an attempt to improve the granularity of randomization or reduce the amount of information gained from a disclosure [22, 34, 48, 49]. As prior work has suggested [30], memory protection keys may have the potential to offer an alternative to randomized defenses like ASLR.

6.2 Policy Based Defenses

Previous tagging schemes require embedding policies within a program to describe how to protect sensitive user data. For example, tagging user data held within a web server protected by HiStar requires manually applying a policy to the web server's implementation [52]. Transforming commodity operating system kernels into a nested design requires designing policies for each modified kernel. In this setting, a developer must decide when to transfer control over to a trusted computing base (TCB) to perform critical operations [18]. When using a library OS, application authors also need to specify which components should be isolated from their workload with assistance provided by the library OS' build system [33]. While secure computing enclaves like Intel Software Guard Extensions (SGX) silo sensitive components from the rest of the system, implementation bugs in the code that interacts with secure enclaves like SGX can have dire security consequences, especially in distributed settings. Prior work has addressed this issue by defining language based approaches for reasoning over programs' interactions with an enclave. Similar to ThreadLock, these approaches require a developer to provide a policy in order to realize the benefits provided by a security feature.

6.3 Intra-Process Isolation

Secure memory views use compiler instrumentation to establish a hierarchical view of a process address space to restrict individual thread's access to memory regions [26]. While secure memory views

may be more flexible than the limited protection keys available in recent hardware, their use requires some instrumentation of the program. While the limited real estate available in page table entries leads to only 16 usable MPK domains, prior work has shown MPKs can be virtualized with the assistance of a custom kernel module [38]. Access to MPKs can be further virtualized by running individual thread pools within virtual machines which guarantees each thread located in a pool maintains its own protection key [24].

Instead of isolating access to memory held by individual threads, prior works have attempted to implement general purpose intra-process sandboxes using MPKs [25, 46]. While innovative, these approaches have been shown to be susceptible to security issues that allow adversaries to break the MPK sandbox abstraction [17]. An Achilles' heel of MPK based sandboxes is the relative ease with which an adversary can break out of the sandbox by issuing a `wrpkru` or `xrstor` instruction or by issuing system calls to interfere with other sandboxes. More recent work has addressed the former limitation by embedding instruction monitors within the hardware to prevent adversaries from breaking out of the sandbox [19]. To monitor the execution of sensitive system calls, custom binary loaders and `ptrace` monitoring have been proposed [47]. Novel system call attacks against MPK sandboxes have been discovered, including using the relatively benign `madvise` system call to clobber MPK permissions. These attacks can be mitigated by using nested-filtering [41] implemented on top of the Donky framework [42].

MPKs have also found applications outside of implementing intra-process sandboxes. For example, developers have sought to use static analysis to guide the introduction of MPKs into security sensitive sections of widely used code bases like cryptographic APIs [28]. Prior work also explicitly restructures allocators in language runtimes to protect allocated chunks while accommodating MPKs' page level granularity [29]. Furthermore, MPKs can be used for restricting access to sensitive pointers in a process, which is helpful for implementing CFI schemes [27]. While embedded micro-controllers usually lack an MMU, recent ARM micro-controllers feature a memory protection unit (MPU) that has been used to isolate control-flow information from firmware data within a real time operating system [20]. Prior work has also proposed policy languages used to restrict untrusted libraries' access within an application [21]. Static analysis and sanitizers have also been proposed to detect and mitigate the consequences of using unsafe features in memory safe programming languages such as Rust [40]. Other policy language techniques prove security properties of applications that rely on SGX enclaves, where the incorrect use of the enclave API can lead to serious security problems [31].

7 DISCUSSION

In this section, we discuss different domain virtualization approaches and the limitations of ThreadLock.

7.1 Domain Virtualization

In order to address the limited number of protection keys available in Intel's MPKs, ThreadLock could take advantage of recent work that makes MPKs a virtual resource [38]. ThreadLock could assign a given thread a random protection key k on thread start-up. In these settings, more than 15 domains could be utilized by ThreadLock. Furthermore, an adversary could not reliably predict the protection key assigned to each thread. However, if a compromised thread were to discover the set of threads that share the compromised thread's protection key, then an adversary may be able to obtain or corrupt sensitive data held by those threads. This could be improved by providing each thread its own address space [51]. This address isolation based approach ensures each thread has its own protection domain (e.g., address space), but may require significant changes to protected applications.

7.2 ThreadLock Limitations

Though native principal isolation can conveniently isolate memory regions using a program's ABI, it is primarily applicable to programs that rely on external libraries. For this reason, the current ThreadLock prototype cannot enforce native principal isolation policies against statically linked programs, which contain an application and all of its dependencies in a single executable file (e.g., small container images with programs statically linked against the "musl" C library). This could be addressed by enforcing policies via compiler passes or binary rewriting. Furthermore, if a program's ABI functions neither return nor receive as input pointers to sensitive memory, it may be difficult to define a useful policy. Furthermore, domain expertise about an application and its libraries may be required to design effective policies. However, we found that tracing and examining source code of real world programs was sufficient to derive useful native principal isolation policies. Finally, if an adversary bypasses control-flow integrity (CFI), they can escalate `PKRU` privileges either via `wrpkru` or `xrstor`.

8 CONCLUSION

In this work, we presented native principal isolation, a technique for specifying intra-process security policies enforced by a memory protection mechanism. We introduced ThreadLock, an implementation of native principal isolation that protects multi-threaded applications using memory protection keys (MPKs). We applied ThreadLock to the widely used Apache web server, Redis in-memory data store, and MySQL RDBMS. We demonstrated that MPKs protect sensitive data held by each of these workloads from real world attacks while producing 1.06% performance overhead in the worst case. We argue that these results show that ThreadLock permits integrating MPKs into production workloads without requiring any modification to executable binaries or application source code.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CNS-1916393 and CNS-1916398.

REFERENCES

[1] 2015. Integer overflow (leading to stack-based buffer overflow) in embedded lua_struct.c. https://github.com/redis/redis/issues/2855. Acc. 2022-10-06.
[2] 2019. Memory Tagging Extension: Enhancing memory safety through architecture. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety. Acc. 2022-02-27.
[3] 2020. A Technical Look at Intel's Control-flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html. Acc. 2022-08-28.
[4] 2022. Binary Compatibility. https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Compatibility.html. Acc. 2022-09-21.
[5] 2022. Close, but No Cigar: On the Effectiveness of Intel's CET Against Code Reuse Attacks. https://grsecurity.net/effectiveness_of_intel_cet_against_code_reuse_attacks. Acc. 2022-08-24.
[6] 2022. CVE-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160. Acc. 2022-05-29.

[7] 2022. CVE-2015-8080. https://nvd.nist.gov/vuln/detail/CVE-2015-8080. Acc. 2022-10-05.

[8] 2022. CVE-2020-14147. https://nvd.nist.gov/vuln/detail/CVE-2020-14147. Acc. 2022-05-29.

[9] 2022. CVE-2022-21617. https://www.cve.org/CVERecord?id=CVE-2022-21617. Acc. 2022-11-22.

[10] 2022. How to Survive the Hardware-assisted Control-flow Integrity Enforcement. https://paper.bobylive.com/Meeting_Papers/BlackHat/Asia-2019/bh-asia-Sun-How-to-Survive-the-Hardware-Assisted-Control-Flow-Integrity-Enforcement.pdf. Acc. 2022-08-24.

[11] 2022. Storage Protect Keys. https://www.ibm.com/docs/en/aix/7.2?topic=concepts-storage-protect-keys. Acc. 2021-08-16.

[12] 2023. Usage statistics of Apache. https://w3techs.com/technologies/details/ws-apache. Acc. 2023-05-02.

[13] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and Systems Security* (2009).

[14] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert NM Watson, and Peter Sewell. 2022. Verified security for the Morello capability-enhanced prototype Arm architecture. In *European Symposium on Programming*.

[15] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. 2014. Hacking blind. In *IEEE Symposium on Security and Privacy*.

[16] William Blair, William Robertson, and Manuel Egele. 2022. MPKAlloc: Efficient Heap Meta-Data Integrity Through Hardware Memory Protection Keys. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

[17] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *USENIX Security Symposium*.

[18] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *ACM Conference on Architectural Support for Progamming Languages and Operating Systems*.

[19] Leila Delshadtehrani, Sadullah Canakci, William Blair, Manuel Egele, and Ajay Joshi. 2021. FlexFilt: Towards Flexible Instruction Filtering for Security. In *Annual Computer Security Applications Conference*.

[20] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J. Walls, and John Criswell. 2022. Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage. In *USENIX Security Symposium*.

[21] Adrien Ghosn, Marios Kogias, Mathias Payer, James R Larus, and Edouard Bugnion. 2021. Enclosure: language-based restriction of untrusted libraries. In *ACM Conference on Architectural Support for Progamming Languages and Operating Systems*.

[22] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*.

[23] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *Network and Distributed System Security Symposium*.

[24] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and Efficient Memory Protection Keys. In *USENIX Annual Technical Conference*.

[25] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-process isolation for high-throughput data plane libraries. In *USENIX Security Symposium*.

[26] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing least privilege memory views for multithreaded applications. In *ACM Conference on Computer and Communications Security*.

[27] Mohannad Ismail, Jinwoo Yom, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. 2021. VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks. In *ACM Conference on Computer and Communications Security*.

[28] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Dawu Gu, Hang Zhang, Siqi Ma, Zhiyun Qian, and Juanru Li. 2022. Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK. In *IEEE Symposium on Security and Privacy*.

[29] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-safe: automatically locking down the heap between safe and unsafe languages. In *European Conference on Computer systems*.

[30] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No need to hide: Protecting safe regions on commodity hardware. In *European Conference on Computer systems*.

[31] Shivendra Kushwah, Ankush Desai, Pramod Subramanyan, and Sanjit A Seshia. 2021. PSec: Programming Secure Distributed Systems using Enclaves. In *ACM ASIA Conference on Computer and Communications Security*.

[32] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*.

[33] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: towards flexible OS isolation. In *ACM Conference on Architectural Support for Progamming Languages and Operating Systems*.

[34] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Network and Distributed System Security Symposium*.

[35] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel Software Guard Extensions Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*.

[36] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M Norton, Simon W Moore, Peter G Neumann, et al. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *IEEE Symposium on Security and Privacy*.

[37] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2018).

[38] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX Annual Technical Conference*.

[39] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. Nojitsu: Locking down javascript engines. In *Network and Distributed System Security Symposium*.

[40] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping safe Rust safe with Galeed. In *Annual Computer Security Applications Conference*.

[41] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *USENIX Security Symposium*.

[42] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86. In *USENIX Security Symposium*.

[43] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security*.

[44] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *International Workshop on Hardware and Architectural Support for Security and Privacy*.

[45] Team Pax. [n. d.]. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt. Acc. 2022-10-11.

[46] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium*.

[47] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In *European Conference on Computer systems*.

[48] Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi. 2021. Making Information Hiding Effective Again. *IEEE Transactions on Dependable and Secure Computing* (2021).

[49] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and deployable continuous code re-randomization. In *USENIX Symposium on Operating Systems Design and Implementation*.

[50] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-agnostic binary recompilation. In *ACM Conference on Architectural Support for Progamming Languages and Operating Systems*.

[51] Ziqi Yuan, Siyu Hong, Rui Chang, Yajin Zhou, Wenbo Shen, and Kui Ren. 2023. VDom: Fast and Unlimited Virtual Domains on Multiple Architectures. In *ACM Conference on Architectural Support for Progamming Languages and Operating Systems*.

[52] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2008. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *USENIX Symposium on Operating Systems Design and Implementation*.

A APPENDIX

In this section, we provide supplementary material for our paper.

A.1 ThreadLock Implementation Details

Policy Synthesis and Monitoring. Currently, policies attached to `abstract` principals are inherently stateless since ThreadLock does not track the execution of `abstract` principals. In the event that a thread's policy is stateless, the generated stubs avoid creating and storing an FA. In these settings, ThreadLock only needs to fetch the thread's protection key k from thread-local memory (see Section 4.3).

Signal Handlers. By default, the current implementation always includes stubs for hardening individual threads' stacks and modifying signal handlers. The former ensures that each thread maintains exclusive access to its own stack. The latter allows signal handlers to operate on memory associated with any principal. This is necessary because a signal handler may need to access memory held by a thread, but ThreadLock cannot predict which thread's data may need to be accessed for every signal. For this reason, the signal handling stub generated by ThreadLock always zeroes the contents of the PKRU register before handling a signal. Recall from Section 2 that applying zeros to every slot in the PKRU grants a thread read and write access to all protection key domains. This prevents signal handling from incurring any unnecessary false positives. This general solution can be refined by enforcing policies tailored to signal handling threads. This allows ThreadLock to limit a potentially compromised signal handler's access to application data.

Thread Stacks. The ThreadLock trampoline restricts a newly created thread by doing the following. First, the trampoline fetches the current stack pointer by declaring a local variable equal to `asm("rsp")`. This allows the trampoline to orient itself on the thread's stack. By default, the trampoline will find the page that RSP refers to and then enumerate pages downward in memory in order to find the bottom of the stack. Since the default size of `pthread` stacks is known, ThreadLock can just use the default size to find the bottom of the stack. However, if an application were to change the stack size through the `attr` pointer, the trampoline can still obtain the size of the stack by inspecting the original `attr` pointer which the trampoline receives as input.

Function Recognition. To determine which function the `start_routine` refers to the trampoline does the following. First, find the location of the `.text` section in virtual memory by consulting `/proc/<pid>/maps` where `<pid>` is the process identifier (PID) of the current process. The function referred to by `start_routine` in the program binary can then be obtained by `start_routine` − $\texttt{.text}_{vm}$ + $\texttt{.text}_{offset}$ where $\texttt{.text}_{vm}$ is the address of the `.text` section in virtual memory and $\texttt{.text}_{offset}$ is the location of the `.text` section in the program binary. Policies refer to individual functions via symbols or by direct addresses. When symbols are provided, the function's address is obtained by inspecting the program's symbol table. The addresses of individual functions of interest can be obtained either by static or dynamic binary analysis. However, knowing a function's address a priori is not always feasible (e.g., just in time (JIT) compiled functions).

Restricting Main Thread. Note that the trampoline is not necessary when isolating the main thread of execution. ThreadLock initializes the `main` thread by intercepting the `__libc_start_main` function. During policy enforcement, ThreadLock detects the main thread by checking that `getpid()` equals `gettid()`.

Isolating Threads. After a new thread is initialized, the trampoline may alter additional permissions based on the policy. For example, a policy may consider a thread either privileged or unprivileged. In these cases, the initial "grant" or "revoke" operations given in the policy are performed by the trampoline.

Memory Access Control. Note that the "grant" and "revoke" operations can be implemented by permitting or denying access and writes to the relevant principal in the current thread's PKRU. For example, if a thread calls a function that advances its FA to a `grant(database)` statement, then the FA will update the PKRU to PKRU & `~(0x3 « (2 * database_pkey))` where `database_pkey` is the protection key provisioned for the `database` principal. Likewise, a `revoke(database)` statement will update the PKRU to PKRU | `(0x3 « (2 * database_pkey))`. A cooperative policy grants a thread access to shared memory pages when needed, and otherwise revokes access.

A.2 Attacking Apache from the Heap

On modern systems, guard pages are placed around the pages holding Apache workers' thread-local data. Hypothetically, if an adversary found other ways to reach sensitive data aside from a buffer overread, MPKs would prevent the data's disclosure. As an example, we consistently observed protected pages located close to and at higher addresses than heap arenas in memory. This implies that a small write or read primitive from a heap pointer could easily reach threads' sensitive data. We observed the same behavior while running Apache built to run without guard pages. In this configuration, ThreadLock would prevent Heartbleed from disclosing protected pages should the top of the heap be located at an address immediately beneath the victim page in memory.

A.3 MySQL Policy Details

While "internal" threads manage the normal operation of the database, the "main" thread of execution accepts incoming connections and spins up a thread for each individual client. Since this is the adversary's first point of contact with the server, it is important that we limit the potential damage caused by a compromised "main" thread. For this reason, we restrict the "main" thread's access to solely protection domain 0. This prevents an adversary from disclosing sensitive information held either by "internal" threads, or individual connection threads.

A.4 Redis Proof of Concept

An adversary can exploit the stack based overflow described in Section 5 by running `EVAL "struct.pack('>I262914270', '42')" 0` in Redis' Lua interpreter. This causes the `putInteger` function in the Lua interpreter to write the value 42 into the address given `262914270` bytes above the victim stack buffer [1]. In a real deployment, an adversary would be likely unable to directly issue `EVAL` commands, but they may be able to compromise a privileged client

that can issue the newer EVAL_RO command. This command only allows Lua scripts to read from the data store. If this vulnerability were to be introduced again, as it was in 2020, then an adversary could use the vulnerability to break out of the read-only sandbox provided by EVAL_RO. ThreadLock prevents an adversary with an arbitrary write primitive with a 31-bit range above the victim buffer from corrupting data store pages. We argue that shrinking the attack surface for this exploit demonstrates that ThreadLock can protect program components, like language parsers, that typically suffer from security bugs.