Remote Direct Memory Introspection

Hongyi Liu, Jiarong Xing, and Yibo Huang, *Rice University;* Danyang Zhuo, *Duke University;* Srinivas Devadas, *Massachusetts Institute of Technology;* Ang Chen, *Rice University*

https://www.usenix.org/conference/usenixsecurity23/presentation/liu-hongyi

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Remote Direct Memory Introspection

Hongyi Liu Jiarong Xing Yibo Huang Danyang Zhuo[†] Srinivas Devadas[‡] Ang Chen

Rice University [†]*Duke University* [‡]*MIT*

Abstract

Hypervisors have played a critical role in cloud security, but they introduce a large trusted computing base (TCB) and incur a heavy performance tax. As of late, hypervisor offloading has become an emerging trend, where privileged functions are sunk into specially-designed hardware devices (e.g., Amazon's Nitro, AMD's Pensando) for better security with closer-to-baremetal performance.

In light of this trend, this project rearchitects a classic security task that is often relegated to the hypervisor, *memory introspection*, while only using widely-available devices. Remote direct memory introspection (RDMI) couples two types of commodity programmable devices in a novel defense platform. It uses RDMA NICs for efficient memory access and programmable network devices for efficient computation, both operating at ASIC speeds. RDMI also provides a declarative language for users to articulate the introspection task, and its compiler automatically lowers the task to the hardware substrate for execution. Our evaluation shows that RDMI can protect baremetal machines without requiring a hypervisor, introspecting kernel state and detecting rootkits at high frequency and zero CPU overhead.

1 Introduction

Security and performance are both first-order objectives at cloud scale, yet today's hypervisors struggle in both aspects. Hypervisor-based virtualization introduces a large TCB with millions of lines of low-level code, where CVEs (Common Vulnerabilities and Exposures) are continuously unearthed [55, 109]. Meanwhile, they also incur a heavy "virtualization tax," consuming significant CPU cycles, taking away resources from revenue-generating VMs, and creating performance contention with tenant workloads [82]. To achieve better security and performance, cloud providers are increasingly invested in hypervisor offloading, using tailor-made hardware devices [4, 8, 14, 19, 37, 39] with closed-source designs.

These emerging devices are particularly important in baremetal-as-a-service (BMaaS) offerings, where entire installations are provided to a single tenant without a software hypervisor. BMaaS not only provides ideal performance and isolation for the tenant, but also increases the provider's revenue as 100% of CPU cycles are for sale; it has gained a foothold in all major clouds [2, 7, 9, 10, 16, 33]. Owing to the absence of the hypervisor, security tasks are often anchored in these customized devices—with Amazon's Nitro [8], Intel's IPU [15], AMD's Pensando [4], and Microsoft's Fungible [14] vying for the market. These devices are attached to host servers as PCIe peripherals, akin to network interface cards (NICs), but their execution environments are shielded from host CPUs. They run protection tasks with a higher privilege than the OS or hypervisor. Since this mode of execution operates beneath what is traditionally known as Dom0, we will henceforth call this paradigm "Dom(-1) security."

Remote direct memory introspection, or RDMI, rethinks a classic security task in light of the Dom(-1) paradigm. Kernel memory introspection [61, 64, 65, 89] is an important forensic technique. It enables the cloud provider to perform security telemetry and detect signs of malice (e.g., rootkits), while staying transparent to the tenants by virtue of operating underneath their workloads. Traditionally, this is relegated to the hypervisor, which periodically acquires memory snapshots from guest VMs for security analysis (e.g., reconstructing `task_struct` lists from raw memory). In contrast to this conventional approach, RDMI sinks memory introspection tasks into the hardware layer by a whole-stack redesign; moreover, it only uses COTS (commercial-off-the-shelf) devices available to everyone instead of closed-source devices. The key enabler for RDMI is the increasing deployment of commodity programmable hardware in the cloud. In particular, RDMA NICs (RNICs) that enable remote direct memory access [50, 58], and P4 programmable switches that can realize hardware control loops [81, 97], form its Dom(-1) substrate:

- *Memory datapaths:* RNICs expose host memory to remote clients, providing a telemetry channel to acquire memory snapshots over a network in a granular manner. The conventional use of RDMA is to accelerate application-layer cloud workloads [50, 58], whereas we use it as a vantage point for kernel memory visibility. RDMA datapaths are simple and fast, and remote accesses are fully transparent to the introspected host.
- *Control loops:* Kernel introspection is a complex task that goes beyond individual memory accesses—e.g., it might need to fetch the Linux process linked list starting at `init_task`, parse its next pointers, and traverse the entire list of `task_structs`. This requires a more expressive programming model than RDMA. We observe

that introspection control loops are an ideal fit for programmable switches, which serve as a platform for realizing new control protocols at hardware speeds [81, 97].

Combined, RDMI executes in Dom(-1) with hardware-based memory accesses (using RDMA NICs) and inspection (using programmable switches) at ASIC speeds. This operating regime also enables RDMI to introspect baremetal installations without a hypervisor. It can be deployed to a ToR (Top-of-Rack) switch that serves a set of baremetal servers equipped with RNICs, offering security protection with novel properties not found in hypervisor-based introspection:

- *Baremetal:* It introspects kernel memory in baremetal installations, without requiring a hypervisor.
- *Remote:* The introspection engine is disaggregated from host CPUs and executes over the network.
- *Efficient:* Both the datapath (memory operations) and the control path (introspection logic) are realized in ASICs.
- *Commodity:* It relies on widely available, COTS hardware technologies without any modification.
- *Programmable:* Introspection tasks can be programmed in a declarative language with a few lines of code.

On the last point, RDMI abstracts away the complexities of the ASICs and the intricacies of kernel introspection from the user. Instead of directly asking the user to program low-level RDMA and P4 ASICs, which would be burdensome and error-prone, RDMI exposes a set of functional operators to specify a variety of introspection tasks. Users program against a uniform "kernel graph" abstraction, where vertexes are the kernel data structures and edges are the pointer relations. Supporting this abstraction are the RDMI compiler and runtime that facilitate its Dom(-1) execution. The RDMI compiler maps a query via an intermediate representation (IR) that manipulates an abstract introspection machine for kernel traversal, with its instruction set instantiated in RDMA and P4 ASICs. The RDMI runtime provides auxiliary utilities (e.g., driving the introspection to different kernel locations and fetching kernel memory with RDMA operations), also shared across tasks. This design enables runtime programmability [104, 107]—queries can be reconfigured in a live manner by the compiler generating different control plane configurations without downtime.

We have developed a RDMI prototype and conducted a comprehensive evaluation, with the following findings. The RDMI defense platform is capable of introspection frequencies that are orders-of-magnitude higher than hypervisor solutions, and it effectively performs memory telemetry and detects rootkits in baremetal machines. Moreover, security protection does not require any CPU involvement and incurs minimal performance disturbance to tenant workloads. We have released our prototype in open source [34].

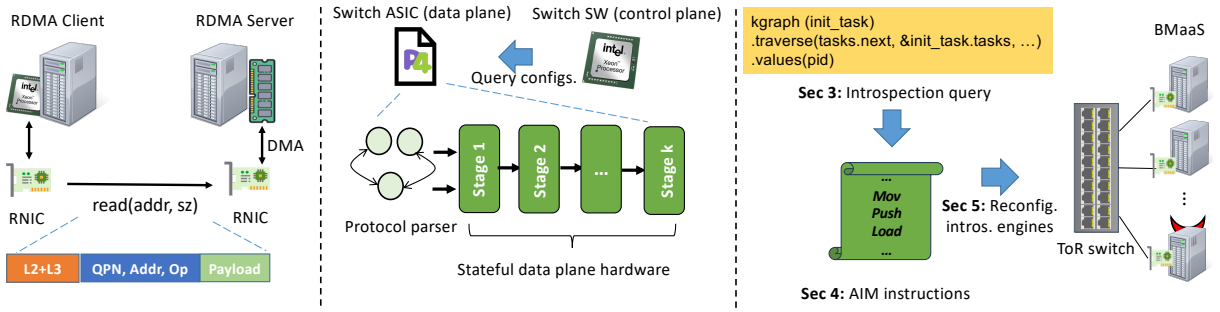
2 Rethinking Memory Introspection

Memory introspection is an important security task for the cloud [57, 64, 76, 90, 91, 110]. An abridged view of the long body of work can be summarized as follows. Since its inception two decades back [61], researchers have shown that rich security insight can be gleaned from memory snapshots. This works through the hypervisor scanning important kernel data structures in the VMs to detect signs of malice (e.g., rootkits). Operating in Dom0, introspection code executes with higher privilege than tenant VMs, which reside in DomU. Thus, it has a full view of VM memory and can inspect any kernel locations. Sinking security protection from VMs into the hypervisor also means that introspection can be provided "as-a-service" transparently to the tenants. The only setup parameters required by the hypervisor are metadata about the guest kernels (e.g., kernel versions, ASLR offsets), and from there on, the introspection code navigates the kernel state by itself. To conquer the "semantic gap" [65], introspection programs must ingeniously piece together disparate data from raw memory bytes—e.g., it may enumerate Linux's `task_struct` process descriptors from raw memory.

Our project rethinks memory introspection in light of the trend of hypervisor offloading to Dom(-1) hardware. Isolating tenants with hypervisor software has been the de-facto cloud paradigm, but Dom(-1) technologies are chipping away at this abstraction. This is not only due to the decline of Moore's law necessitating better use of CPU cycles, but also a desire for stronger security due to leaner hardware TCBs. Historically, the desire for baremetal execution was felt in various virtualization hardware features (e.g., DPDK [13], SPDK [35], VT-x [20], MPK [17, 102]), but the recent rise of baremetal-as-a-service and Dom(-1) hardware devices are a more significant milestone. Industry vendors have developed tailor-made devices [1, 4, 8, 14, 19, 39], where virtualization functions are not only offloaded to hardware but also gated from host CPUs via an "airgap" for security. Implementing security functions in Dom(-1) reduces the TCB, minimizes interference with the tenants, provides stronger protection in case of host compromises, and saves operational costs as more server CPU cycles are made available to tenants. RDMI rearchitects memory introspection to operate in Dom(-1), but does so only with COTS devices.

2.1 Remote direct memory introspection

Introspection datapath: The RDMI memory datapaths are built upon one-sided RDMA "verbs." For instance, RDMA READ operations take memory addresses and sizes as parameters, and the requests are encapsulated as Ethernet packets and sent over the wire. Once the requests arrive at the remote machine, the Ethernet packets are translated by the NIC hardware into DMA requests over PCIe, eliminating remote CPU involvement from the datapath and achieving ASIC speeds for memory accesses. In the context of introspection, the RDMI READs might fetch a `task_struct` or several of its fields. RDMA connections are established using queue pairs (QPs) with unique identifiers (QPNs, or queue pair numbers) at both the sender and the receiver sides. By default, RDMA uses vir-



(a) RDMA one-sided verbs (b) P4 programmable switches (c) Remote direct memory introspection

Figure 1: The Dom(-1) substrate of RDMI includes (a) RNICs and (b) P4 programmable switches; (c) the RDMI workflow.

tual memory addresses and thus requires address translation at the NIC hardware, but it is configurable to use physical addresses directly as well [88, 101]. This is important for RDMI as it manipulates kernel objects, most of which are directly (i.e., linearly) mapped to the physical addresses.

Introspection control. While one-sided RDMA is a promising start, memory introspection goes far beyond individual memory operations. Introspection tasks require various types of kernel traversals (e.g., traversing a linked list of `task_structs`), which in turn involve pointer arithmetic, range checks, and complex control flow. One naïve approach is to implement the control path in software (e.g., on another server under the same rack). However, many downsides of driving RDMA operations in software have been noted by prior work, such as CPU cycle wastage due to polling [46], and longer latency due to software processing [42]). More importantly, this offsets our goal of Dom(-1) execution in hardware ASICs. Our solution is inspired by recent projects that use P4 programmable switches to implement control logic that drives RDMA tasks [73, 77, 105]. RDMI executes the control loop at hardware speeds inside a programmable switch, driving the memory introspection datapath.

2.2 Overview

Figures 1(a)-(b) depict the Dom(-1) substrate and (c) shows the key components of RDMI. To the best of our knowledge, RDMI is *the first defense platform* that a) exposes a declarative interface for users to articulate introspection tasks; b) compiles a wide range of such tasks to a reconfigurable set of hardware engines; c) executes introspection tasks at ASIC speeds with zero CPU overheads.

Threat model. Our threat model is that of a fully untrusted kernel (e.g., OS compromises due to kernel-level rootkits), thus the OS may exhibit arbitrary behaviors and is capable of removing or tainting any software security agent running on host CPUs (e.g., kernel modules). However, we assume that the kernel can boot into a known-good state (e.g., by leveraging trusted boot [38] hardware) and compromises only occur after that during runtime. This trusted setup also initializes RDMI—upon boot, we create a number of RDMA connections between the switch and the introspected machine, and grant these connections physical access to the host memory.

Operator	Description
`kgraph(addr)`	Initialize traversal at kernel `addr`
`traverse(ptr_nxt, ptr_end, type)`	Traverse `ptr_nxt` until `ptr_end` with `type`
`in(ptr)`	Deference `ptr` into a different data structure
`iterate(array, n, type)`	Iterate an `array` of `type` for `n` steps
`values(f1, .., fn)`	Acquire values from current address
`assert(pred1, .., predn)`	Assertion on acquired values

Table 1: Introspection operators. Highlighted are new operators or those that take a different meaning from Gremlin [95].

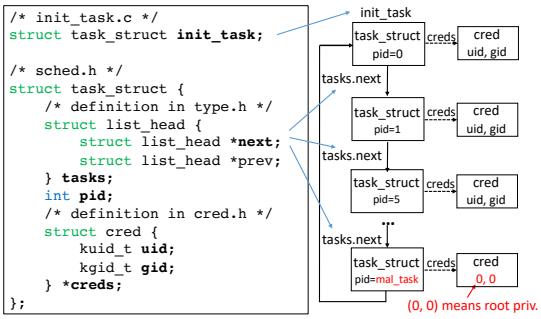
After the setup, we assume that the Dom(-1) substrate, as well as the introspection programs that it hosts, are trusted. Notably, at runtime, the trusted computing base excludes software hypervisors, which is a sizable reduction.

Non-goals. There are several worthwhile goals that are nevertheless beyond the scope of RDMI. Although RDMI enables expressive policies to be developed, our goal is not to propose new introspection policies that are more adept at detecting kernel compromise. Similarly, improving detection accuracy with new analysis algorithms is also not our focus. Section 8 also describes a few other limitations in detail.

3 Programming Introspection Queries

RDMI exposes a declarative interface for users to specify introspection tasks, so that they are not burdened with low-level operations with baremetal RDMA and P4 ASICs. We observe that a custom programming model is possible because RDMI tasks are highly specialized, essentially treating the kernel data structures as a graph and traversing the graph following pointers. Thus, we propose a domain-specific language (DSL) drawing inspiration from a widely-used graph query language, Gremlin [95]. Table 1 includes the key operators, and we showcase their expressiveness with concrete tasks.

Q1: Task list traversal [40]. Let us start with a "hello world" example, where the user wishes to query all active processes and their IDs. This functionality is akin to the 'pslist' volatility toolkit [40] for memory dump analysis. Linux uses `struct task_struct` as the data structure for a process, organized in a linked list with the global kernel symbol `init_task` as the entry. Each `task_struct` contains key process attributes—e.g., process IDs (`int pid`) and credentials (`struct cred`; used in Q2). We depict the data structures and key variables.



RDMI articulates this traversal in three lines of code below. A useful mental model for a RDMI query is that of a "cursor" pointing to a specific kernel address, which moves about across the kernel graph based on the introspection logic:

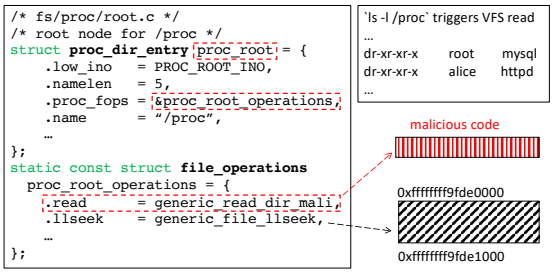
```
1  /* Traverse all tasks and acquire pids */
2  kgraph(init_task) // traversal source
3  // traverse init_task.tasks.next until wraparound
4  .traverse(tasks.next, &init_task.tasks, task_struct)
5  .values(pid) // query pid for each task
```

A query always starts with a `kgraph` operator (Line 2), which initializes the cursor to some predefined address, such as the global symbol `init_task` whose address is statically determined upon boot. Line 4 defines the footprint of the traversal, performing a sequence of pointer chasing operations with the `ptr_nxt` field (see Table 1 for operator arguments; in this case the argument is the `tasks.next` field in `task_struct`) until it encounters `ptr_end` (in this case set to `init_task.tasks`). In other words, the traversal halts when it wraps around and revisits `init_task`. The `type` is set to `task_struct`, so RDMI understands the data structure type of each traversed element. Finally, Line 5 uses `values(pid)` to acquire the process ID field in each visited element. Query results are forwarded to a logging server for further analysis.

Q2: Privilege escalation analysis [5]. This query traverses each `task_struct` as in Q1, but it takes an excursion from the linked list to another data structure `struct cred`, which stores process credentials (user ID `uid`, group ID `gid`). Non-root processes have `uid` and `gid` values larger than 1000; a rootkit may maliciously modify these values to zero for some user process (e.g., a Shell) to escalate its privilege to root access [5]. This query is a four-liner. Line 4 zooms `in` on the external data structure `struct cred`, which hangs off of the linked list. (Note that cursor movements in Q1 do not require `in`, as the visited fields are contained in the current data structure (i.e., `struct task_struct`) for traversal.)

```
1  /* Credential telemetry for all processes */
2  kgraph(init_task)
3  .traverse(tasks.next, &init_task.tasks, task_struct)
4  .in(creds)
5  .values(uid, gid)
```

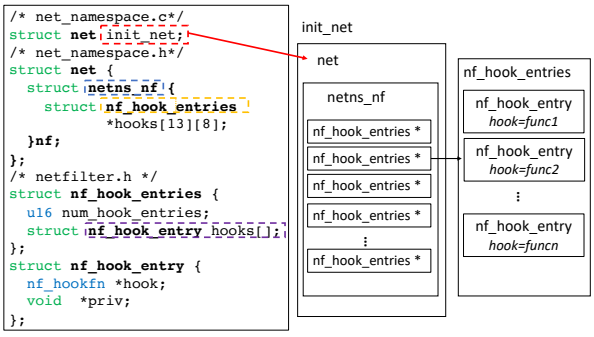
Q3: Virtual filesystem hook detection [76]. A rootkit may modify function pointers to divert execution to its malicious code. Q3 asserts that VFS function hooks must be within a known-good range. As depicted below, `/proc` is a virtual filesystem providing administrative utilities—e.g., user-level forensic tools such as `ps` rely on information from `/proc`. Its root inode is represented by the `proc_root` data structure. Filesystem operations eventually invoke read, llseek, and other file operations, which are specified as function pointers in `struct file_operations` as contained in `proc_root`.



By modifying the function pointers, a rootkit can manipulate the forensic outputs and hide certain processes from such administrative tools [76]. This RDMI query checks that the `read` function pointer must be within a known-good range (e.g., kernel text from `0xffffffff9fc00000` to `0xffffffffa08031d1`). The assertion in Line 4 evaluates a predicate and triggers notifications upon failure.

```
1  kgraph(proc_root)
2  .in(proc_fops)
3  .values(read)
4  .assert(KERN_TXT_BEGIN < read < KERN_TXT_END)
```

Q4: Network filter hijacking detection [25]. Netfilter [26] is a framework within the network stack, allowing registered callback functions upon packet events. Rootkits commonly inject adversarial callbacks to intercept network traffic. For instance, a rootkit may watch for port-knocking packet sequences as command-and-control signal for triggering an attack, and then drop these packets immediately [64]. As shown below, netfilter hooks are retrieved at the `init_net` symbol where `struct netns_nf` is stored. Inside `struct netns_nf`, `hooks` holds a two-dimensional array, and each array element points to a `struct nf_hook_entries`. This query requires a two-dimensional, nested traversal.



Lines 2+6 denote the nested traversal, where `iterate` operates on array elements instead of linked lists. Line 4 dereferences the value contained at the current array element as a

pointer, resulting in an excursion to a different data structure `struct nf_hook_entries`. This struct contains a second array of registered hooks, and `num_hook_entries` is the number of entries. Iterating through this dynamically-allocated array, RDMI checks each of the hook functions.

```
kgraph(init_net)
.iterate(nf.nf_hooks, 13 * 8, ptr_t)
//NFPROTO_NUMPROTO=13, NF_MAX_HOOKS=8
.in(this) // deref current value, see appendix
.values(num_hook_entries)
.iterate(hooks, num_hook_entries, nf_hook_entry)
.values(hook)
.assert(KERN_TXT_BEGIN < hook < KERN_TXT_END)
```

Q5-Q11. RDMI is expressive enough to support a range of introspection queries, summarized in Table 2. The Appendix contains the detailed queries and descriptions.

Policy	LoC	Policy	LoC
P1. Task list traversal	3	P7. Process memory map check	7
P2. Privilege escal. analysis	4	P8. Keyboard sniffer check	5
P3. VFS hook detection	4	P9. Module list traversal	4
P4. Netfilter hijacking detection	7	P10. Afinfo operation check	6
P5. TTY keylogger check	11	P11. Open file list	11
P6. Syscall check	4	-	-

Table 2: Example RDMI tasks. Code in Appendix.

4 Abstract Introspection Machine

We now describe how the RDMI compiler decomposes operator-level introspection logic into hardware-level implementations through an intermediate language. We draw inspirations from existing projects that compile functional operators to P4 programmable switches [63, 72, 108]); however, unlike existing compilers that directly lower the policy onto the hardware layer, RDMI introduces an indirection layer, which is an intermediate language that manipulates an *abstract introspection machine* (AIM). The key benefit provided by the AIM layer is to support runtime programmability [107]—that is, the ability to perform live query reprogramming without taking down the deployment. Existing work [63, 72, 108] compiles each security task into a different P4 program, so deploying a new query requires reflashing the switch with a different program. This incurs downtime and cannot be performed in a live manner [104], so the switch is fixed to specific queries and cannot be reprogrammed with a new task on demand. In RDMI, the AIM layer exposes a minimalistic set of five instructions, which are instantiated in hardware and shared across RDMI operators for runtime programmability.

4.1 Designing the AIM

RDMI achieves runtime programmability by designing a "master" P4 program that provides several introspection engines in hardware, corresponding to five AIM instructions: LOAD, MOV, PUSH, POP, JMP. Since all instructions are embedded in the master program, query changes do not require program modifications. Rather, deploying a new query only requires generating a new stream of AIM instructions, which in turn produces a different set of control plane configurations to the master program. Configurations are installed and removed from the switch control plane software without reflashing the hardware, so introspection tasks can be reprogrammed on demand. In addition, the AIM layer also enables resource sharing across introspection primitives—e.g., `traverse` and `iterate` have shared logic for pointer chasing (e.g, MOV) and memory acquisition (e.g., LOAD), which can be supported by the same underlying introspection engines. The five instructions operate on the AIM (virtual) registers and stack.

AIM registers: Registers store temporary introspection state (e.g., memory addresses, loop bounds) and enable arithmetic operations (e.g., pointer arithmetic, bound checking). We use R_i to denote the i-th register allocated by the compiler. The LOAD(R, ADDR, SZ) instruction fetches a chunk of memory that starts at address ADDR with size SZ, and its variant LOAD(R, $CONST) assigns a compile-time constant to the register. Predicates over register values are used to implement control flow branches—the conditional jump JMP(PRED, L, L′) checks the predicate PRED (e.g., `R < LOOP_MAX`) and branches to the L/L′ labels in the instruction-level program (see §4.2). We use R_B to denote a special AIM register that holds the current introspection base address, such as the starting address of a `task_struct` under introspection, and R_B is used in conjunction with relative offsets within the data structure to fetch data. The MOV(ADDR) instruction rebases introspection to a new address by fetching the pointer stored at ADDR (i.e., pointer chasing), and its variant MOV($ADDR) sets the base to ADDR. All ADDR fields in the LOAD/MOV instructions are compiled into base addresses and offsets.

AIM stack: The stack is manipulated in last-in-first-out order for traversal loops, and each stack frame contains a previously used R_B. For instance, when traversing the `task_struct` linked list, the PUSH instruction pushes the current `task_struct` base to the stack top. This may be further followed by a MOV to rebase introspection to the next element. The POP instruction pops the stack top to R_B, restoring the previous introspection base and resuming work from there (e.g., returning from an inner traversal to the outer layer). Every nested traversal produces exactly one stack frame, so stack depth can be analyzed statically by the compiler.

4.2 Compiling to the AIM

RDMI compiles introspection operators to the AIM instructions enabled by the master program. `kgraph(addr)` initializes the introspection by setting R_B to the specified `addr`. `in(ptr)` is realized by a MOV instruction that rebases to a different data structure. `values(f)` is realized by a LOAD instruction to fetch the value. `assert(pred)` further performs predicate checking on the LOADed results. `traverse` and `iterate` are the most complex as they involve loops, and the loop body could further vary based on the query. RDMI compiles them into AIM instructions that implement a loop skeleton, but with loop bodies initialized to placehold-

ers; they are later filled by the compiler when processing the operators within the traversals. We show the skeleton for `traverse(ptr_nxt, ptr_end, type)`:

```
1       Mov($ptr_nxt)  //move base to ptr_nxt addr
2   L:  //loop skeleton compiled from traversal
3       Push    //record base addr before moving away
4       /*
5           loop body placeholder, to be compiled from
6           subsequent operators, e.g., acquiring values
7           from the current element or nested traversals.
8       */
9       Pop     //back from inner traversal
10      Mov(ptr_nxt) // move base to visit next entity
11      Jmp(R_B != ptr_end, L, Lend)  //loop guard
12  Lend: //traversal completes
```

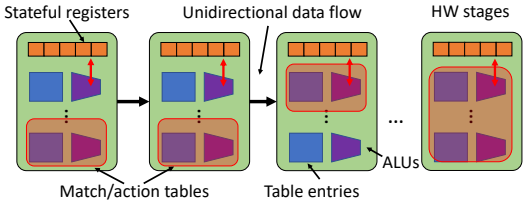
Lines 1+10 successively move the cursor across a linked list of elements. To support nested traversals, Lines 3+9 use PUSH and POP to maintain base addresses in the stack. Line 11 checks for loop termination conditions. The loop body is left as a placeholder denoted by Lines 4-8, and it will be generated by the compiler when processing subsequent introspection primitives. For instance, the compiler may generate LOAD instructions if the operator nested in the loop is `values(f)`.

`iterate` follows a similar compilation strategy with `traverse`. The Appendix includes more details for reference.

5 Reconfigurable Introspection Engines

We now describe how the AIM instructions are instantiated in hardware engines, which can be reconfigured to implement different AIM instruction streams. At a high level, reconfiguration is achieved by installing control entries generated from different AIM instructions onto the match/action tables from the control plane. To explain this design, we first provide more background information on the Dom(-1) substrate.

P4 programmable switches consist of a sequence of hardware *stages* in their most popular models (i.e., Intel Tofino). A P4 program is a pipeline of *match/action tables* that are allocated on the stages, which select specific packet headers using match fields and activate processing actions. The tables have access to *stateful registers*, which are persistent memory that keeps state across packets, as well as *ALUs* (arithmetic logical units) that are capable of performing arithmetic operations with registers. A packet can only access each stage and its resources (e.g., registers, ALUs) once, and each ALU supports at most two distinct arithmetic operations. The AIM instructions are instantiated by a set of match/action tables, and the table entries are generated by the RDMI compiler and populated by the switch control plane to realize different introspection policies. This is the control path for introspection.



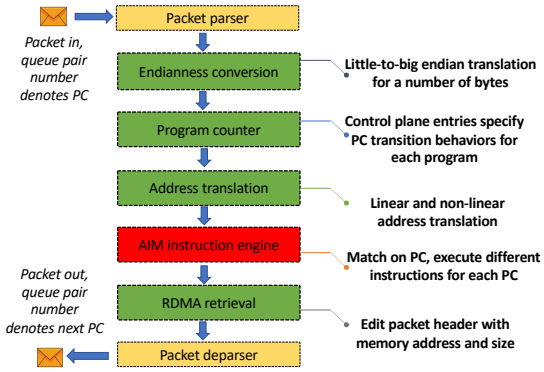
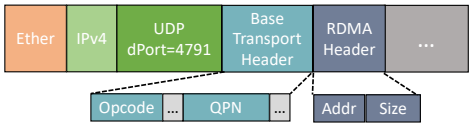


Figure 2: Reconfigurable introspection engines in RDMI. Red: AIM instruction engine; green: the runtime system engines.

RDMA NICs transform Ethernet packets received over the wire into DMA transactions over the PCIe bus for memory access, and vice versa. We depict the RoCEv2 [36] format, a commonly used RDMA protocol, with select fields:



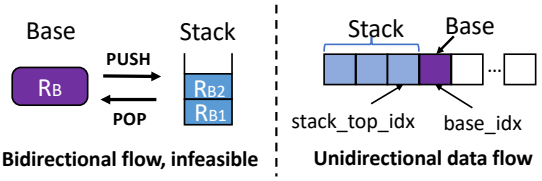
A packet carries a queue pair number (QPN) that uniquely identifies an RDMA connection. Other header fields include the memory address to read from, the read size, as well as the read opcode itself. RDMA packets are encapsulated in UDP, IP, and Ethernet protocols, and they are recognized by the host and the switch by their distinct destination port number (i.e., 4791). This is the datapath for memory introspection.

Introspection engines are depicted in Figure 2. The AIM instruction engine matches on the program counter (PC) carried in the packet header (§5.2), and executes different instruction streams based on the match/action entries. These entries map from PCs to their corresponding AIM instructions. The compiler generates these entries from the AIM instructions, and the control plane software installs them into the match/action tables to realize different programs. RDMI also has a runtime system, with four engines for endianness conversion, program counter, RDMA retrieval, and address translation, respectively. The lifetime of a typical introspection packet in RDMI is as follows. When the switch receives an RDMA packet, it first extracts specific headers (e.g., memory content) from the packet and performs endianness conversion. Next, the PC engine advances the program execution based on PC transition rules, which are also compiled from the AIM instructions as match/action entries. As needed, the packet also triggers address translation and page table walk. The AIM instruction engine then executes a batch of instructions, and triggers the RDMA retrieval engine for the next step of introspection. Thus, an introspection task requires several rounds of RDMA requests, each of which triggers an iteration of switch execution over the next several instructions.

5.1 Reconfigurable AIM instruction engines

We now describe how the introspection engines are instantiated, deferring the runtime system description to §5.2.

Push, Pop, Mov: These instructions operate on the base register and the stack: pushing the base register value onto the stack, popping off the base from the stack, and modifying it, respectively. The stack is created using an array of stateful registers in P4, with additional designs to overcome the constraints imposed by the sequential hardware stages. As shown below, PUSH transfers data from the base register onto the stack, and POP in the other direction, requiring bidirectional data flow. However, if we allocate the base register at stage n and the stack at stage $n+1$, backward access will incur heavy overhead; reversing their layout raises similar problems.



Our design addresses this by observing that the stack depth can be statically analyzed by the compiler. Thus, we integrate the stack and base into a single register array, as illustrated above. Two packet metadata variables, `stack_top_idx` and `base_idx`, record the logical stack top and the base register, although physically they reside in the same register array. PUSH increases `stack_top_idx` by one, subsuming the current base without data copy. POP decrements the stack top index by one and updates the base to the popped value, again with unidirectional data flow. Further, the RDMI compiler statically computes the value for `stack_top_idx` and `base_idx` at each point of the program execution (as denoted by the program counter/PC; details in §5.2). It produces match/action entries that match against the PC values and retrieves the current indexes for stack and base register operations; different policies result in different table entries. MOV only produces unidirectional data flow, modifying the base register to a new value. When rebasing introspection to a new address, the RDMA retrieval engine fetches the data from kernel memory.

Load, Jmp: These instructions operate on the AIM registers. The compiler allocates a stateful register in P4 hardware for each AIM (virtual) register, with an optimization that statically analyzes whether a fetched value via LOAD will be used in subsequent instructions. For instance, a LOAD(R_1,ADDR,SZ) instruction in conjunction with a JMP(R_1--,L1,L2) will result in a stateful register allocated for R1. On the other hand, if a LOADed value is only used in the current round and never reused again (e.g., if the LOADed data is inspected but does not trigger additional pointer chasing), the compiler allocates a P4 metadata variable instead of a stateful register for resource savings. (Metadata variables are akin to packet headers, temporary and discarded after the packet leaves the switch.) Recall that the LOAD instruction supplies a memory address and read size; the match/action table modifies the current packet's headers to transform them into a proper RDMA read packet, and emits it from the RDMA retrieval engine. When the response packet arrives, the P4 switch distinguishes the response based on the queue pair number, parses its value, and stores it into the stateful register or metadata variable, and the LOAD instruction retires. JMP is supported by match/action tables that use an ALU to check the predicate over the register value, producing exactly two branches as required by the ALU constraints. As before, control plane entries are generated by the compiler to determine the specific checks and branching locations. If a branch is taken, the PC is modified to reflect the control flow transfer.

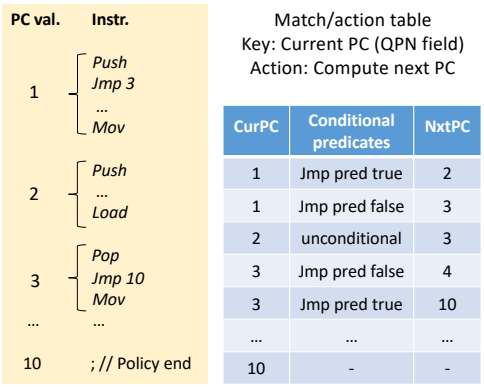
5.2 Reconfigurable introspection runtime

Supporting the instruction engines is the RDMI runtime system, with four components depicted in Figure 2.

Endianness conversion. LOAD and MOV instructions fetch data from host memory. Since host data uses little-endian encoding, we develop a translation engine to convert the endianness of the RDMA read output. Its match/action tables are configured with entries that match on the read size (i.e., number of bytes) and perform bytewise conversion as actions.

RDMA retrieval. LOAD and MOV instructions supply kernel addresses to the retrieval engine, which handles interactions with the kernel memory via RDMA. The match/action tables for the retrieval engine are reconfigurable to edit the packet header with the RDMA read opcode, address, size, and the queue pair number before sending it out to the host.

Program counter. To keep track of instruction execution, we need a program counter that specifies the next AIM instruction to be executed. Execution may either fall through to the next instruction (for non-JMP instructions) or branch to a different location (for JMPs). The PC value is encoded as the queue pair number (QPN), which is an RDMA packet header. The PC transition logic is realized by match/action tables that match on the current PC value as the key and compute the next PC as the action. Compared to a naïve design that uses a P4 stateful register to record the PC, carrying PC values in packet headers is a judicious design choice as it enables better support for concurrent queries. As each RDMA packet comes in, RDMI locates the execution context (i.e., the query it belongs to as well as the instruction executed) based on its QPN without ambiguity. Consider some example match/action entries that implement the PC transition for a batch of AIM instructions: First, notice that the PC is not per-instruction but counts blocks of AIM instructions; each block ends with either a MOV or a LOAD instruction. This is because MOV/LOAD sends the packet out and PC (as a packet header) will disappear from the switch. Eventually, its response packet comes back asynchronously, and we need to determine where to resume the execution. This, in turn, requires us to update the QPNs of outgoing packets with the next PC values, so that match/action processing will resume based on the incoming packets' QPNs. Moreover, we can see that JMP instructions

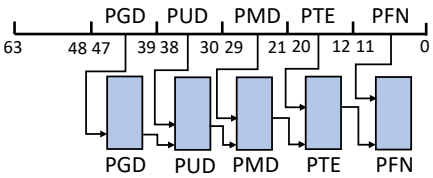


Match/action table
Key: Current PC (QPN field)
Action: Compute next PC

CurPC	Conditional predicates	NxtPC
1	Jmp pred true	2
1	Jmp pred false	3
2	unconditional	3
3	Jmp pred false	4
3	Jmp pred true	10
...	...	...
10	-	-

are executed based on a predicate evaluation as part of the match/action processing (e.g., PC 1 may transition to 2 or 3 depending on the branching condition). Finally, a default label (e.g., PC 10) represents policy termination.

Address translation. Linear kernel addresses (e.g., direct mapping area of kernel text) are translated by applying a fixed offset to obtain the physical address. Non-linear addresses (e.g., kernel modules) require a page table walk. Thus, the RDMI address translation engine is configured with two parameters—a) the translation offset, which remains fixed for a single boot, for linear address translation, and b) the global kernel symbol address for `init_top_pgt`, which resides in the linear address space and holds the entry to the kernel page table. Linear address translation works by extracting the virtual page number from an address and then applying an offset in the ALU. Non-linear translation requires several steps:



The figure above depicts a typical four-level page table, where a memory address is segmented into several components: a) bits [47..39] as the index to the first-level PGD (page global directory) table, as located by the global kernel symbol, b) bits [38..30] for the second-level PUD (page upper directory), c) bits [29..21] for the third-level PMD (page middle directory), and d) bits [20..12] for the PTE (page table entry directory). Successive RDMA packets are sent to fetch the respective translation entries to compute the physical frame number (PFN). The page offset (i.e., bits [11..0]) is then concatenated with the PFN to form the actual physical address.

5.3 Reconfiguring queries at runtime

Each of the introspection engines is reconfigurable from the switch control plane. Thus, adding, removing, or colocating queries is a seamless operation without downtime. To support co-existing queries, RDMI simply needs to configure the execution context (i.e., PC values and their QPNs) and PC-to-instruction mappings for each of the queries individually.

6 Security analysis

Trusted boot. RDMI relies on a trusted boot process, where the RDMA NIC is initialized and appropriate QPNs (which denote the various PCs) are assigned to RDMI so that it can perform subsequent introspection. This is a practical assumption, as the boot process can be protected by initializing the system with a known image and relying on hardware support available in modern CPUs [38].

Runtime TCB reduction. Hypervisor-based introspection has a large TCB—the virtualization layer often exceeds several million lines of C code [109]. In RDMI, after the trusted boot, the software TCB includes the P4 master program and its control plane entries generated by the compiler, totally less than 3K lines. Even the RDMI compiler itself can be excluded from the software TCB, as eventually only its outputs are deployed to the switch. If desired, one could even formally verify the correctness of P4 programs [59, 80] as a further step for assurance in RDMI. For hypervisor-based solutions, this is much harder to achieve. Dom(-1) hardware encloses vendor-provided firmware, which RDMI relies on for correct execution. The size of device firmware varies, and an example NIC (Netronome Agilio [29]) contains 52K lines of code in its firmware.

Runtime tampering. We now consider adversaries that specifically attempt to tamper with RDMI operations. Since RDMA bypasses kernel and CPU software, this already provides a degree of stealth by virtue of executing in Dom(-1). Nevertheless, a powerful attacker may attempt to guess RDMA configurations (e.g., queue pair and sequence numbers) and launch attacks in the following scenarios.

The adversary can launch (1) a disconnection attack where she forcibly shuts down RDMI's queue pairs—or even the entire RNIC hardware itself—so that operations on these queue pairs will fail. RDMI detects such attacks by constantly monitoring the packet-level behavior for each of its queue pairs, and by raising alarms when certain queue pairs are unresponsive despite read requests. In addition, the adversary can also launch (2) an injection attack without shutting down queue pairs, where she attempts to guess the correct RDMA sequence number used for introspection, and inject spoofed data (e.g., incorrect `task_struct` addresses) to confound RDMI. Our defense relies on the fact that the RNIC hardware will execute RDMI requests and produce responses with identical sequence numbers as the adversary's injected packets. (Otherwise, if the adversary injects packets with incorrect sequence numbers, they will be rejected.) Therefore, RDMI keeps track of the correct sequence numbers and raises alarms when it detects duplicate packets, which is a sign of a spoofing attack. We have developed both defenses as part of the P4 master program, and Figure 3 shows the results for two introspection experiments, where around t=2s and t=3s, respectively, an adversary launches the disconnection and spoof attacks. In both cases, the P4 switch is able to detect the malicious behaviors

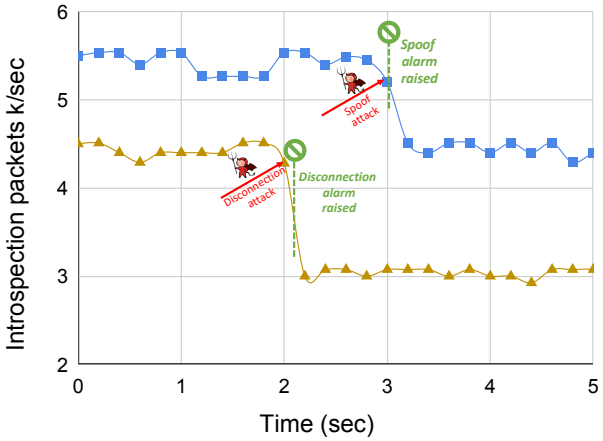


Figure 3: RDMI successfully detects disconnection and spoof attacks and raises alarms upon detection.

Policies	LoC (vs. libVMI)	#instr.	#entries
P1	3 (225)	8	109
P2	4 (217)	10	135
P3	4 (145)	8	161
P4	7 (167)	16	135
P5	11 (200)	20	234
P6	4 (135)	8	91
P7	7 (252)	18	195
P8	5 (151)	8	123
P9	4 (104)	8	119
P10	6 (138)	9	173
P11	11 (231)	20	226

Table 3: RDMI is expressive for a range of introspection policies and is much more concise than LibVMI implementations.

and raise alarms to the operator. The throughput drop also shows that the effect of disrupting the RNIC operations is noticeably different from the normal RDMI operations.

7 Evaluation

We present a comprehensive evaluation to answer three research questions: a) how well can RDMI support diverse introspection policies? b) how effective is RDMI in detecting rootkits? c) what are the introspection overheads of RDMI?

7.1 Prototype and setup

We have implemented RDMI in 5200 lines of code. The RDMI compiler (2700 lines of code in C++) ingests an introspection policy in our domain-specific language, and emits control plane configurations for the master program. The master program implements the introspection engines in a P4 program with 2500 lines of code. We deploy RDMI to an Intel Tofino Wedge 100BF-32X P4 programmable hardware switch, with 32x100Gbps ports connected to a set of servers. All servers come with six Intel Xeon E5-2643 Quad-core 3.40 GHz CPUs (24 cores), 128 GB RAM, and run Ubuntu 18.04 as the OS. Each server also has a Mellanox CX-4 RDMA NIC operating at 25Gbps.

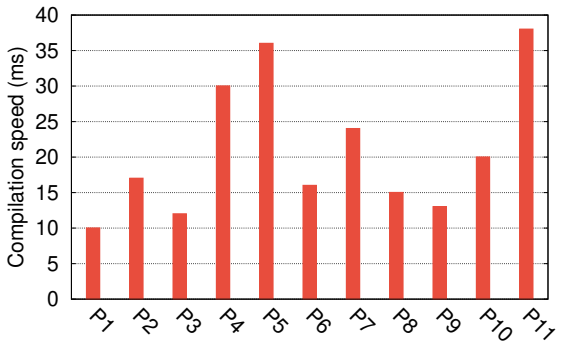


Figure 4: The RDMI compiler works efficiently.

Our primary baseline defense is LibVMI [22], a state-of-the-art hypervisor-based introspection engine that obtains and analyzes guest memory snapshots in software. We use the KVM/QEMU (v4.2.1) hypervisor, and have manually implemented policies P1-P11 using LibVMI (v0.13.0). A LibVMI program runs inside the hypervisor and establishes a KVMI (KVM introspection) socket with a guest VM on the same physical server, and issues acquisition requests via this channel. Since LibVMI cannot support remote or baremetal introspection, we have created another defense system by stripping RDMI of its hardware control loop in the P4 switch. The introspection program runs in a dedicated server to implement the control loop in software, and the introspected machine is connected via the same P4 switch that only runs a basic forwarding program; the memory datapath is still implemented using RDMA NICs. We call this baseline "Software RDMI," as it can be viewed as an intermediate step toward full RDMI assuming that the top-of-rack switch is not P4 programmable. We have implemented policies P1-P11 in software, and further integrated them with RDMA for remote memory acquisition.

7.2 RDMI language and compiler

We start by evaluating the domain-specific language and compiler in expressiveness, TCB size, compilation speeds, compiled configurations, and switch resource utilization.

Expressiveness. Table 3 shows the lines of code for each of the 11 policies in RDMI and in LibVMI implementations. RDMI supports the policies in at most 11 lines of code, where LibVMI implementations are one or two orders of magnitude larger—with 104-252 lines of code across policies. Further, LibVMI programs are developed inside the hypervisor, requiring low-level programming skills from the developer. We also show the number of AIM instructions compiled from the functional operators, as well as the control plane entry counts for each policy, which range from 91 to 234. Thus, the RDMI compiler successfully hides the task complexity and shifts a substantial amount of work inside itself, while automatically configuring different introspection tasks.

TCB reduction. The P4 master program has 2500 lines of code, and the control plane entries generated by the compiler are less than 210 lines across all policies. Thus, the runtime

Resource	ALU (%)	Hash unit (%)	SRAM (%)	TCAM (%)
Endian conversion	0	1.68	0.21	0
Program counter	2.08	3.61	2.6	0
Address translation	8.33	6.61	4.69	3.47
AIM instructions	4.17	4.81	3.12	6.94
RDMA retrieval	4.17	4.09	2.81	1.39
Overall	**22.92**	**26.68**	**16.15**	**11.8**

Table 4: Switch resource utilization with 11 policies

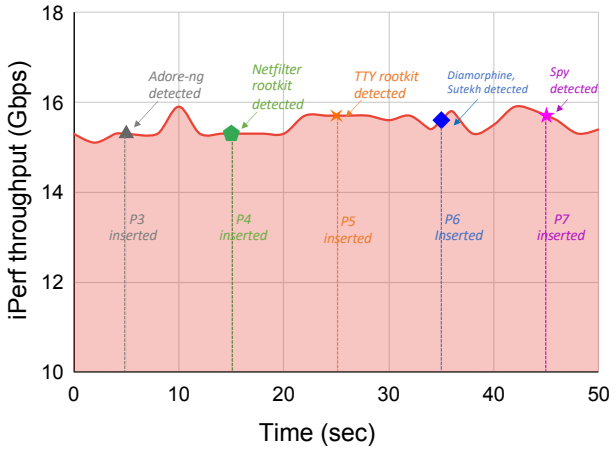


Figure 5: RDMI reprograms queries seamlessly.

TCB is smaller than 3K lines of code and configurations, a significant reduction compared to the size of a hypervisor.

Compilation speed. Next, we measure the turnaround time for the RDMI compiler to generate the control plane configurations for each policy. Figure 4 shows the turnaround time: across all policies, RDMI spends 10–38 milliseconds to produce the compiled configurations, which is very efficient. Also, the turnaround time is correlated with the number of AIM instructions and control plane configuration entries that the compiler needs to generate—more complex configurations tend to have higher compilation time (e.g., P11 > P10 > P9). RDMI supports policy composition naturally, as each policy results in its set of entries that are installed to the same set of introspection engines in the master program. Compiling multiple policies is equivalent to compiling each of them one by one, and then installing all the resulting entries to the switch (not shown, but see Figure 5 for concurrent queries).

Switch resources. Table 4 measures the switch resource utilization of the RDMI master program—installed with all 11 policies—and decomposes the usage across several components. In RDMI, header operations are performed in ALUs and hash units, stateful registers are supported in SRAM, and match/action entries are in SRAM and TCAM. Across all resource types, the switch has 11.8%-26.68% utilization, leaving plenty of room for other types of switch programs.

7.3 Detecting rootkits, remotely

We now evaluate the effectiveness of RDMI to detect rootkits in *baremetal installations* over the network *remotely*. We collected four rootkits [23, 24, 27, 28] that are commonly used in kernel security evaluation, and added two more by implementing attack mechanisms in existing projects [25, 64, 87]. RDMI is configured with all 11 policies in the switch.

Adore-ng [23] is a rootkit that has been evaluated in several existing detectors [64, 67, 76, 91, 94]. It hooks itself to function pointers in the kernel virtual file system, such as the inode lookup and file iterate operations. After hooks are installed, the rootkit will collect parameters passed from the inode lookup function and match them against predefined secrets for triggering privilege escalation of a requested process. Further, this rootkit covers its tracks by hiding information from administrative tools—the file `iterate` function hides data about malicious files, and its hook on `tcp_seq_afinfo` pointers hides network connections from `netstat`. RDMI successfully detected this rootkit from three policies. P2 detected a userland process whose privilege has been escalated; P3 and P10 detected function pointer values in the virtual file system and TCP stack that are outside the regular kernel text.

sutekh [28] is a rootkit that hooks into the `execve` and `umask` functions in the system call table (i.e., `sys_call_table`, which is an array of syscall pointers) [69]. Invoking hooked syscalls will result in modification to the credential data structures for specified userland processes. RDMI detected this rootkit using P2 (which detects escalation) and P6 (which detects system call table tampering).

Diamorphine [24] is a rootkit that also targets system call hooks, and it manipulates `kill`, `getdent` and `getdent64` [67, 74, 84, 92]. For instance, the `kill` syscall is repurposed as a communication mechanism between the rootkit and a userland process—e.g., `kill -sig -para` sends signals from the userland to the rootkit, and the signal numbers further trigger information hiding or privilege escalation capabilities of the rootkit. on behalf of certain userland processes. RDMI detected this rootkit with policies P2 and P6.

Spy [27] is a keyboard logging rootkit, which manipulates `register_keyboard_notifier` in the kernel to add itself to a set of consoles (i.e., `keyboard_notifier_blocks`) that receive notifications upon keystrokes [92]. The rootkit then converts the keystrokes into a buffer maintained by the `debugfs` virtual filesystem. Our system detected this rootkit using policy P8, which checks keyboard logging functions.

TTY rootkit is a rootkit that we have implemented using the techniques proposed in a related project [87]. It targets Linux tty units and manipulates the `receive_buf` function, which is a function called by `tty_driver` for sending characters to the tty line discipline. By doing this, it can hijack and eavesdrop on any data typed in a terminal. RDMI detected this rootkit with policy P5, which monitors the TTY activities.

Netfilter rootkit implements techniques utilized by two projects [25, 64] that target Linux Netfilter. It registers a Netfilter handler `nf_register_net_hook`, specifically at the location `NF_IP_LOCAL_IN`. It thus obtains control when receiving network packets, and can take arbitrary actions including monitoring port-knocking traffic [32] for activation and then dropping such traffic to avoid suspicion. RDMI detected this

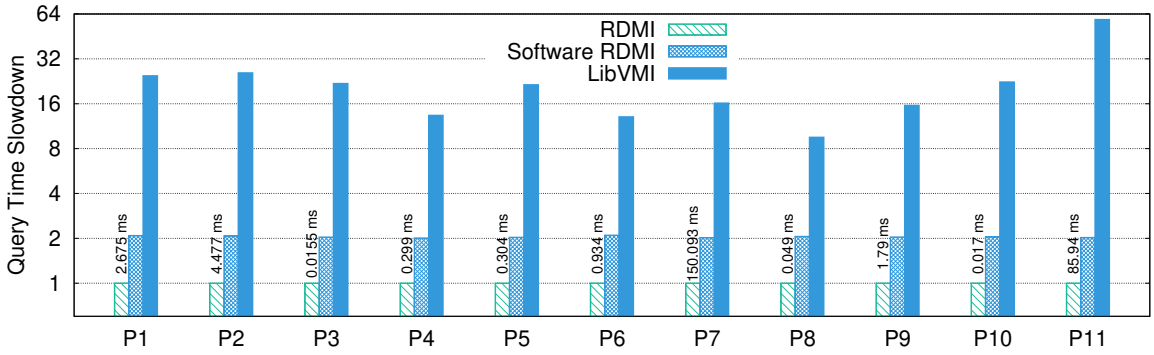


Figure 6: Introspection turnaround times of varying security policies with RDMI, Software RDMI and LibVMI. We normalize all systems based on the RDMI speed, and also include the RDMI turnaround times on top of the RDMI bars.

rootkit with policy P4 that checks the Netfilter system.

Dynamic, concurrent queries. To demonstrate RDMI's flexibility to deploy new queries at runtime, we start with the master program with an empty configuration, and the gradually add five policies to detect the above rootkits that are installed inside a server. Figure 5 shows the throughput of an `iperf` client during the reconfiguration, and labels the respective policies and the detected rootkits. We can see that query reprogramming does not disrupt the network transfer or impact service availability. In all cases, the added policies were able to detect the respective rootkits effectively. Without this capability, deploying new policies would require reflashing the switch and taking down the cluster for reconfiguration.

7.4 Benefits of baremetal security

Next, we showcase the benefits of baremetal security, as enabled by RDMI. As discussed, LibVMI only supports virtualized environments, and "Software RDMI" is an approximation of RDMI that relegates introspection logic to remote server software (but still uses RDMA to fetch kernel memory).

Introspection time. Figure 6 shows the time it takes to complete an introspection policy across the three defenses—RDMI executes 9-58 times faster than the state-of-the-art LibVMI solution. "Software RDMI," as an approximation, is also much faster than LibVMI but it still falls behind the full RDMI, which outperforms the former by roughly two times. This is not only because "Software RDMI" still involves remote CPU overheads, but also that the introspection and introspected machines are necessarily located farther away from each other, connected by an intermediate switch. RDMI, on the other hand, is a switch-resident defense and has an immediate reach to all servers under the same rack. The introspection time also varies across policies—e.g., P3 is the fastest (15.5μs) and P7 is the slowest (150.1ms) for RDMI.

We further measure the capture rates across the three defenses by introducing a "cat-and-mouse" game, where a kernel rootkit rapidly modifies the credential data structures of specific processes and then modifies them back. By setting the attacker to different modification frequencies, we compare how well the defenses can capture the modifications by measuring their capture rates. Figure 7 shows the results for up to 500 modifications per second, with RDMI consistently achieving the highest capture rates. When the attack goes beyond 50 modifications per second, LibVMI drops to about 20% capture rate; when it increases to 200 modifications per second, even Software RDMI drops to 57%—but at both frequencies RDMI stays at 100% and it only drops for much faster attacks.

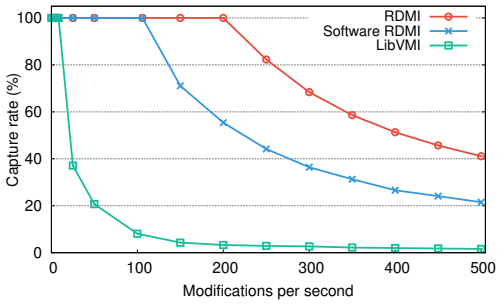


Figure 7: Capture rates of the three defenses.

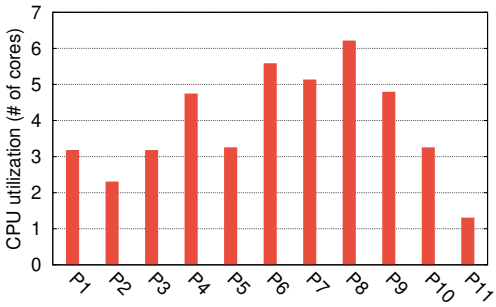


Figure 8: CPU overheads for security policies with software RDMI under the same introspection throughput as LibVMI.

Introspection CPU overheads. As motivated before, hypervisor offloading aims to reduce software CPU overheads from the servers. RDMI achieves introspection entirely in programmable hardware, without CPU software overheads. Thus, we measure the CPU overhead for the other defenses to understand the cost for security. We configure LibVMI to introspect varying numbers of VMs with each of the policies, and measure the number of CPU cores that are required for the introspection tasks. Each KVMI introspection socket is

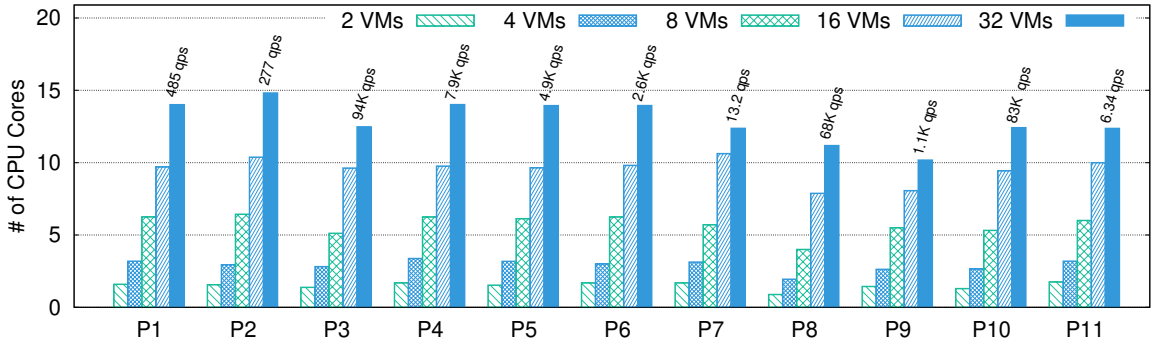
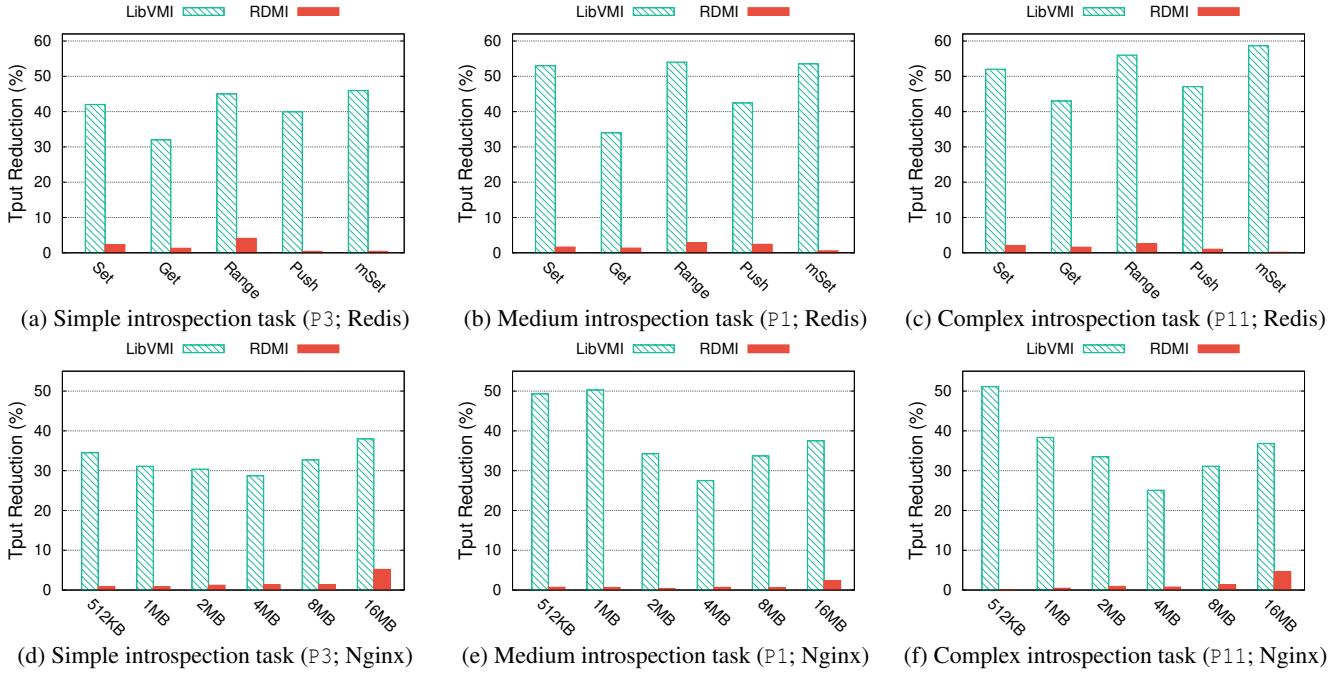


Figure 9: CPU costs (# of cores) of LibVMI introspection for different policies with varying numbers of VMs. The query speeds (i.e., introspection queries per second) for 32 VMs are labeled on the top of the bars.



(a) Simple introspection task (P3; Redis)
(b) Medium introspection task (P1; Redis)
(c) Complex introspection task (P11; Redis)
(d) Simple introspection task (P3; Nginx)
(e) Medium introspection task (P1; Nginx)
(f) Complex introspection task (P11; Nginx)

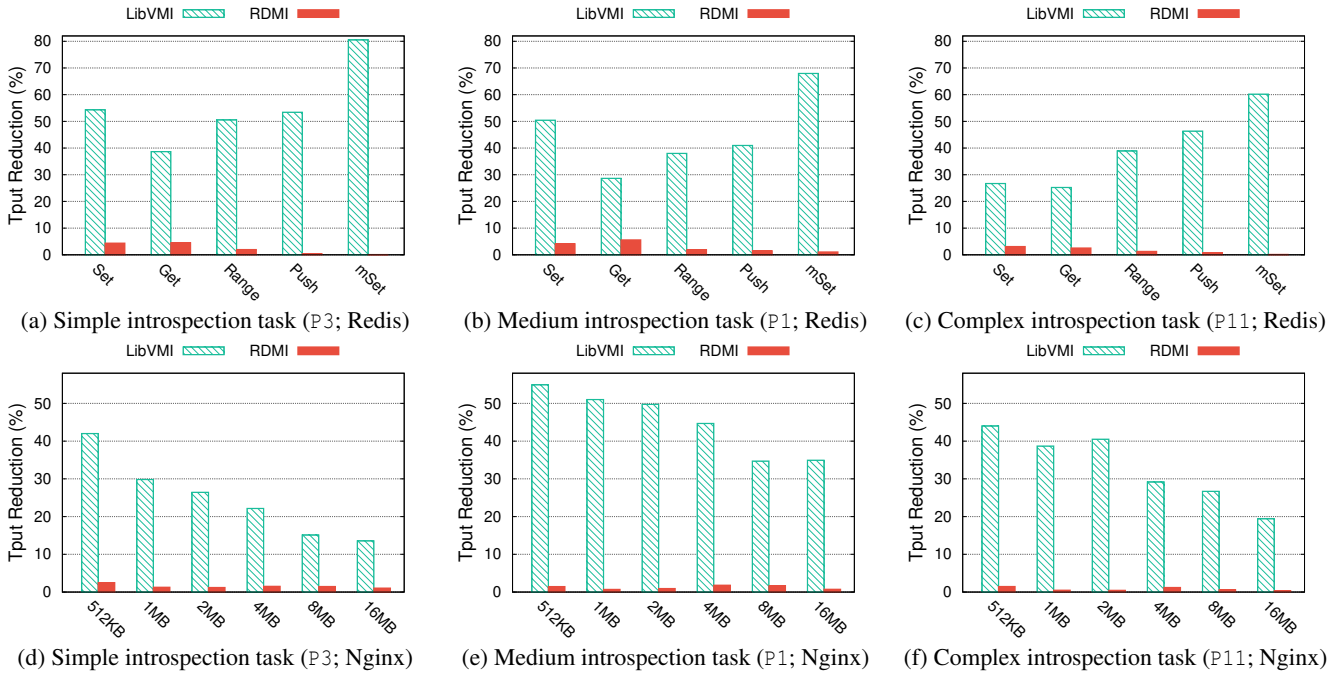
Figure 10: Throughput reduction of local Redis and Nginx with LibVMI and RDMI, normalized to their respective baselines.

attached to a single introspection program, which is dedicated to a policy and executes it repeatedly. As shown in Figure 9, for 32 VMs, LibVMI requires 10-14 CPU cores (out of 24 cores) just for the introspection tasks. This is a significant cost, as the resulting CPU overheads take away valuable cycles that are no longer provisioned for the tenants. Next, we configure Software RDMI to execute at the same query speeds that are achievable by LibVMI with 32 VMs, and measure the CPU costs of the introspection machine. As Figure 8 shows, the CPU cost is significantly lower—roughly on par with what would be needed to introspect 4-8 VMs with the LibVMI solution. Nevertheless, Software RDMI still requires 1.3-6.2 CPU cores to drive the introspection task, whereas RDMI removes the CPU overhead entirely by executing in Dom(-1).

Summary. Concretely, the performance gain of RDMI comes from two factors. First, RDMI executes entirely in programmable ASICs at hardware speeds both for memory retrieval and for introspection computation, whereas the hypervisor solutions execute on CPU software with high CPU overhead. Moreover, LibVMI requires the presence of a hypervisor, and as shown in §7.5, this by itself incurs a large footprint and takes away resources from any colocated tasks. In contrast, RDMI executes in a baremetal setting off the host, so introspection tasks and tenant workloads cause minimal interference to each other. In addition, our "cat-and-mouse" experiment shows that this performance gain translates to concrete security benefits in capture rates.

7.5 Introspection interference

We have seen that introspection is a heavyweight task with high CPU overheads. Next, we quantify the introspection interference to tenant workloads. For LibVMI, we use the 32-VM setting where one of the VMs is executing tenant services, and measure the performance overhead to these workloads



(a) Simple introspection task (`P3`; Redis) (b) Medium introspection task (`P1`; Redis) (c) Complex introspection task (`P11`; Redis)

(d) Simple introspection task (`P3`; Nginx) (e) Medium introspection task (`P1`; Nginx) (f) Complex introspection task (`P11`; Nginx)

Figure 11: Throughput reduction of remote Redis and Nginx with LibVMI and RDMI.

with hypervisor introspection. For RDMI, we measure the same workloads on a baremetal machine and quantify the performance downgrade of the introspected machine. We omit Software RDMI from this measurement, as its overhead on the *introspected* machine is the same with RDMI—the downside of Software RDMI comes from overheads of the remote *introspection* machine, which is dedicated to introspection tasks and does not run tenant workloads. We use two common workloads—key/value operations (Redis) and web transfers (Nginx)—and test them both locally and via the network.

Redis key/value workloads (local). Figure 10 shows the throughput reduction of key/value workloads, with the Redis client and server colocated on the same machine, and using varying key/value operations common for Redis benchmarking (i.e., Set, Get, Range-100, Push, mSet). For fairness of comparison, we normalize the LibVMI-enabled throughput against the Redis throughput without LibVMI (both in VMs), and RDMI-enabled throughput against the case without RDMI (both in baremetal servers). Further, we choose three representative introspection policies based on their complexity in the LibVMI implementations (simple: P3, medium: P1, complex: P11). We can see that RDMI incurs 0.1%-4% throughput overhead across all key/value workloads, whereas LibVMI is 11-486 times higher with reductions ranging from 22%-56%. This is because LibVMI introspection incurs high CPU overheads, and creates severe contention. RDMI, on the other hand, does not involve the remote CPUs. Its overhead comes from the RDMA reads that are converted to memory accesses, which incur a small amount of memory contention.

Web server workloads (local). Using a similar methodology, we have measured the introspection interference of LibVMI and RDMI with Apache web server workloads. In this experiment, we generate HTTP requests to download files of varying sizes from the web server, and measure the throughput of requests served per second. As Figure 10 shows, RDMI only introduces 0.4%-5.1% throughput overheads across all workloads, whereas LibVMI incurs an overhead ranging from 25%-51%, which is 7-711 times higher.

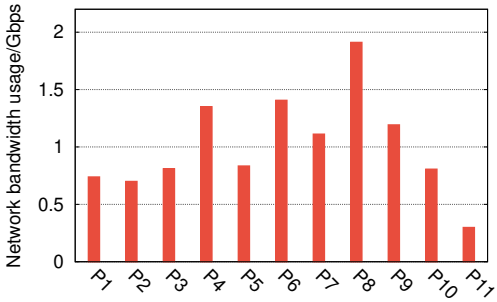


Figure 12: Bandwidth consumption for RDMI.

Remote workloads. Next, we measure the same workloads when the requests are coming through the network. Whereas the local experiments are designed to measure CPU and memory overheads due to introspection interference, this setup additionally accounts for the impact of network traffic as generated by RDMI. LibVMI only performs local operations, so it does not incur any network IO. Figure 11 measures the Redis and Nginx throughputs over the network, respectively. Even accounting for network overheads, RDMI only incurs a throughput degradation between 0.1%-5.6%. In comparison, when LibVMI is handling remote client requests, the degradation ranges from 20%-60% across the workloads, which is

5-864 times higher. Therefore, these results show that, with requests coming through the network, RDMI is still able to perform security tasks with minimal performance interference, unlike state-of-the-art hypervisor solutions.

Figure 12 further shows the amount of network bandwidth overhead under different introspection policies. We set RDMI to introspect the baremetal machine at the same speed (in terms of queries per second) as what LibVMI can achieve at its peak throughput with 32 VMs locally. Across all queries, the network bandwidths due to remote introspection range from 0.3–1.91Gbps; over a 100Gbps link, this translates to 0.3%-1.91% network overheads.

8 Discussions

Cost of deploying RDMI. Since RDMA NICs and programmable switches have been in use at major cloud providers [44, 47, 60, 78], we believe that the barrier to deploying RDMI is reasonably low. Nevertheless, for a cloud provider that does not already use RDMA NICs and programmable switches, there will be an extra cost for upgrading these devices. We provide some data points for understanding the CapEx and OpEx cost based on available prices. Our programmable switch costs $10,060 [41], and a non-programmable switch operating at the same port speeds (32x100Gbps) costs $9,399 [11]. Our RDMA NIC costs $488 [30], and a non-RDMA NIC at the same speed costs $355 [18]. The extra cost for upgrading a single switch and a single RDMA NIC is therefore $794. As we have shown in §7.4, hypervisor-based introspection requires 11.2–14 CPU cores on average with 32 VMs across policies P1-P11. These core counts are similar to what is provisioned in Amazon's m5zn.3xlarge EC2 instances, sold at $0.991/hour [3, 6]; hence, RDMI would be more cost-effective after 33.4 days of operation. Although device and VM costs change over time and across vendors, we believe that this back-of-the-envelope calculation paints a representative picture. Also, we note again that cloud providers that already invest in these devices would not incur additional capital cost.

Attacks to RDMA and P4 systems. RDMI NICs and P4 programmable switches are part of our TCB, and they are assumed to be trustworthy. However, existing work has identified security issues with both types of devices [71, 96]. As some examples, adversaries could launch side channel [100], exfiltration, injection, and denial-of-service attacks [96] to RDMA deployments. P4 programmable devices may also exhibit corner-case behaviors under carefully crafted traffic patterns by the attacker [71]. However, these generic attacks are not specific to RDMI, and known defenses exist [105].

Cache coherence, consistency, registers. RDMI performs introspection in an out-of-band (OOB) manner, and OOB introspection [90] has several limitations shared with RDMI. *(i) Cache coherence:* RDMA memory accesses are not cache coherent with host CPUs unless more advanced interconnects (e.g., CXL [12]) become available. Thus, when RDMI acquires a data structure from the main memory, it is not guaranteed to be the latest version as modified data may exist in the CPU cache. *(ii) Consistency:* Kernel state is in constant flux, and this leads to another degree of asynchrony between RDMI's view and the true system state. For instance, while RDMI traverses the `task_struct` linked list, processes could be added to or removed from the data structure and these changes may not be reflected in RDMI's view. In fact, even for hypervisor-based solutions, inconsistent views could arise unless guest VMs are paused during introspection, but this would lead to significant overheads. *(iii) Registers:* OOB solutions cannot introspect CPU state such as register values [76, 90, 98]. Previous work [66] has shown that advanced attackers could manipulate the CR3 register so that the actual page tables used by the OS are different from those seen by OOB introspection. Despite these limitations, OOB solutions have been shown to be effective in existing work [90], and RDMI corroborates these findings. Importantly, in baremetal settings, the host machine does not have a hypervisor to perform in-band introspection, so OOB solutions like RDMI are necessary in order to protect baremetal kernels.

Introspection capability. Our current RDMI experiments disable the IOMMU, but when it is enabled, DMA requests from PCIe devices may be further translated by the IOMMU. In this case, an attacker could create incorrect IOMMU mappings to confound security mechanisms like RDMI [45, 98]. As a potential mitigation, PCIe devices that implement the ATS (Address Translation Service) feature can tag DMA requests so that they are not further translated by the IOMMU, thus ensuring trusted memory acquisition [45]. Recent RNICs have been built with ATS features [31]. In terms of introspection tasks, RDMI supports tasks that traverse the kernel graph in a well-defined footprint to detect attacks. However, not all memory forensic tasks fall into this category—e.g., performing a cryptographic hash over kernel text to ensure integrity [90, 91], or regular expression matches over memory content [51, 56, 99], would go beyond RDMI's current DSL and hardware capabilities. With an imperative language (e.g., C programs that use LibVMI), one could also write introspection tasks that walk kernel pointers in arbitrary patterns; such tasks also create challenges for the current DSL. Nevertheless, we have demonstrated that RDMI is sufficiently expressive for a range of tasks and can detect real-world rootkits.

9 Related work

Memory introspection. The art of introspecting memory snapshots to detect malice dates back to two decades ago [61]. Since then, many techniques have been developed to improve the accuracy of kernel memory analysis [21, 48, 49, 52, 54, 57, 79, 91, 93, 98, 103] and narrow the semantic gap [43, 53, 62, 86]. Hypervisor-based systems, such as ImEE [110] and livewire [61], use software solutions to introspect guest VMs. Our project is, in particular, related to out-of-band (OOB) introspection techniques that leverage hardware

assistance. In this space, KI-Mon [76] and Vigilare [85] add a special security module that snoops the memory bus and detects kernel object modifications. Copilot [90] contributes a system prototype for an Intel StrongARM EBSA-285 evaluation board that can acquire memory over PCIe. In contrast to existing OOB platforms, RDMI a) only uses COTS devices for baremetal introspection; b) it contributes a domain-specific language, compiler, and runtime for introspection tasks, which c) can be executed efficiently in programmable hardware without CPU involvement.

P4 and RDMA. P4 and RDMA programmable devices have been used for performance acceleration in cloud systems—separately [68, 70] and in conjunction [73, 75]. The security community has developed a line of work using P4 programmable switches for network protection [83, 106], including for RDMA vulnerabilities [105]. RDMI demonstrates their use in a novel setting for kernel security.

10 Conclusion

Hypervisor offloading has gained popularity in datacenters. We rearchitect a classic security task that is usually relegated to the hypervisor—memory introspection—to enable introspection of baremetal servers entirely in programmable ASICs. RDMI leverages recent hardware advances in RDMA NICs and P4 programmable switches, and designs a domain-specific language, compiler, and runtime system. RDMI incurs no CPU overheads in introspection tasks, outperforming state-of-the-art hypervisor-based solutions, and detects a variety of rootkits even in baremetal installations.

Acknowledgments: We thank our anonymous reviewers and shepherd for their insightful feedback. We also thank Yiming Qiu, Kuo-Feng Hsu, and Haggai Eran for their valuable comments on earlier drafts. This work was supported in part by NSF grants CNS-1942219, CNS-1955270, CNS-2106751, CNS-2106388, CNS-2115587, CNS-2214272, a Google PhD Fellowship, and a VMware Early Career Faculty Grant.

References

[1] Alibaba cloud X-Dragon NIC. https://www.alibabacloud.com/blog/introducing-the-sixth-generation-of-alibaba-clouds-elastic-compute-service_595716.

[2] Alibaba ESC baremetal instance. https://www.alibabacloud.com/product/ebm.

[3] Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[4] AMD Pensando Infrastructure Accelerators. https://www.amd.com/en/accelerators/pensando.

[5] Average Coder Rootkit. https://volatility-labs.blogspot.com/2012/09/movp-14-average-coder-rootkit-bash.html.

[6] AWS EC2 m5zn.3xlarge instance information. https://instances.vantage.sh/aws/ec2/m5zn.3xlarge.

[7] AWS new baremetal cloud service. https://aws.amazon.com/about-aws/whats-new/2021/02/introducing-amazon-ec2-m5n-m5dn-r5n-and-r5dn-bare-metal-instances/.

[8] AWS Nitro card. https://aws.amazon.com/ec2/nitro/.

[9] Azure dedicated host. https://azure.microsoft.com/en-us/services/virtual-machines/dedicated-host/.

[10] Bare Metal Solution for Oracle. https://cloud.google.com/bare-metal.

[11] Broadcom N8560-32C, 32-Port Ethernet switch. https://www.fs.com/products/110480.html.

[12] Compute Express Link. https://www.computeexpresslink.org.

[13] DPDK Project. https://www.dpdk.org.

[14] Fungible DPU. https://www.fungible.com.

[15] Google C3 machine series. https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu.

[16] IBM baremetal cloud service. https://www.ibm.com/uk-en/cloud/bare-metal-servers.

[17] Intel Corporation. Intel(R) 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[18] Intel Ethernet Network Adapter XXV710-DA2. https://www.intel.com/content/www/us/en/products/sku/97303/intel-ethernet-network-adapter-xxv710da2-for-ocp/specifications.html.

[19] Intel IPU Based Cloud Infrastructure White Paper . https://www.intel.com/content/www/us/en/products/docs/programmable/ipu-based-cloud-infrastructure-white-paper.html.

[20] Intel virtualization technology. https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html.

[21] KNTLIST tool. http://old.dfrws.org/2005/challenge/kntlist.shtml.

[22] libVMI. https://libvmi.com.

[23] Linux Adore-ng rootkit. https://github.com/yaoyumeng/adore-ng.

[24] Linux Diamorphine rootkit. https://github.com/m0nad/Diamorphine.

[25] Linux kernel backdoors and their detection. https://slidetodoc.com/linux-kernel-backdoors-and-their-detection-joanna-rutkowska/.

[26] Linux netfilter project. https://www.netfilter.org.

[27] Linux Spy rootkit. https://github.com/jarun/spy.

[28] Linux Sutekh rootkit. https://github.com/PinkP4nther/Sutekh.

[29] Netronome Agilio SmartNICs firmware. https://github.com/Netronome/nic-firmware/.

[30] Nvidia ConnectX-4 adapter card. https://store.nvidia.com/en-us/networking/store/?page=1&limit=9&locale=en-us&category=ADAPTER_CARDS&gpu=ConnectX-4LxEN.

[31] Nvidia ConnectX-6 ATS feature. https://docs.nvidia.com/ai-enterprise/deployment-guide-multi-node/0.1.0/getting-started.html#enable-ats-on-the-nvidia-connectx-6-dx-nic.

[32] Port knocking. https://en.wikipedia.org/wiki/Port_knocking.

[33] Pure storage Baremetal-as-a-service. https://www.purestorage.com/products/staas/baremetal-as-a-service.html.

[34] RDMI code repository. https://github.com/aladinggit/RDMI.

[35] SPDK Project. https://spdk.io.

[36] Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.1. https://cw.infinibandta.org/document/dl/7781.

[37] The Security Design of the AWS Nitro System. https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.html.

[38] Trusted computing boot. https://www.amd.com/system/files/2017-06/Trusting-in-the-CPU.pdf.

[39] VMware Monterey project. https://blogs.vmware.com/vsphere/2020/09/announcing-project-monterey-redefining-hybrid-cloud-architecture.html.

[40] Volatility–Process list walking. https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/linux/pslist.py.

[41] Wedge 100BF-32X Tofino programmable switch. https://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf.

[42] E. Amaro, Z. Luo, A. Ousterhout, A. Krishnamurthy, A. Panda, S. Ratnasamy, and S. Shenker. Remote memory calls. In *Proc. HotNets*, 2020.

[43] N. Amit and M. Wei. The design and implementation of hyperupcalls. In *Proc. ATC*, 2018.

[44] M. Arumugam, D. Bansal, N. Bhatia, J. Boerner, S. Capper, C. Kim, S. McClure, N. Motwani, R. Narasimhan, U. Panchal, et al. Bluebird: High-performance SDN for bare-metal cloud services. In *Proc. NSDI*, 2022.

[45] A. Atamli, G. Petracca, and J. Crowcroft. IO-Trust: an out-of-band trusted memory acquisition for intrusion detection and forensics investigations in cloud IOMMU based systems. In *Proc. ARES*, 2019.

[46] J. Bae, L. Liu, Y. Wu, G. Su, and A. Iyengar. Rdmabox: Optimizing rdma for memory intensive workload. In *Proc. IEEE CIC*, 2021.

[47] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema, et al. Empowering Azure storage with RDMA. In *Proc. NSDI*, 2023.

[48] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proc. IEEE ACSAC*, 2008.

[49] E. Bauman, G. Ayoade, and Z. Lin. A survey on hypervisor-based monitoring: approaches, applications, and evolutions. *ACM Computing Surveys (CSUR)*, 2015.

[50] S. L. Blond, D. Choffnes, W. Caldwell, P. Druschel, and N. Merritt. Herd: A scalable, traffic analysis resistant anonymity network for voip systems. In *Proc. SIGCOMM*, 2014.

[51] M. Botacin, F. B. Moreira, P. O. Navaux, A. Grégio, and M. A. Alves. Terminator: A secure coprocessor to accelerate real-time antiviruses using inspection breakpoints. *ACM Transactions on Privacy and Security*, 2022.

[52] C. Bugcheck. Grepexec: Grepping executive objects from pool memory. In *Proc. Digital Forensic Research Workshop*, 2006.

[53] M. Carbone, M. Conover, B. Montague, and W. Lee. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Proc. RAID*, 2012.

[54] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proc. CCS*, 2009.

[55] J. Chen, D. Li, Z. Mi, Y. Liu, B. Zang, H. Guan, and H. Chen. Duvisor: a user-level hypervisor through delegated virtualization. 2022. arXiv preprint arXiv:2201.09652.

[56] A. Costin and J. Zaddach. IoT malware: Comprehensive survey, analysis framework and case studies. *BlackHat USA*, 1(1):1–9, 2018.

[57] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In *Proc. USENIX Security*, 2012.

[58] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *Proc. NSDI*, 2014.

[59] D. Dumitrescu, R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Dataplane equivalence and its applications. In *Proc. NSDI*, 2019.

[60] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, et al. When cloud storage meets rdma. In *NSDI*, 2021.

[61] T. Garfinkel, M. Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, 2003.

[62] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *Proc. IEEE SRDS*, 2011.

[63] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *Proc. SIGCOMM*, 2018.

[64] O. Hofmann, A. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *Proc. ASPLOS*, 2011.

[65] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion. SoK: Introspections on trust and the semantic gap. In *IEEE Symposium on Security and Privacy*, 2014.

[66] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. B. Kang. Atra: Address translation redirection attack against hardware-based external monitors. In *Proc. CCS*, 2014.

[67] X. Jiang, M. Lora, and S. Chattopadhyay. Efficient and trusted detection of rootkit in IoT devices via offline profiling and online monitoring. In *Proc. VLSI*, 2020.

[68] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proc. SOSP*, 2017.

[69] J. Junnila. Effectiveness of linux rootkit detection tools. 2020.

[70] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be general and fast. In *Proc. NSDI*, 2019.

[71] Q. Kang, J. Xing, Y. Qiu, and A. Chen. Probabilistic profiling of stateful data planes for adversarial testing. In *Proc. ASPLOS*, 2021.

[72] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo. Programmable in-network security for context-aware BYOD policies. In *Proc. USENIX Security*, 2020.

[73] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. TEA: Enabling state-intensive network functions on programmable switches. In *Proc. SIGCOMM*, 2020.

[74] P. Krishnamurthy, H. Salehghaffari, S. Duraisamy, R. Karri, and F. Khorrami. Stealthy rootkits in smart grid controllers. In *Proc. IEEE ICCD*, 2019.

[75] J. Langlet, R. B. Basat, S. Ramanathan, G. Oliaro, M. Mitzenmacher, M. Yu, and G. Antichi. Zero-CPU collection with direct telemetry access. In *Proc. HotNets*, 2021.

[76] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang. KI-Mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Proc. USENIX Security*, 2013.

[77] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee. MIND: In-network memory management for disaggregated data centers. In *Proc. SOSP*, 2021.

[78] Y. Li, J. Gao, E. Zhai, M. Liu, K. Liu, and H. H. Liu. Cetus: Releasing P4 programmers from the chore of trial and error compiling. In *Proc. NSDI*, 2022.

[79] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS*, 2011.

[80] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, C. C. Robert Soulé, Han Wang, N. McKeown, and N. Foster. p4v: Practical verification for programmable data planes. In *Proc. SIGCOMM*, 2018.

[81] D. Loehr and D. Walker. Safe, modular packet pipeline programming. In *Proc. POPL*, 2022.

[82] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci. Taming performance variability. In *Proc. OSDI*, 2018.

[83] R. Meier, P. Tsankov, V. Lenders, L. Vanbever, and M. Vechev. NetHide: Secure and practical network topology obfuscation. In *Proc. USENIX Security*, 2018.

[84] P. Mishra, I. Verma, S. Gupta, V. S. Rana, and K. Kadarla. vproval: Introspection based process validation for detecting malware in KVM-based cloud environment. In *Proc. FMEC*, 2019.

[85] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: toward snoop-based kernel integrity monitor. In *Proc. CCS*, 2012.

[86] A. More and S. Tapaswi. Virtual machine introspection: towards bridging the semantic gap. *Journal of Cloud Computing*, 2014.

[87] J. Navarro, E. Naudon, and D. Oliveira. Bridging the semantic gap to mitigate kernel-level keyloggers. In *2012 IEEE Symposium on Security and Privacy Workshops*, 2012.

[88] S. Novakovic, Y. Shan, A. Kolli, M. Cui, Y. Zhang, H. Eran, B. Pismenny, L. Liss, M. Wei, D. Tsafrir, et al. Storm: a fast transactional dataplane for remote data structures. In *Proc. SYSTOR*, 2019.

[89] F. Pagani and D. Balzarotti. Back to the whiteboard: A principled approach for the assessment and design of memory forensic techniques. In *Proc. USENIX Security*, 2019.

[90] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot: A coprocessor-based kernel runtime integrity monitor. In *Proc. USENIX Security*, 2004.

[91] N. L. Petroni Jr and M. Hicks. Automated detection of persistent

kernel control-flow attacks. In *Proc. CCS*, 2007.
[92] D.-P. Pham, D. Marion, and A. Heuser. ULTRA: Ultimate rootkit detection over the air. In *Proc. RAID*, 2022.
[93] J. Rhee, R. Riley, D. Xu, and X. Jiang. LiveDM: kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *Proc. Annual Information Security Symposium*, 2011.
[94] R. Riley, X. Jiang, and D. Xu. Multi-aspect profiling of kernel rootkit behavior. In *Proc. EuroSys*, 2009.
[95] M. A. Rodriguez. The Gremlin Graph Traversal Machine and Language. In *Proc. DBPL*, 2015.
[96] B. Rothenberger, K. Taranov, A. Perrig, and T. Hoefler. ReDMArk: Bypassing rdma security mechanisms. In *Proc. USENIX Security*, 2021.
[97] J. Sonchack, D. Loehr, J. Rexford, and D. Walker. Lucid: A language for control in the data plane. In *Proc. SIGCOMM*, 2021.
[98] C. Spensky, H. Hu, and K. Leach. LO-PHI: Low-observable physical host instrumentation for malware analysis. In *Proc. NDSS*, 2016.
[99] F. Tchakounté, R. C. N. Ngassi, V. C. Kamla, and K. P. Udagepola. LimonDroid: a system coupling three signature-based schemes for profiling android malware. *Iran Journal of Computer Science*, 2021.
[100] S.-Y. Tsai, M. Payer, and Y. Zhang. Pythia: Remote oracles for the masses. In *USENIX Security Symposium*, 2019.
[101] S.-Y. Tsai and Y. Zhang. LITE kernel RDMA support for datacenter applications. In *Proc. SOSP*, 2017.
[102] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Proc. USENIX Security*, 2019.
[103] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proc. CCS*, 2009.
[104] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen. Runtime programmable switches. In *Proc. NSDI*, 2022.
[105] J. Xing, K.-F. Hsu, Y. Qiu, Z. Yang, H. Liu, and A. Chen. Bedrock: Programmable network support for secure RDMA systems. In *Proc. USENIX Security*, 2022.
[106] J. Xing, Q. Kang, and A. Chen. Netwarden: Mitigating network covert channels while preserving performance. In *Proc. USENIX Security*, 2020.
[107] J. Xing, Y. Qiu, K.-F. Hsu, H. Liu, M. Kadosh, A. Lo, A. Akella, T. Anderson, A. Krishnamurthy, T. S. E. Ng, and A. Chen. A vision for runtime programmable networks. In *Proc. HotNets*, 2021.
[108] J. Xing, W. Wu, and A. Chen. Ripple: A programmable, decentralized link-flooding defense against adaptive adversaries. In *Proc. USENIX Security*, 2021.
[109] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proc. SOSP*, 2011.
[110] S. Zhao, X. Ding, W. Xu, and D. Gu. Seeing through the same lens: introspecting guest address space at native speed. In *Proc. USENIX Security*, 2017.

11 Appendix

11.1 Compiling the iterate operator

We include the AIM instructions and placeholders that are compiled from an `iterate` operator:

```
    Load(R1, $n) // initialize loop bound w/ counter.
    Mov($array_addr) //move base to first elem.
L:
    Push   //for potential nested loops
    /*
        Loop body placeholder, compiled from
        inner operators, same as in 'traverse'
    */
    Pop    //returning from inner loop
    Mov($nxt_entry) //Move to next array element
    Jmp(R1--, L, Lend) // loop guard
Lend: //Iterate completes
```

11.2 RDMI policies

In the main paper, we have already presented four RDMI tasks in detail. Here, we include the RDMI code for the remaining seven policies and describe their introspection goals.

P5: TTY keylogger checks [87]. Keyloggers are a class of malware that secretly records user keystrokes, usually by hooking themselves onto the input handlers of tty devices. The Linux kernel maintains a `struct tty_driver` linked list, each of which represents a specific device driver. Further, each `struct tty_driver` can be attached with a set of devices, so an array of pointers at `struct tty_struct *ttys` maintains this information. When a device is opened, a new `struct tty_struct *` element will be added in this array. For each attached device, its `struct tty_struct` contains a pointer to the external "line discipline" `ldisc` data structure, which serves as the glue between device drivers and high-level interface calls (e.g., read, write). It further contains a `receive_buffer` function pointer, which is a common hook point for keyloggers [87]. The type annotation '@' is used for handling generic list_head structs where they can be embedded in a range of kernel data structures—following existing work that proposed similar annotation methods [91]. The keyword '`this`' refers to the current introspection address.

```
kgraph(tty_drivers)
// type annotation @ for 'in' to handle list_head
.in(next, @struct tty_driver, @tty_drivers)
.traverse(tty_drivers.next, &tty_drivers.next,
    tty_driver)
.values(num)
.in(ttys)
// iterate tty_struct pointer array
.iterate(this, num, ptr_t)
.in(this).in(ldisc).in(ops).values(receive_buf)
.assert(KERN_TXT_BEGIN < receive_buf < KERN_TXT_END)
```

P6: System call checks. This policy starts with the global kernel symbol `sys_call_table` and iterates through all exported system calls, where SYSCALL_NR denotes the number of entries. Further, it asserts that these system call pointers

must lie within a well-defined range. The keyword `this` is implicitly filled in by the compiler using the current introspection address at that point of the traversal.

```
1  kgraph(sys_call_table)
2  .iterate(this, SYSCALL_NR, ptr_t)
3  .values(this)
4  .assert(KERN_TXT_BEGIN < this < KERN_TXT_END)
```

P7: Process memory map check. This query checks the virtual memory area information of each process. It performs a nested traversal over two linked lists at Lines 2+8. Similar to P1 and P2, this query begins with an outer traversal that visits the linked list located at `init_task`. For each `task_struct`, the policy zooms `in` on the `mm` and `mmap` data structures that hang off of the main linked list. It then further performs an inner linked list traversal which goes through each `vm_area_struct`, where further details such as VMA addresses and access permissions are stored.

```
1  kgraph(init_task)
2  .traverse(tasks.next, &init_task.tasks, task_struct)
3  .values(pid)
4  .in(mm)
5  .in(mmap)
6  // Traverse the VMA linked list utill a NULL pointer
7  .traverse(vm_next, NULL, vm_area_struct)
8  .values(vm_start, vm_end, vm_page_prot)
```

P8: Keyboard sniffer checks [87]. Keyboard sniffers eavesdrop on keystrokes from user keyboards, similar to P6 but hooked into the system at different locations. This query examines the `notifier_block` registered for `keyboard_notifier_list` linked list. It traverses the linked list until the next pointer is null. For each element in the traversal, RDMI checks if the `notifier_call` pointer is within a known-good range to detect malicious sniffing behaviors.

```
1  kgraph(keyboard_notifier_list)
2  .in(head)
3  .traverse(next, NULL, notifier_block)
4  .values(notifier_call)
5  .assert(KERN_TXT_BEGIN < notifier_call < KERN_TXT_END)
```

P9: Module list traversal. This query starts with the global kernel symbol `modules` and further traverses the module list and analyzes the loaded kernel modules. Similar to global symbol `tty_drivers` in P6, `modules` is a `list_head` data structure requiring type annotation. Then, by following the `next` pointer inside each `struct module`, the module list can be traversed until the starting point has been reached again.

```
1  kgraph(modules)
2  .in(next, @struct module, @list)
3  .traverse(list.next, &modules.next, module)
4  .values(name)
```

P10: Afinfo operation checks. `tcp_seq_afinfo` operations are important for administrative utilities such as netstats that list socket activities. By hijacking such operations, rootkits can hide open ports and connections from malicious processes. This query first checks the `seq_ops` inside the `tcp4_seq_afinfo` for integrity validation. Then, it zooms `in` on the `file_operations` data structure and checks the `open` operation contained within this data structure.

```
1  kgraph(tcp4_seq_afinfo)
2  .values(seq_ops.show)
3  .assert(KERN_TEXT_BEGIN < show < KERN_TEXT_END)
4  .in(seq_fops)
5  .values(open)
6  .assert(KERN_TEXT_BEGIN < open < KERN_TEXT_END)
```

P11: Open file list. This query aims to check all files opened by each process. Similar to previous process related introspection tasks, this query starts with a `task_struct` traversal in the outer loop. It then zooms `in` by several layers eventually reaching the file descriptor table (`struct fdtable`) related data structure, where it fetches the number of entries inside dynamically allocated `fd` array and iterates each entry inside the array. Each entry is a pointer that points to a `struct file` data structure. Line 10 further fetches the `f_path.dentry` pointer and Line 11 acquires the file names.

```
1  kgraph(init_task)
2  .traverse(tasks.next, &init_task.tasks, task_struct)
3  .values(pid)
4  .in(files)
5  .in(fdt)
6  .values(max_fds)
7  .in(fd)
8  .iterate(this, max_fds, ptr_t)
9  .in(this)
10 .in(f_path.dentry)
11 .values(d_iname)
```

Kernel variables to offsets. To parse kernel variables (e.g., `int pid`) into their offsets from the starting address of their containing data structure (e.g., `struct task_struct`), we rely on an automated translation process inside the compiler that is integrated with `.json` database that RDMI has curated for a kernel version. This curating process is hidden within the RDMI compiler as well, shielded from RDMI users. Our current database supports the kernel variables in Linux `v4.15`, and adding support for more kernel versions is a mechanical process and is easily achievable with more engineering efforts.