# Bedrock: Programmable Network Support for Secure RDMA Systems

Jiarong Xing Kuo-Feng Hsu Yiming Qiu Ziyang Yang Hongyi Liu Ang Chen
*Rice University*

## Abstract

Remote direct memory access (RDMA) has gained popularity in cloud datacenters. In RDMA, clients bypass server CPUs and directly read/write remote memory. Recent findings have highlighted a host of vulnerabilities with RDMA, which give rise to attacks such as packet injection, denial of service, and side channel leakage, but RDMA defenses are still lagging behind. As the RDMA datapath bypasses CPU-based software processing, traditional defenses cannot be easily inserted without incurring performance penalty. Bedrock develops a security foundation for RDMA inside the network, leveraging programmable data planes in modern network hardware. It designs a range of defense primitives, including source authentication, access control, as well as monitoring and logging, to address RDMA-based attacks. Bedrock does not incur software overhead to the critical datapath, and delivers native RDMA performance in data transfers. Moreover, Bedrock operates transparently to legacy RDMA systems, without requiring RNIC, OS, or RDMA library changes. We present a comprehensive set of experiments on Bedrock and demonstrate its effectiveness.

## 1 Introduction

Remote direct memory access (RDMA), a technology that originates in high-performance computing (HPC), has gained popularity in modern cloud datacenters [1, 2]. In RDMA systems, servers expose a "remote memory" abstraction to networked clients, offering high throughput and low latency. Bypassing server CPUs, one-sided RDMA operations (e.g., READ and WRITE) enable remote accesses at hardware speeds. This is achieved by the use of RDMA-enabled network interface cards (RNICs), which enclose special ASICs for translating RDMA operations to local DMA requests to the main memory over PCIe. Only control operations like connection setup and teardown require CPU-based software intervention. The datapath itself is free of software bottlenecks and enables low-latency remote memory access.

The expansion from private HPC environments to public, multi-tenant clouds, however, has put RDMA security under greater scrutiny. Exposing server memory to remote clients without CPU mediation comes with a host of security implications. Recent work has demonstrated that RDMA systems lack secure mechanisms for authentication [46, 52]; that they have rigid access control mechanisms that are easy to bypass [46, 49]; and that they cannot produce audit trails for forensics [46, 49]. This has culminated in a systematic study on RDMA security problems under varied threat models in ReDMArk [46]. The authors have considered varied threat models and identified scenarios in which attackers can inject spoofed RDMA packets, bypass access control mechanisms, launch various types of denial-of-service attacks, and leak data via side channels. Many of the vulnerabilities are deeply rooted in the insecure RNIC hardware designs, therefore fundamental to today's RDMA systems. While defense analogues exist for traditional TCP/IP networks (e.g., against IP spoofing or TCP injection attacks), RDMA systems present a distinct challenge in bypassing the CPU. The key to RDMA performance lies in the exclusion of CPU software processing. This creates significant difficulty in using existing software-based defenses without negating the performance benefits afforded by RDMA.

The key research goal in this paper is to develop a suite of defenses that are compatible with the CPU-bypassing paradigm in RDMA systems, therefore preserving their raw performance, but without requiring changes to RNIC hardware, OS, or RDMA libraries. Bedrock secures the datapath traffic directly inside the network, relying on advances in networking technology that developed programmable data planes in switches and NICs. With this technology, it becomes feasible to develop datapath defenses that operate at hardware speeds. We can further programmatically insert them underneath the RDMA layer in a transparent manner. The Bedrock defenses consist of a set of network programs in P4 [13, 19], a high-level language for programming network devices. Bedrock provides support for many security mechanisms that are missing or inadequate in RDMA, including source authentication, access control, as well as monitoring and logging, which further serve as building blocks for mitigating myriad attacks. Operating in network hardware, Bedrock only incurs CPU processing for RDMA setup operations, where software processing is already involved and inevitable. In addition, these extra setup operations are also achieved without OS or RDMA library changes via the eBPF framework [4].

In TCP/IP networks, researchers have used programmable switches for various security applications [23, 34, 41, 58, 63, 64], but compared to these work Bedrock represents a departure in several dimensions. It focuses on RDMA security instead of traditional TCP/IP, it develops a suite of protections under varied threat models, and its conceptual novelty is to demonstrate that we can develop *a secure network foundation underneath the RDMA layer while maintaining transparency*

Xing and Hsu contributed to this work equally.

to legacy systems. This design principle makes Bedrock easier to adopt, and it also points to a path for seamlessly integrating new RDMA defenses if more attacks should be discovered in the future: by programming them into the network.

Concretely, the Bedrock defense primitives enable source authentication, access control, and monitoring and logging in cloud datacenters. The individual defenses in Bedrock draw parallels from defenses in TCP/IP settings, but they are architected to bypass CPUs and customized for RDMA:

- The Bedrock *authentication* mechanisms are inspired by source authentication and path validation techniques for IP networks [36–38]. But instead of relying on "clean-slate" architectures, Bedrock is compatible to today's networks. It binds RDMA endpoints to network invariants (e.g., the datacenter topology), which in cloud settings are outside the adversary's control. This enables Bedrock to recognize spoofed RDMA packets by checking these invariants as they are processed by the network.
- The Bedrock *access control* refines the isolation mechanisms in RDMA, such as memory regions (MRs), memory windows (MWs), and protection domains (PDs), which not only use insecure, easily-bypassable tokens, but also are hardwired in the RNIC. Bedrock takes a "software-defined" approach, exposing the ACL mechanisms for programmability by building access control primitives inside the switch. This also opens up opportunities to customize or extend RDMA ACLs for network- or application-specific policies.
- The Bedrock *monitoring and logging* mechanisms borrow techniques from network telemetry [29, 50], which uses programmable switches to monitor network traces or collect compressed TCP/IP traffic logs. In Bedrock, the defense in effect monitors memory access patterns inside the network, and the produced traces can detect abnormal accesses and enable security auditing.

Combined, these techniques significantly improve the status quo of RDMA security. The rest of this paper presents the Bedrock design in detail, and evaluates it comprehensively using realistic setups. Our results demonstrate that Bedrock can mitigate a range of RDMA attacks with minimal overheads. Bedrock has been released in open source [3].

## 2 Motivation and Background

In this section, we provide necessary background on RDMA, explain its security mechanisms and known attacks, ending with an overview of Bedrock.

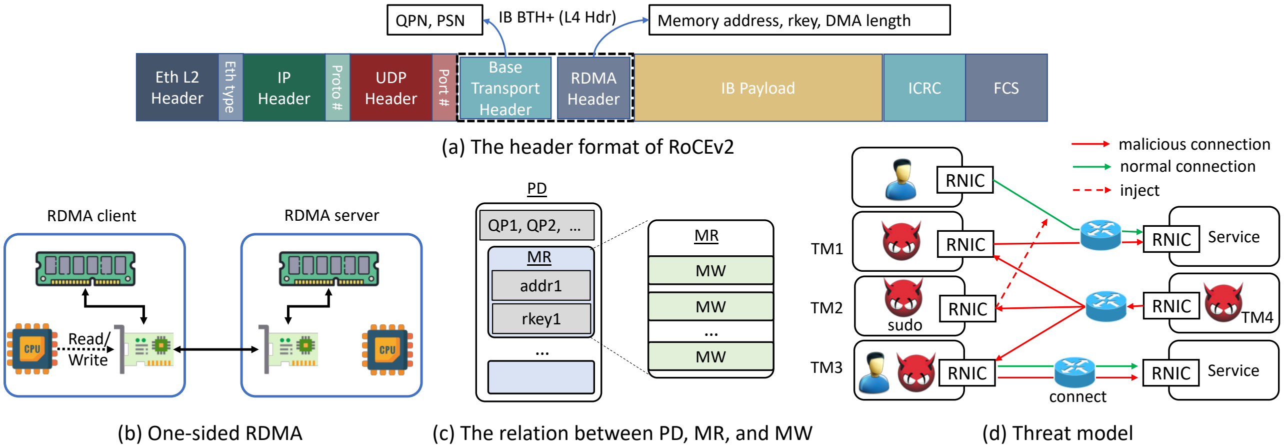
### 2.1 RDMA primer

Remote Direct Memory Access (RDMA) was proposed by the HPC community decades ago for high-performance computing, where low latency and high throughput are crucial to application performance. Compared to TCP/IP networks, RDMA significantly reduces software latency and CPU utilization by kernel and CPU bypassing. An RDMA application can directly issue read/write requests to the remote server as if the memory is local. Further, these requests are processed by the recipient without involving the CPU—hence the name remote "direct" memory access. Under the hood, RDMA requests are processed by special hardware engines in RNICs (RDMA network interface cards) for high performance. Given its performance advantages, RDMA has gained wide deployment in datacenters recently in response to the growing demands on network performance and become foundational to many modern datacenter applications [1, 2, 24, 25, 39, 43, 48, 60–62].

**RDMA API.** RDMA provides two types of API calls. Two-sided API calls, such as SEND and RECV, are similar to traditional RPC messaging as they require CPU involvement. The sender CPU issues a SEND request with a data buffer to the RNIC, which transmits RDMA packets to the receiver. The recipient CPU issues RECV to its RNIC to setup receive buffers for incoming data. One-sided API calls, such as READ, WRITE, and ATOMICS, eliminate CPU overhead at the receiver side. Except for connection setup, clients directly manipulate remote memory without the recipient CPU's knowledge. One-sided calls deliver strong performance benefits as they have an accelerated datapath at ASIC speeds. Figure 1(b) depicts the workflow of one-sided memory accesses. The vulnerabilities discovered by the security community also center on one-sided, CPU-bypassing RDMA calls [46, 49, 52, 54, 56].

**RDMA mechanisms.** In one-sided RDMA, a server exposes its memory to remote clients by memory regions (MRs). A server registers an MR with a pair of local and remote protection keys (lkey and rkey, respectively), intending only for RDMA clients with the knowledge of the rkey to access the corresponding MR. The communication channel is represented by two dedicated hardware queues on the sender and receiver RNICs, comprising a queue pair (QP) identifiable by its queue pair number (QPN) at each end. The connection setup process involves the exchange of rkey, QPN, and MR, all as a form of (unfortunately, insecure) credentials. RNICs usually come with software connection managers [14] as a library to manage connection setups. Such RDMA libraries use unencrypted TCP channels, but applications can employ secure protocols (e.g., HTTPs) if they choose to implement their own setup process.

**RoCEv2.** A widely used implementation is "RDMA over Converged Ethernet Version 2" (RoCEv2) [15], where RDMA packets are carried by Ethernet frames over traditional network infrastructures using UDP and IP as the carrier. Compared to Infiniband (IB) in HPC networks, which requires specialized network infrastructures, RoCEv2 is backward compatible with the Ethernet L2 infrastructure [68]. Therefore, RoCEv2 is the technology of choice for large-scale deployments in cloud datacenters—it is also Bedrock's protection target. In this setting, RNICs add and remove Ethernet head-



(a) The header format of RoCEv2

(b) One-sided RDMA

(c) The relation between PD, MR, and MW

(d) Threat model

Figure 1: Remote direct memory access and its security implications. (a) A common RDMA protocol called RoCEv2. (b) One-sided, CPU-bypassing RDMA operations. (c) Security mechanisms in RNIC hardware. (d) Threat models under which attacks have been discovered in recent projects.

ers before interpreting RDMA semantics and completing the memory operations via local DMA. Figure 1(a) shows the packet format. A well-known UDP destination port is used to indicate the use of RDMA, and the inner RDMA header itself contains rkey, destination QPN, and the target memory address. An RDMA packet also contains a packet sequence number (PSN) for in-order delivery and a checksum (ICRC) for packet integrity. All headers and payload are transmitted in plaintext.

### 2.2 RDMA-native security support

RDMA has several built-in, basic security mechanisms. Recent work has demonstrated that they are insufficient in shared, multi-tenant cloud datacenters.

**Authentication.** Upon receiving a request, the RNIC performs basic authentication in three aspects. First, the request must target an existing server-side queue pair number (QPN) that has been negotiated in the setup phase. Second, it must target a valid memory region (MR) with a correct rkey. Finally, the accessed virtual memory address must be within the destination MR. Requests that violate any of the above constraints will be dropped by the RNIC without notifying the receiving application. Those that pass all checks are translated into local DMA requests from the RNIC, via the PCIe interconnect, to the main memory.

**Authorization.** RDMA supports three insecure and fixed-function access control mechanisms: protection domains, memory regions, and memory windows. As Figure 1(c) shows, a protection domain (PD) contains a group of queue pairs that have the same access control privileges for the same set of memory regions. A memory region (MR) is further associated with virtual memory boundaries and access control privileges (e.g., read-only vs. read-write). A memory window (MW) is akin to a fine-grained protection domain with only one queue pair. These mechanisms are insecure and easily bypassed if the attacker presents the correct rkeys and QPNs. Also, RDMA ACL mechanisms pose integration issues with other forms of cloud ACLs, as they are baked into hardware and CPU intervention is hard to come by [30]. Elsewhere in the cloud, non-RDMA, software-based access control systems are easily reprogrammable to incorporate alternative ACL policies.

**Integrity.** RDMA packets are unencrypted, and only use two naïve mechanisms for integrity. A 32-bit ICRC checksum is included for each packet, which is inherited from the Infiniband (IB) standard from HPC settings. When RDMA packets are carried over RoCEv2, RNIC vendors view ICRC as redundant as Ethernet already has checksum mechanisms. [1] A 24-bit packet sequence number (PSN) functions similarly as TCP sequence numbers, enforcing ordered delivery and preventing packet injection. However, the checksum algorithms and seeds are publicly available, which leads to low-entropy PSNs. A recent project, sRDMA [52], has developed cryptographic support for RDMA packet encryption.

### 2.3 Threat model and attacks

We base our threat model upon ReDMArk [46], the state-of-the-art study of RDMA vulnerabilities. In particular, we target threat models for multi-tenant cloud datacenters, as depicted in Figure 1(d), where clients access remote servers via the network. In this setting, we assume that the network and server infrastructure are part of the trusted computing base (TCB). Only clients are considered to be potentially malicious. Although ReDMArk has also identified attacks that are possible under an actively malicious network, such a threat model is beyond the scope of Bedrock. We present the considered threat models below and note on the correspondence to those presented in ReDMArk when appropriate.

[1] Discussion with Nvidia/Mellanox.

**Threat Model TM1:** The attacker is located at a different end host from the victim, either in a virtual machine (VM) or container. She does not have root privilege on the cloud machine, so cannot inject raw RDMA packets or sniff network traffic. However, she can launch arbitrary malicious RDMA apps, which can issue reads and writes to remote servers—e.g., racks hosting RDMA-enabled storage services. This corresponds to the T1 threat model in ReDMArk.

**Threat Model TM2:** An enhancement of TM1. The adversary runs in baremetal cloud machine, or she has compromised the cloud software stack and obtained root privilege. She can therefore fabricate and inject arbitrary RDMA packets without needing to establish queue pairs first. The spoofed packets are manipulated to use fake UDP, RDMA, or other headers. This corresponds to the T2 threat model in ReDMArk.

**Threat Model TM3:** The adversary and victim run VMs or containers on the same machine. The attacker launches malicious RDMA apps, but she does not have root privilege and cannot subvert the client VM/container directly. However, unlike TM1 and TM2, the adversary's and victim's RDMA traffic originate from the same cloud node, and their packets are routed by the network via the same paths. This distinction is relevant for Bedrock as it leverages network invariants for defense; but this does not correspond to a separate threat model category in ReDMArk.

**Threat Model TM4:** Finally, we also consider cases where attackers have compromised an RDMA client on another machine via techniques like code injection and malware. This corresponds to the T4 threat model in ReDMArk, where adversaries can exfiltrate data to other machines silently.

The T3 threat model in ReDMArk assumes a malicious network, which is beyond the scope of our discussion. Under threat models TM1-TM4, we summarize the possible attacks identified in existing work [46, 49, 54, 56].

### 2.3.1 Insecure source authentication

Existing RDMA authentication mechanisms can be bypassed easily. An attacker can gain access to other users' remote memory regions if she obtains the correct QPN, rkey, and memory address. Unfortunately, these elements can be predicted by the attacker [46]. *(i) QPNs:* ReDMArk [46] finds that RNICs generate the next QPN by incrementing the current one. If an attacker finds her QPN to be $n$, she can reliably infer that $[0..n-1]$ are valid numbers as well. *(ii) rkey:* The rkey generation mechanism is also predictable [46, 56], e.g., either by incrementing rkeys by a fixed delta (e.g., Broadcom RNICs increment by 0x100 for each new rkey), or using low-entropy randomness (e.g., static initialization values and the same key generator for all protection domains) [46]. *(iii) Memory addresses:* Since traditional ASLR [57] (address space layout randomization) does not apply to memory ranges exposed by RDMA, virtual addresses are trivial to guess. If an adversary wishes to launch injection attacks, she needs to additionally guess the packet sequence number (PSN), which is intended to guarantee ordered delivery just like in TCP. However, TCP sequence numbers have random initial values as generated by the OS, whereas many open-source RDMA apps hardcode their initial PSNs [46]. RDMA sequence numbers are also more susceptible to brute-force attacks as they are shorter than TCP sequence numbers (24 vs. 32 bits). An attacker can enumerate the entire space in 10s for Mellanox RNICs and 23s for Broadcom RNICs [46]. These vulnerabilities lead to the following attack scenarios.

**Scenario 1 (S1):** *Unauthorized memory access via malicious queue pairs (TM3).* An attacker Bob creates its client QP, but accesses server memory granted to Alice's QP. He achieves this by using Alice's rkeys and using Alice's server-side QP number without negotiating with the server. This is feasible under TM3 where Bob and Alice reside in the same machine and have the same IP address.

**Scenario 2 (S2):** *Unauthorized memory access via raw packet injection (TM2).* An attacker Bob can inject packets to Alice's queue pair if he can craft raw packets as root user. He also needs to obtain Alice's QPN, memory address, rkey, and PSN using the methods described in ReDMArk. This corresponds to the A1 attack in ReDMArk.

**Scenario 3 (S3):** *DoS attacks by increasing the expected sequence number (TM2,TM3).* Bob can inject packets to advance Alice's PSN, either using root privilege (TM2) or by locally modifying his QP's PSN without root privilege (TM3). This will cause the server to drop actual packets from Alice with lower sequence numbers. The DoS attack can be prolonged by continued packet injection.

**Scenario 4 (S4):** *DoS Attack by transitioning QPs to an error state (TM2,TM3).* Bob forces an existing QP to transition into an irrecoverable error state via injecting malformed packets either using root privilege (TM2) or by modifying his valid QP to target Alice's QP at the server side without root privilege (TM3). Injected packets may use incorrect memory opcode (e.g., a read-only QP receives a write request) or inconsistent payload and DMA lengths [46]. These errors can cause RNICs to trigger error detection and bring down the victim QP [7]. This corresponds to the A2 attack in ReDMArk.

### 2.3.2 Inadequate access control

Existing RDMA isolation mechanisms fall short in several aspects. First, they rely on insecure tokens (e.g., rkeys) that are generated by hardware in predictable patterns [46]. An adversary can easily guess another queue pair's rkey and perform injection attacks. Second, they have fixed-function semantics and cannot be easily integrated with other forms of cloud ACLs as they bypass the software stack [30]. The authors of ReDMArk have performed a study of open-source RDMA projects, and found that many existing systems only use one single protection domain (PD) [46]. Following ReDMArk's methodology, we have studied 13 publicly available RDMA systems [6, 24, 28, 31–33, 40, 44, 51, 52, 55, 60, 61], and found that none of them uses memory windows (MWs) either. This means that rkey-guessing attacks can be easily launched against these systems due to inadequate access control. Implementing ACLs in the network switch re-exposes programmability and puts control back to the network operator's or the RDMA application's hands. They can enforce customized ACL policies in the network, or integrate RDMA ACLs with other forms of access control.

**Scenario 5 (S5):** *Unauthorized memory access that crosses access control boundaries (TM1-TM3).* An adversary can cross the boundaries of RDMA access control and gain access to other clients' queue pairs. This corresponds to attack A3 in ReDMArk.

### 2.3.3 Lack of monitoring and logging

Security auditing is critical for cloud services—logs are usually maintained by the servers for forensic purposes, which in turn enable attack detection, postmortem analysis, and service repair. However, CPU-bypassing RDMA requests are not visible to these auditing systems. The following attacks can easily slip under the covers without leaving a trace.

**Scenario 6 (S6)**: *DoS attacks via queue pair exhaustion (TM1-TM3).* An attacker can exhaust RNIC resources by creating a large number of QPs to deny the service to other clients. In principle, 24-bit QPNs can distinguish between $2^{24}$ QPs, but in practice RNICs only support a much smaller number: 32,707 for Broadcom and 261,359 for Mellanox RNICs [46]. This corresponds to attack A4 in ReDMArk.

**Scenario 7 (S7):** *Performance degradation attacks (TM1-TM3).* RDMA packets bypass server CPUs but incur processing overhead at RNIC engines, and these engines have their own processing limits. An attacker can issue a large volume of RDMA requests to overload the RNIC engine itself, so that normal clients' packets cannot be processed. ReDMArk [46] shows that an attack from 1-2 nodes can easily reduce normal clients' read throughput by 8x-10x. This corresponds to attack A5 in ReDMArk.

**Scenario 8 (S8):** *Side channel attacks (TM1-TM3).* RNICs have on-board memory to cache data structures like page table entries, and they use the main memory as a backing store to handle cache evictions and misses. Cache misses trigger expensive main memory accesses as they cross the PCIe bus in a round-trip, leading to detectable latency differences compared to cache hits. An attacker can use prime-and-probe or evict-and-reload techniques to cause cache evictions and construct timing channels to observe memory access patterns of a victim client served by the same RNIC [54].

**Scenario 9 (S9):** *Data exfiltration (TM4).* CPU bypassing can also lead to exfiltration attacks. Normally, exfiltration attacks will leave audit trails in software logs, which can be used for forensic analysis. However, if an attacker compromises an RDMA node, she can leak data by initiating connections from remote machines and directly reading the memory of the compromised node without leaving a trace. This makes it challenging to even detect data exfiltration attacks. This corresponds to attack A6 in ReDMArk.

## 2.4 BedRock Overview

Bedrock develops RDMA defenses in network devices with programmable data planes, which accept P4 programs for customizing packet processing behaviors. Consider programmable switches as an example. A P4 program specifies necessary packet headers and protocols for the switch. The programmable parser extracts these headers and construct packet header vectors (PHVs). The PHVs are sent through multiple hardware stages in the switch for match/action processsing. The programmable deparser reassembles headers before forwarding the packet. Bedrock leverages this capability to process RDMA headers on-the-fly for source authentication.

The main P4 processing takes place in a series of match/action tables. Each table may have different keys (e.g., header fields) and perform varied actions (e.g., arithmetic or bitwise operations) on the headers. Hardware ALUs and CRC hash units are integrated with every processing stage to enable programmable actions per packet. Table entries, on the other hand, are stored in switch SRAM (Static RAM) or TCAM (Ternary CAM) for exact and range matches, respectively. Each type of resource has its own constraints—typically, O(100Mb) for SRAM and O(10Mb) for TCAM for a programmable switch. TCAM tables are especially important for RDMA access control in the design of Bedrock.

Switch SRAM can also be used as register arrays to store data across packets. These are akin to arrays in C, where each register is accessible via an index. A packet is limited to accessing one register per stage, so K accesses are possible across K stages. Bedrock leverages registers for RDMA monitoring and logging; it also uses packet mirroring primitives to duplicate packets.

**Deployability.** Bedrock requires the use of P4-programmable network devices in RDMA deployments. Although P4 is a recent development, it is gaining traction from industry [12]. Academic projects also make extensive use of P4 switches [23, 35, 41, 42, 50, 58, 67]. As an example of real-world adoption, Alibaba has deployed Tofino switches in their production networks at scale [53]; the same cloud provider also relies on RDMA deployments for their storage service [26]. We believe that these industry trends show promise of the deployability of Bedrock in realistic scenarios. Our primary design goal is for Bedrock to run in top-or-rack switches to protect racks of servers.

As P4 is a target-independent language, P4 programs can be deployed to NIC-based targets as well. Existing P4-capable platforms not only include programmable switches like Intel Tofino and Nvidia Spectrum, but also programmable NICs like Netronome Agilio and Xilinx Alveo. These platforms primarily differ in their cost-to-performance ratios, performance characteristics, and deployment scenarios. Switching ASICs have lower cost-to-performance ratios and can be deployed to serve entire racks of servers. Programmable NICs have higher cost-to-performance ratios and are suitable for server-local deployments. Programmable NICs also exhibit more performance variability than switching ASICs. We show some concrete data points by analyzing the list price from the same vendor: today, a 3.2Tbps P4 switch costs $8,695 [5], translating to $2.7 per Gbps; a 40 Gbps P4 NIC costs $555 [11], and this translates to $13.8 per Gbps, which is several times higher. Later, we also include performance benchmarks on a P4-programmable NIC.

## 3 Network Support for RDMA Security

Next, we present the three components of the Bedrock system.

### 3.1 RDMA source authentication

Bedrock is inspired by work in source authentication mechanisms for the general Internet [36–38], where mutually-untrusted parties wish to authenticate packets' origins. However, unlike the Internet at large where ISPs and their network infrastructures are assumed to be untrusted, RDMA deployments occur in controlled environments where the attacker cannot easily subvert the underlying infrastructure. Bedrock harnesses the fact that the network itself is part of the TCB, and develops lighter-weight mechanisms without requiring architectural changes and cryptographic operations for routing, forwarding, and source authentication. Instead, Bedrock derives identifiers that are coupled with the cloud infrastructure (e.g., the network topology and system information) that is beyond the adversary's control.

#### 3.1.1 Packets from different machines

We first consider threat models TM1 and TM2, where the attacker is located at a different machine from the victim client. Bedrock leverages network topological invariants for authentication, in a manner that is transparent to applications. Spoofed traffic can be easily detected by Bedrock as such packets will violate the topological invariants. Concretely, Bedrock enables this by constructing a mapping for each RDMA client that maps from its IP address to the topological information—the switch ingress port ID (iPort) for the client. This requires assigning a unique ID for each switch port in the network (e.g., by the operator or network management tools), and configuring the mapping in the switches' match/action tables. At runtime, Bedrock switches check these invariants for each RDMA packet, and detects packets whose IP addresses do not match the topological information. In the P4 program, Bedrock enforces this based on the network mapping stored in match/action tables. Under this design, Bedrock ensures that RDMA packets are bound to the original endpoints from which the connections have been initiated. If an attacker uses the IP of the victim, her packets will be detected and blocked by Bedrock. If she uses her own IP address, her packets will be rejected by the remote RNICs as the source IPs are inconsistent. Combined, Bedrock mitigates impersonation attacks effectively.

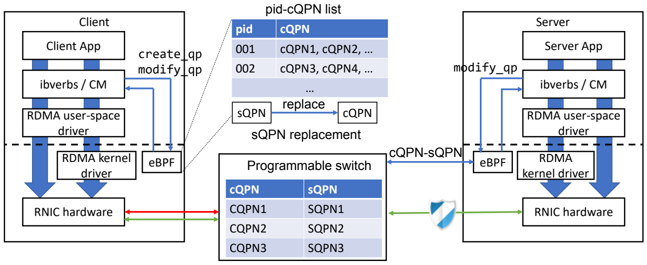


Figure 2: The illustration of source authentication for clients in the same machine.

#### 3.1.2 Packets from the same machine

Next, we consider threat model TM3, in which spoofing attacks can be launched from the same machine as the victim. This creates more challenges for recognizing spoofed packets, as traffic originating from attackers and the victim are indistinguishable by the ingress port information. Therefore, we need to go one level further and identify other types of system information that the attacker cannot easily control.

Figure 2 illustrates our design. Bedrock relies on the process ID (pid) which is generated by the OS kernel, as another class of identifiers. RDMA connections are bound to identifiers that reflect both topological constraints and system information. To achieve transparency, we extract the pid using eBPF [4], which is an infrastructure present in most mainstream Linux kernels. eBPF allows the injection of runtime monitors into the kernel without affecting user applications. We intercept the RDMA verb `create_qp`, which creates a hardware queue at the local RNIC and returns the generated QPN to the application. Bedrock constructs a list of client QPNs (cQPNs) for each pid during its lifetime, and destroys the list upon application exit. Therefore, a malicious application that misuses other applications' QPNs can be detected by the eBPF script.

However, the above defense does not prevent attackers from using their own cQPNs to communicate with some server QPN (sQPN) that they have not authenticated with. At the server RNIC, the ASIC only checks if the server QPN is valid. It does not validate the originating client QPN—in fact, cQPNs are not even included in RDMA packets. This is the root cause why attack S1 is feasible. One way to mitigate this attack is to modify the RDMA standard for the inclusion and validation of cQPNs; Bedrock instead resorts to a transparent approach: overwriting the sQPN header with the cQPN value. Specifically, Bedrock first ensures that the client uses its own cQPN by checking the pid-cQPN list. It then uses the eBPF framework at the client side to intercept the `modify_qp` call and replace the parameter sQPN with cQPN. In this way, Bedrock tightly controls which server QPN a client can communicate with. Even if clients specify a different server QPN, it will get modified by Bedrock to the correct value.

So far, Bedrock establishes an invariant that, for all inflight RDMA requests, the sQPNs that they carry are equal to the originating cQPNs. At the server side, Bedrock authenticates the request in the network by querying a switch-based mapping to obtain the actual server QPN. This mapping is obtained by the server-side eBPF framework, which similarly monitors `modify_qp` calls. The actual server QPNs, and the originating client QPNs, are inserted to the switch in a match/action table. RDMA requests match on this table, and their sQPN headers are replaced with actual server QPNs before they are sent from the switch to the RNIC. In effect, Bedrock interposes a layer of indirection for security.

One practical issue here is the ICRC checksum of RDMA packets need to be recalculated when the last-hop device changes the sQPN. This can be easily done using programmable NICs, but current programmable switches cannot easily support this [17]. However, ICRC fields are redundant for RoCEv2 packets as Ethernet frames already have checksums. This feature was inherited from the Infiniband (IB) version of RDMA, and can be disabled in RoCEv2 settings. Our setup disables ICRC to resolve this issue.

## 3.2 RDMA access control

Next, Bedrock develops a "software-defined" approach to RDMA access control. Today, RDMA ACLs are hardwired and pose integration issues with other types of cloud ACLs [30]. By offloading ACLs to a programmable switch, Bedrock exposes RDMA access control for programmability. Datacenters regain the ability to customize or modify ACL policies for CPU-bypassing traffic. Advanced, scenario-specific ACL policies also become possible without having to resort to software intervention. Section 3.3 presents several policies that monitor RDMA traffic patterns and make access control decisions based on these patterns—e.g., denying access if signs of DoS attacks are detected. Here, we first focus on developing RDMA ACL support in programmable switches.

Figure 3(a) shows our design, which consists of an RDMA external library on servers, a policy setup daemon in the switch control plane, and a policy executor in programmable data planes. The library offers an easy-to-use API for users to specify ACL groups and add/remove QPNs to/from a particular group. The API is invoked in an application-independent manner without RDMA application changes—i.e., the user writes a configuration script without modifying her apps. If desired, the RDMA apps can also call into this library for direct integration. The ACL policies are sent to the switch daemon by Bedrock, which configures them into the switch programs. Memory ranges, queue pairs, and RDMA opcodes are all part of the policy decisions. The policies are implemented in programmable match/action table, where rule insertions and deletions reflect ACL changes.

| Start address range | End address range |
|---|---|
| 0000 FF00 0000~00FF 00FF FFFF | 0000 FF00 0000~00FF 00FF FFFF |

Table 1: An example memory range for ACL.

| ID | bit[47:32] | bit[31:16] | bit[15:0] |
|---|---|---|---|
| r1 | 0000~0000 | FF00~FFFF | 0000~FFFF |
| r2 | 0001~00FE | 0000~FFFF | 0000~FFFF |
| r3 | 00FF~00FF | 0000~00FF | 0000~FFFF |

Table 2: Each memory address requires three TCAM rules.

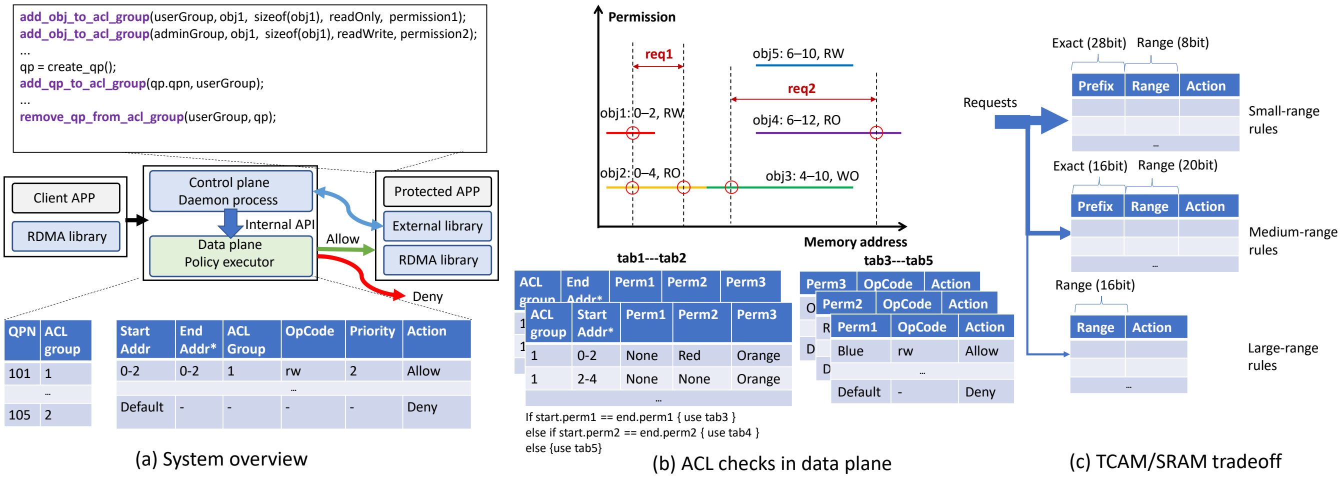
The policy executor is a P4 program that enforces access control in the switch data plane. Its rules are populated by the configuration scripts stated above. The executor checks *i*) whether the memory range of an RDMA request is within the memory boundary assigned to the client; and *ii*) whether the requested operation is allowed using the match/action tables. As shown in Figure 3(a), Bedrock instantiates an ACL table with five keys: the start and end memory addresses, ACL group, opcode, and priority. The memory addresses use range matches in TCAM; other keys use exact matches in SRAM.

### 3.2.1 Compressing RDMA ACLs

The design of the data plane executor creates challenges due to switch resource limitations. Whereas memory addresses are 48-bit long (for 64-bit systems), TCAM range matches have shorter lengths (20 bits in Intel/Barefoot Tofino [10]). A simple solution is to segment a 48-bit memory address into three 16-bit segments and designate a range match for each of segment. The overall match result will depend on the three individual matches. However, this will consume large amounts of TCAM resources since the segments will contain duplicate information. Consider a memory range 0000 FF00 0000~00FF 00FF FFFF in Table 1. To check whether a request is contained by this, logically, we only need to check the request's start address against this range and then its end address. However, each memory address (start/end) needs to be split into three 16-bit rules (see Table 2) due to TCAM restrictions. Checking a request's start and end addresses against this memory range further requires six rules stemming from a "cross product". This creates high overhead, as switches only have O(10Mb) TCAM. We address this using a combination of three techniques.

*#1: Adjustable ACL granularity.* First, the above overhead is only necessary if we require byte-level access control. If coarser-grained ACLs are sufficient, Bedrock can ignore the least significant bits. Bedrock makes the ACL granularity adjustable—for instance, if 4kB page-level ACLs are sufficient, Bedrock ignores the 12 LSBs and uses two segments to represent a page address.

*#2: Table decomposition.* Second, we reduce the redundancy of start/end address combinations by decomposing a logical table to two different tables, one for the start address and another for the end address. This deduplicates the logical table by avoiding the "cross product" problem. Figure 3(b) visualizes this idea, where the Y-axis indicates different ACL permissions (e.g., RO vs. RW) and the X-axis indicates memory ranges. In this illustration, a memory object is represented as a horizontal line that specifies a memory range at a priority



```
add_obj_to_acl_group(userGroup, obj1,  sizeof(obj1),  readOnly,  permission1);
add_obj_to_acl_group(adminGroup, obj1,  sizeof(obj1), readWrite, permission2);
...
qp = create_qp();
add_qp_to_acl_group(qp.qpn, userGroup);
...
remove_qp_from_acl_group(userGroup, qp);
```

| QPN | ACL group |
|---|---|
| 101 | 1 |
| … | |
| 105 | 2 |

| Start Addr | End Addr* | ACL Group | OpCode | Priority | Action |
|---|---|---|---|---|---|
| 0-2 | 0-2 | 1 | rw | 2 | Allow |
| … | | | | | |
| Default | - | - | - | - | Deny |

(a) System overview

| ACL group | Start Addr* | Perm1 | Perm2 | Perm3 |
|---|---|---|---|---|
| 1 | 0-2 | None | Red | Orange |
| 1 | 2-4 | None | None | Orange |
| … | | | | |

| Perm1 | OpCode | Action |
|---|---|---|
| Blue | rw | Allow |
| … | | |
| Default | - | Deny |

```
If start.perm1 == end.perm1 { use tab3 }
else if start.perm2 == end.perm2 { use tab4 }
else {use tab5}
```

(b) ACL checks in data plane

(c) TCAM/SRAM tradeoff

Figure 3: The Bedrock access control overview and optimization techniques.

denoted by the Y-axis value. An RDMA request is represented with two vertical lines that represent the start and end addresses as denoted by the X-axis values. The ACL decisions are made by checking if a request's vertical lines intersect at the same memory object. If there are multiple intersected objects, the one with the highest ACL permission is used. In the example, `obj2` will be selected for `req1`; but request `req2` will be rejected, as the start and end addresses do not intersect at the same objects.

*#3: TCAM/SRAM tradeoff.* Further, Bedrock groups rules that share the same address prefix and use SRAM-based exact matches on their common prefix. As SRAM is more abundant than TCAM, this tradeoff enables Bedrock to support more ACLs. Figure 3(c) shows how Bedrock organizes the policies hierarchically based on the common prefix lengths, and requests match against the three tables in parallel. Moreover, Bedrock adjusts the prefix lengths and table sizes based on the object size distributions, in order to maximize the use of common prefixes for each scenario. Storage objects usually follow zipfian distributions in terms of the size [16, 20, 27, 66]. As a concrete example, say, if 70% objects are smaller than 1MB and 99% smaller than 4GB, and if page-level isolation is the desired granularity, then Bedrock would create three tables. The first table uses a 28-bit common prefix and 8-bit range match, covering all memory objects under 1MB. The second uses a 20-bit range match, covering all objects between 1MB and 4GB. The tail distribution matches the third table with range matches.

## 3.3 RDMA monitoring and logging

Bedrock enables RDMA systems to regain visibility by in-network monitoring and logging.

### 3.3.1 Building blocks

Bedrock performs RDMA monitoring in the switch. Monitoring results are further used by ACL policies for advanced access control. To maintain monitoring state across packets, Bedrock borrows from work in *approximate data structures*—such as count-min sketches (CMSes) [22] and bloom filters (BFs) [18]—for space savings. We eschew the technical details of these data structures and refer interested readers to existing work [18, 22]. At a high level, a count-min sketch performs approximate counting, a bloom filter performs approximate membership checks, both with strong error bounds guarantees. For monitoring, Bedrock also supports tumbling windows, where one sketch or bloom filter records data for the current window and another for the next. It rotates across these entities for each time epoch.

Bedrock enables RDMA logging by tracking RDMA requests and recording them at backend servers. It does not log every single RDMA data packet, but only RDMA requests, which already contain sufficient metadata (e.g., QPs, memory addresses, opcodes) to build audit trails. By cherrypicking these request packets from the rest, Bedrock enables NetFlow-like logging and auditing on RDMA traffic. This is done in a P4 program that extracts RDMA metadata for request packets, and stores them in stateful registers as a temporary buffer. It reads/writes one register per switch stage, but multiple accesses are possible across several stages. Therefore, one RDMA log entry is generated for a batch of requests, which further correspond to a much larger number of data packets. This reduces the bandwidth that is needed for auditing.

### 3.3.2 Regaining visibility

Using the auditing capability, Bedrock enables the detection and mitigation of the following types of attacks.

**DoS attacks.** Bedrock is able to detect and block DoS attacks, including S3, S4, S6, and S7. To detect S3, Bedrock monitors the highest PSN of each remote QPN that has been seen by the switch. Bedrock detects S4 by checking the consistency between opcodes and MR privileges and between payloads and DMA lengths. Inconsistency would transition queue pairs into an error state for denial of service. For S6 and S7, Bedrock monitors the resource usage of each connection, application, or IP address to make access control decisions using the approximate data structures. Concretely, to detect and mitigate S6, Bedrock checks the number of QPs created by each client IP in a time window against a threshold. Upon detection, Bedrock drops or rate-limits the requests. Bedrock detects S7 by tracking the sizes of the requests sent to each QPN in a time window.

**Side channel attacks.** Another application is to detect side channel attacks [54] based on the fact that malicious traffic has different memory access patterns from normal traffic [47]. Side channel attack S8 builds an eviction set by reading a set of specific memory pages, which produces a distinct memory access pattern. Bedrock logs memory read requests to a backend server from the switch. The server further uses software-based algorithms (e.g., SCADET [47]) to perform memory pattern analysis to detect S8.

**Exfiltration attacks.** Armed with in-network logging, RDMA reads and writes become auditable—operators can scrutinize the log to find unexpected requests. To enable forensic investigation of data exfiltration attacks (S9), all reads/writes are logged to the backend server, and the log entries are processed in software to identify data exfiltration.

**Further customization.** The monitoring and logging capability in Bedrock serves as a building block for scenario-specific security applications. As a concrete example, consider reconnaissance attacks, which are a necessary step for the adversary to guess rkeys. In this reconnaissance phase, the attacker enumerates the rkey space and tests whether a particular rkey is valid, e.g., by trying many different rkeys until success and reconnecting if a guess fails [46]. Bedrock tracks the number of distinct rkeys that a machine (as identified by its IP address) has attempted to a remote memory region. First, it uses a combination of the source IP and the rkey as the key to a bloom filter to check whether this probe has been seen before. If not, it increments the count-min sketch counters using the source IP as the key to track the number of probes. Further probes are blocked by the Bedrock ACL if the sketch counters exceed a threshold.

## 4 Evaluation

In this section, we describe our prototype implementation, experimental setup, and present a comprehensive set of experiments to evaluate Bedrock. Our evaluation focuses on several dimensions: a) the effectiveness of Bedrock against RDMA attacks; b) the overhead of Bedrock; c) workload-based evaluations; and d) comparison with a programmable NIC deployment.

### 4.1 Prototype and setup

**Prototype.** We have implemented a Bedrock prototype using approximately 6700 lines of code, including the various defense primitives, eBPF functions, switch control plane functions, and backend server logging module.

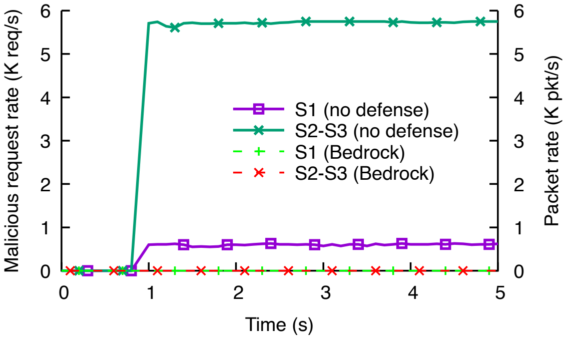


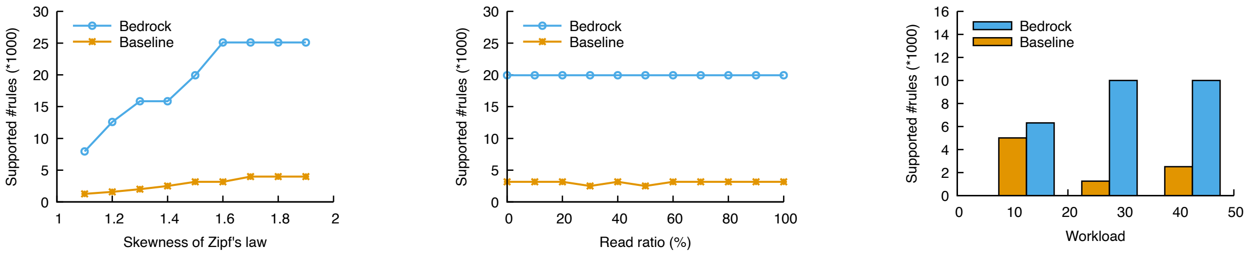
Figure 4: Bedrock enables source authentication. With Bedrock, S2-S3 attacks are blocked at the switch (right Y-axis); S1 impersonation uses its own QP and is dropped by the RNIC (left Y-axis).

**Experimental setup.** For our main experiments, we have deployed Bedrock to a Wedge 100BF-32X Tofino switch, which has 32×100Gbps ports and is programmed in P4. The switch is furnished with an eight-core Intel CPU at 1.60GHz for the control plane, which runs a Debian 8.9 Linux distribution as the operating system. It is configured as a Top-of-Rack switch in our cluster, which is connected to a RDMA client node, a RDMA server node, and a backend logging node. Our cluster also contains several other machines, from which we launch RDMA attacks based on the methodology in ReDMArk (S1-S7, S9) [46] and Pythia (S8) [54]. All machines have a six-core Intel Xeon E5-2643 CPU, 128 GB RAM, 1 TB hard disk, all running an Ubuntu 18.04 OS. They are also equipped with Mellanox ConnectX-4 MT27710 25Gbps RNICs that are configured to use RoCEv2.

**Methodology.** We validate the ability of Bedrock defenses to detect and mitigate attacks S1-S9. In addition, we measure the overhead of Bedrock (e.g., control and data plane overheads, switch resource utilization). Our workload-based evaluation generates realistic workloads following the metholodogy of existing projects, and measures the impacts of Bedrock on request completion times (RCTs) and throughputs. Last but not least, we deploy Bedrock to a P4-programmable NIC, Netronome Agilio CX, to understand its performance characteristics as compared to switch-based deployments.

### 4.2 RDMA source authentication

Figure 4 shows the effectiveness of Bedrock against impersonation attacks via authentication. We test each attack with and without Bedrock enabled to evaluate the difference. Attack S1 launches impersonation attacks using its own QP, and the figure plots the attacker's request rates. Attacks S2 and S3 work in different ways, but the defense effects are similar; we group them in the same curve that shows the number of packets that are successfully injected to existing QPs. As we can see, without Bedrock, the attack traffic reaches the servers



(a) Influence of skewness

(b) Influence of read rule ratio

(c) Workload-based evaluation

Figure 5: Bedrock outperforms the baseline by its ACL compression techniques. We further measure Bedrock under different skewness of memory region sizes, read/write rule ratios, and use realistic workloads based on Twitter, Idiada, and Arctur traces.

and successfully masquerades as the client, resulting in malicious memory modification. Under Bedrock, all attacks are detected and blocked. We then launch attack S4, which attempts to inject 1000 packets with inconsistent payloads and DMA lengths. Bedrock has detected and dropped all packets (not shown in figure).

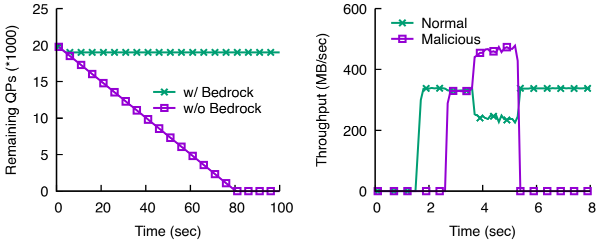
### 4.3 RDMA access control

Next, we evaluate the effectiveness of Bedrock ACLs by launching attack S5 that violates access control policies. Specifically, we create attacks that attempt to a) access memory addresses beyond the granted ranges or b) to access valid memory addresses using ungranted permissions, e.g., writing to an object when read-only privilege is granted. All such attacks are recognized by Bedrock and blocked at the switch.

We then evaluate the effectiveness of Bedrock in compressing RDMA ACLs, against the baseline solution without compression. The key metric is the number of ACL rules supported in the Tofino switch. We vary the skewness of zipfian workloads [16, 20, 27, 66], and also measure Bedrock using realistic workloads based on Twitter, Idiada, and Arctur traces [27, 66], for a comprehensive evaluation. Since the P4 compiler statically rejects a program if the ACLs consume more memory than switch resources, our methodology is to gradually increase the number of ACLs until the P4 compiler rejects the program due to resource limitations.

Figure 5a shows the results for different skewness, with a fixed read rule ratio of 0.5. We can see that both Bedrock and the baseline can support more ACLs at higher skewness with many small MRs, because smaller ranges can be encoded in fewer ACL rules. Bedrock outperforms the baseline as it leverages cheaper SRAM matches to reduce TCAM usage: the smaller ranges are supported using exact matches in more abundant SRAM. Figure 5b further indicates that the number of rules supported by Bedrock is robust to different read rule ratios. Figures 5c evaluates Bedrock with three realistic workloads. Bedrock outperforms the baseline consistently, supporting 26%, 6.95×, and 2.98× more ACLs on Twitter, Idiana, and Arctur traces, respectively.

The latest Tofino hardware contains more resources than our switch, so we extrapolate based on the specifications of Intel/Barefoot Tofino2 [9]. We estimate that the number of ACLs that Bedrock can support using Tofino2 is up to 2.98× more than that in our current switch.



(a) Defense against S6

(b) Defense against S7

Figure 6: Bedrock detects and mitigates attacks S6 and S7.

### 4.4 RDMA monitoring and logging

Next, we evaluate Bedrock with attacks S6-S9, which require monitoring and logging capabilities as well as ACL decisions based on runtime traffic patterns.

Figure 6a shows the S6 attack, which consumes a large quantity of QPs to deny the service to other users. Without Bedrock, the attack has exhausted 20k+ queue pairs within 80 seconds. Bedrock monitors and enforces an upperbound limit of 1k QPs per user, successfully detecting the DoS attack and preventing it from exhausting available QPs. Excessive requests are denied in the network.

Figure 6b evaluates attack S7, where the attacker uses multiple machines to launch a performance degradation DoS attack. At time t=1.7s, the normal client sends traffic within the enforced bandwidth limit. At time t=2.7s, the malicious client starts: it uses normal rates at first but one second later it boosts its traffic rate to launch the attack. As we can see, this significantly degrades the performance of the normal client. At time t=5.4s, we enable Bedrock, which enforces a rate limit per user. It detects and blocks the malicious client immediately, and the normal client's performance goes back to normal.

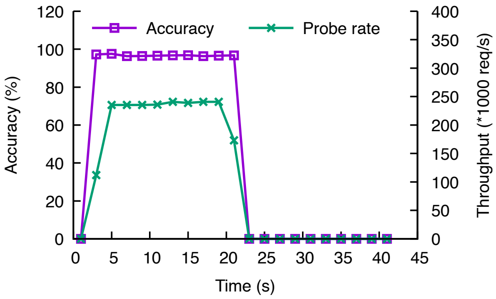


Figure 7: Bedrock can detect and mitigate the side channel attack S8 effectively.

| Defense | SRAM | TCAM | Hash bits | Meter ALU |
|---|---|---|---|---|
| **Auth.** | 4.17 % | 2.08 % | 2.48 % | 0 % |
| **ACL** | 10.52 % | 68.75 % | 27.78 % | 0 % |
| **Mon.** | 26.56 % | 7.64 % | 16.39 % | 27.08 % |
| **Log.** | 12.29 % | 11.46 % | 12.22 % | 68.75 % |

Table 3: The Tofino switch resource usage of each defense.

Next, we evaluate the ability of Bedrock to log and analyze side channel attacks (S8) using the Pythia setup [54]. The normal client and the attacker are both connected to the RDMA server, and the attacker infers the client's memory access patterns in the following way. It evicts a target victim address's cache line from the RNIC by generating many different page accesses, and then measures the access latency to the target memory address to determine whether the client has triggered RNIC caching. This attack is launched continuously to different memory addresses, and we show the accuracy of the attack over time in Figure 7. Close to Pythia's results, the inference accuracy is around 97%. At time t=22s, we enable Bedrock to detect this attack. The Bedrock switch collects memory access addresses for all QPs, and logs these entries to the backend server. The server implements the SCADET [47] side channel detection algorithm in software to detect attacks. After detection, Bedrock blocks the attacker's probes and its traffic rate drops to zero.

Bedrock enables audit trails for S9, the data exfiltration attack, using its logging capability. Figure 8 measures the logging rates and the CPU utilization of the backend server, using workloads generated from an RDMA benchmark tool, `perftest` [8]. We also show the results for different request sizes. Larger requests produce more data packets, so the logging rate is lower—this is because Bedrock only logs request metadata, not the data packets. Since each log packet contains 8 entries, the logging rate is in proportion to the request rate by a factor of 8. Moreover, the server uses one single CPU core to receive the logging data, and as the figure shows, the utilization is under 90% of one core.

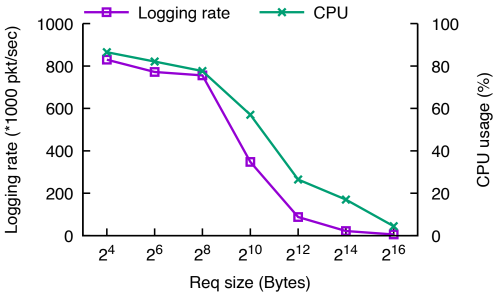


Figure 8: The traffic logging rates and logging server CPU usage of different request sizes.

### 4.5 Defense overhead

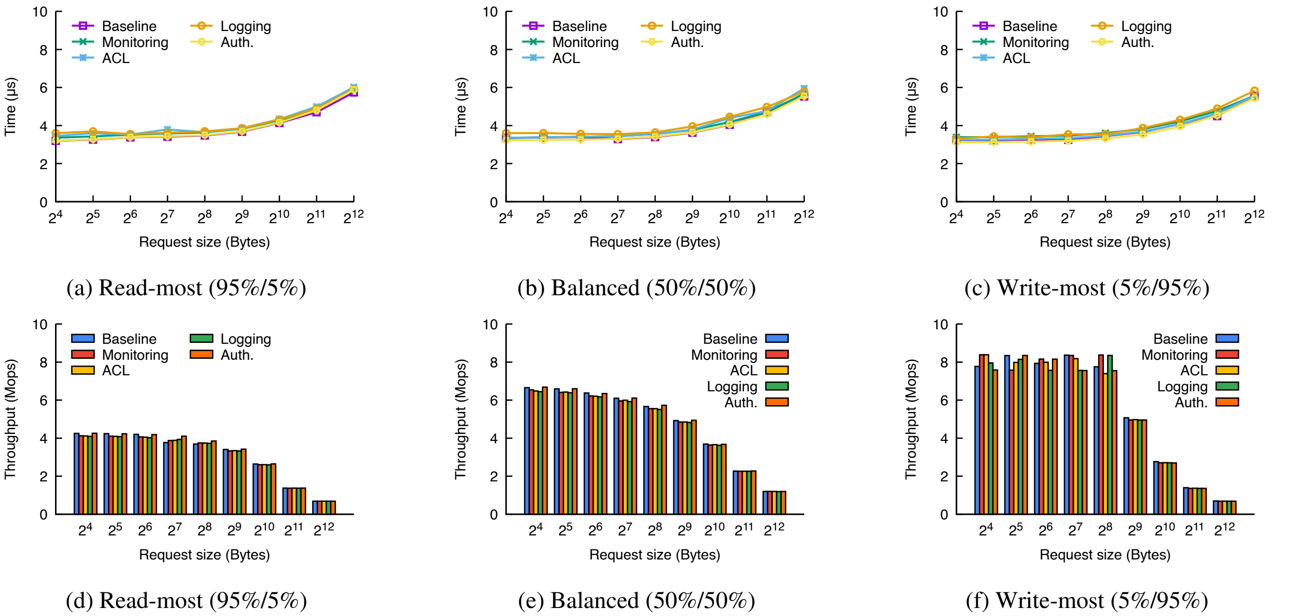
**Control operation overhead.** Since Bedrock uses eBPF in the authentication defenses, this incurs extra latency when a queue pair is initialized. To evaluate this overhead, we use a microbenchmark to measure the time needed for the RDMA library to accept new connections. With 7.04 ms as the native performance for setting up queue pairs, Bedrock increases the latency to 7.11 ms, which represents 1% additional overhead. Since this overhead is only incurred for control operations when setting up a queue pair, it preserves the goal of achieving native RDMA datapath performance.

**Resource overhead:** Table 3 shows the resource usage of Tofino switch for each of our defenses. Overall, Bedrock has reasonable resource utilization across different metrics. SRAM and TCAM are used for match/action table entries. Hash bits are used for header transformation. Meter ALUs are used to access stateful registers for monitoring as well as logging. The ACL component is evaluated with the maximum number of rules that the switch can support.

### 4.6 Workload-based evaluation

Next, we perform workload-based evaluation of Bedrock using YCSB benchmarks [21], which are widely used in recent RDMA projects [54, 59, 61]. We first use YCSB workloads at different read/write ratios, using request sizes ranging from 16 bytes to 4096 bytes. Following the HERD [31] methodology, we generate YCSB workloads and replay them to the RDMA system. We also adopt object size distributions from realistic traces based on Twitter [66], Idiada [27], and Arctur [27] workloads. Our main evaluation metrics are a) request completion times (RCTs), which measure the time it takes for an RDMA request to finish; and b) throughput, as measured by the number of RDMA requests per second. Each metric is measured with and without Bedrock.

**YCSB.** We evaluate read-most (95% reads), write-most (95% writes), and balanced (50% reads and 50% writes) YCSB traces. As we can see, Bedrock incurs low overhead. The RCTs of different workloads increase by 3.2% on average; and the throughputs decrease by 1.0% on average. The different defense components in Bedrock also have similar perfor-



(a) Read-most (95%/5%)

(b) Balanced (50%/50%)

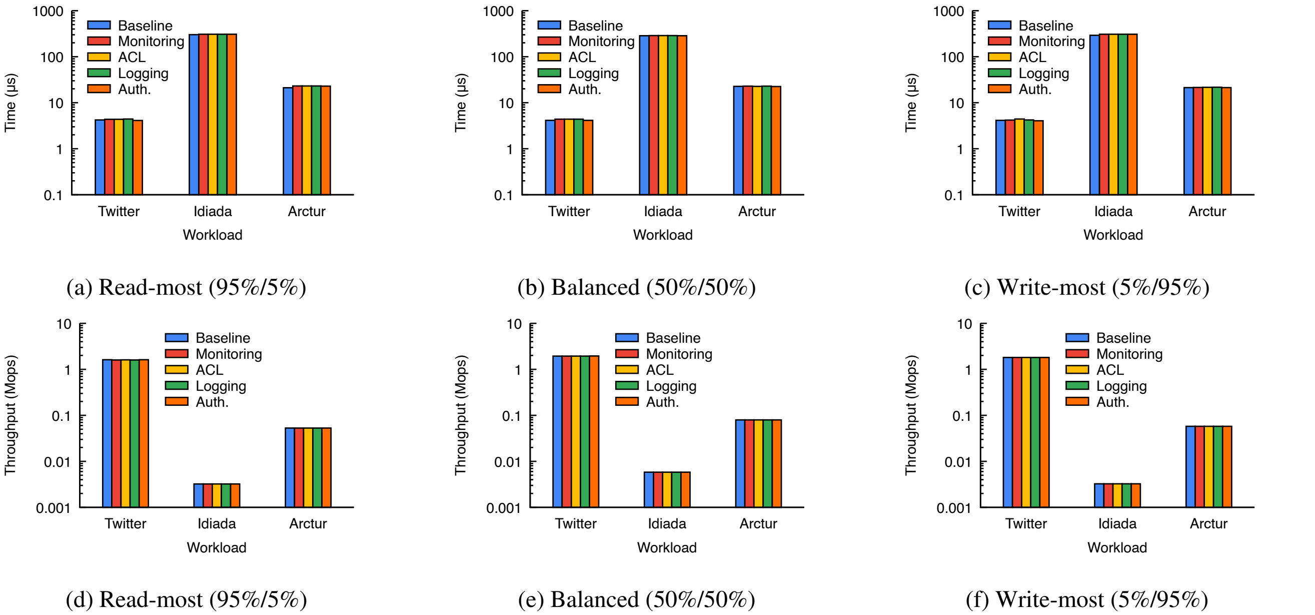
(c) Write-most (5%/95%)

(d) Read-most (95%/5%)

(e) Balanced (50%/50%)

(f) Write-most (5%/95%)

Figure 9: Bedrock incurs minimal overheads with different read/write ratios. (a)-(c): request completion times (RCTs). (d)-(e): throughputs as measured by the number of request operations per second.



(a) Read-most (95%/5%)

(b) Balanced (50%/50%)

(c) Write-most (5%/95%)

(d) Read-most (95%/5%)

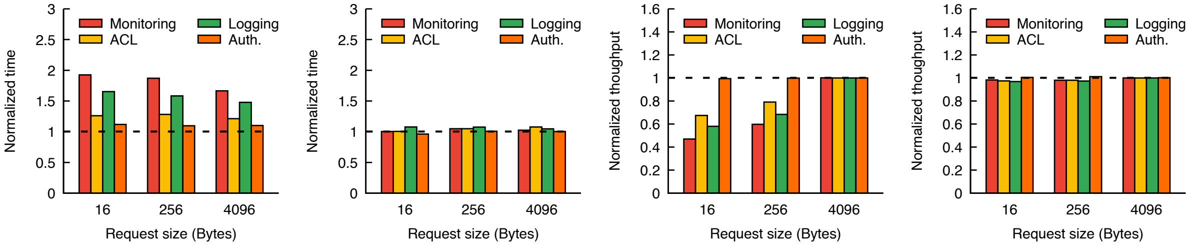
(e) Balanced (50%/50%)

(f) Write-most (5%/95%)

Figure 10: The request completion time and throughput under realistic workloads.

mance, which is expected as switching ASICs are designed to achieve near-constant performance. Larger requests lead to higher RCTs. The throughput (as measured by Mops/s) also decreases with larger requests. These trends hold for both the baseline and Bedrock.

**Twitter, Idiada, and Arctur.** Next, we use object and request size distributions from real-world traces: Twitter [66], Idiada [27], and Arctur [27], with the same set of read/write ratios as before. We again measure the RCTs and throughputs with and without Bedrock. Figure 10 shows the results, with similar takeaways: Bedrock adds an average latency overhead of 2.94% and a throughput overhead of 0.09%. Workloads enjoy minimum performance interference. Different defense components in Bedrock have on-par performance.



(a) Netronome RCT (b) Tofino RCT (c) Netronome throughput (d) Tofino throughput

Figure 11: The request completion time (RCT) and throughput of Bedrock deployed in the Netronome NIC and Tofino switch. The number has been normalized to their baselines respectively.

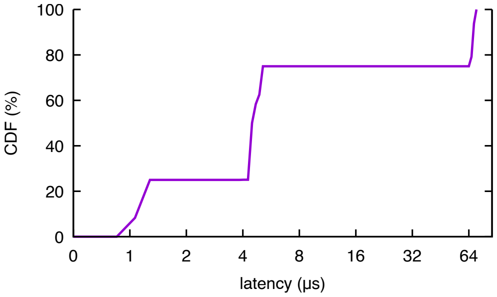


Figure 12: Microbenchmarks: The Netronome NIC exhibits variable latency across packets.

### 4.7 Programmable switches vs. NICs

Earlier, we have already discussed the differences between programmable switches and NICs in terms of their cost-to-performance ratios and deployment scenarios. Further, we have conducted a set of experiments to understand their performance characteristics. We have deployed Bedrock to a P4-programmable NIC (Netronome Agilio CX) and retested all the attacks. Our high-level findings are a) Bedrock works effectively on both platforms; and b) programmable NICs experience more notable performance variability depending on the program logic. We use the NIC as a forwarding device to connect four RNIC-based servers for benchmarking.

Figure 11 shows the performance variability of the programmable NIC across defenses, using the switching ASIC performance as a comparison point. In a programmable switch, the latency variance of Bedrock is under $0.5\mu$s consistently across defenses, but in the programmable NIC, the variance can be up to $12\mu$s. Moreover, switching ASICs also have more stable throughputs across defenses, with at most 3.2% degradation as compared to the baseline. This is because switching ASICs are designed for near-constant performance under worst-case assumptions. The Netronome NIC, on the other hand, has variable performance (up to 53% degradation across defenses) compared to baseline.

As a further microbenchmark, we have used a stress-test trace on the ACL defense in the programmable NIC, and plotted the latency CDF in Figure 12. We find that the latency of each packet depends on the program paths it takes, as well as whether the packet happens to hit the flow cache on the programmable NIC. This performance variability is consistent with existing studies of programmable NIC performance [45].

In summary, programmable switches and NICs have distinct cost-to-performance ratios, deployment scenarios, and performance characteristics. In Bedrock, we have targeted switches as our primary scenario. However, we believe that both programmable switches and NICs are important platforms for security applications, and the specific choice would depend on the deployment requirements.

## 5 Related Work

**RDMA security**. RDMA systems have gained popularity in cloud datacenters [24, 60, 61], and their security implications have been recently studied in a series of work [46, 49, 52, 54, 56]. ReDMArk [46], in particular, has described a comprehensive range of RDMA vulnerabilities under varied attack models. Many of the problems we address are from this project. sRDMA [52] has developed cryptographic authentication and encryption of RDMA packets on programmable NICs, whereas Bedrock considers a complementary set of defenses. Bedrock's main target is programmable switches, but as P4 is a target-independent language, it can be deployed to programmable NICs as well. Bedrock inherits the performance characteristics of the underlying platforms.

**Programmable switches.** Programmable switches have found applications in many cloud applications [35, 50, 67], and more recently in TCP/IP security [23, 34, 41, 58, 63, 65]. Bedrock is inspired by such work, but it develops security support for RDMA systems. As these systems offload their datapath operations to hardware, the design decision of using programmable network support respects the performance goals of RDMA while providing stronger security.

## 6 Conclusion

We have presented Bedrock, a defense system that provides a secure foundation for RDMA systems. RDMA systems bypass server CPUs, achieving high performance; but at the same time, security problems are much harder to mitigate as software defenses cannot be easily added in the datapath.

Bedrock leverages programmable data planes in modern network devices to build CPU-bypassing defense primitives for authentication, access control, and monitoring and logging. Using a set of comprehensive experiments, we have shown that Bedrock can effectively mitigate many attacks, and that it incurs low overheads with realistic workloads.

## 7 Acknowledgments

We thank the anonymous reviewers for their helpful feedback on this work. This work was partially supported by NSF grants CNS-1942219 and CNS-1801884.

## References

[1] Alibaba builds high-speed RDMA network for AI and scientific computing. https://www.alibabacloud.com/blog/alibaba-builds-high-speed-rdma-network-for-ai-and-scientific-computing_594895.

[2] Availability of Linux RDMA on Microsoft Azure. https://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-available/.

[3] Bedrock source code. https://github.com/alex1230608/Bedrock.

[4] eBPF official website. https://ebpf.io/.

[5] Edgecore wedge 100bf-32x 32-port 100gbe p4 switch price. https://colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3485.

[6] Experimental analysis of state-of-the-art RDMA-based in-memory key-value stores. https://github.com/mashemat/local-key-value.

[7] Infiniband Architecture Volume 1 and Volume 2. https://www.infinibandta.org/ibta-%20specifications-download/.

[8] Infiniband verbs performance tests. https://github.com/linux-rdma/perftest.

[9] Intel Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[10] Intel tofino p4-programmable ethernet switch asic that delivers better performance at lower power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[11] Netronome agilio cx 40gbps smartnic price. https://colfaxdirect.com/store/pc/viewPrd.asp?idproduct=2871.

[12] Open networking foundation: P4. https://opennetworking.org/p4/.

[13] The P4 language repositories. https://github.com/p4lang.

[14] RDMA communication manager. https://linux.die.net/man/7/rdma_cm.

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

[16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proc. SIGMETRICS*, 2012.

[17] R. Beltman, S. Knossen, J. Hill, and P. Grosso. Using P4 and RDMA to collect telemetry data. In *Proc. INDIS*, 2020.

[18] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, volume 13, 1970.

[19] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3), 2014.

[20] Z. Cao, S. Dong, S. Vemuri, and D. H. Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proc. FAST*, 2020.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. SoCC*, 2010.

[22] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, Apr. 2005.

[23] T. Datta, N. Feamster, J. Rexford, and L. Wang. SPINE: Surveillance protection in the network elements. In *Proc. FOCI*, 2019.

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

[25] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. SOSP*, 2015.

[26] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, F. Feng, Y. Zhuang, F. Liu, P. Liu, X. Liu, Z. Wu, J. Wu, Z. Cao, C. Tian, J. Wu, J. Zhu, H. Wang, D. Cai, and J. Wu. When cloud storage meets RDMA. In *Proc. NSDI*, 2021.

[27] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom. Crystal: Software-defined storage for multi-tenant object stores. In *Proc. FAST*, 2017.

[28] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *Proc. NSDI*, 2017.

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

[30] Z. He, D. Wang, B. Fu, K. Tan, B. Hua, Z.-L. Zhang, and K. Zheng. MasQ: RDMA for virtual private cloud. In *Proc. SIGCOMM*, 2020.

[31] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proc. SIGCOMM*, 2014.

[32] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *Proc. ATC*, 2016.

[33] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *Proc. OSDI*, 2016.

[34] 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.

[35] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. SOSR*. ACM, 2016.

[36] T. H.-J. Kim, C. Basescu, L. Jia, S. B. Lee, Y.-C. Hu, and A. Perrig. Lightweight source authentication and path validation. In *Proc. SIGCOMM*, 2014.

[37] J. Kwon, T. Lee, C. Hähni, and A. Perrig. SVLAN: Secure & scalable network virtualization. In *Proc. NDSS*, 2020.

[38] M. Legner, T. Klenze, M. Wyss, C. Sprenger, and A. Perrig. EPIC: Every packet is checked in the data plane of a path-aware Internet. In *Proc. USENIX Security*, 2020.

[39] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proc. SOSP*, 2017.

[40] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an rdma-enabled distributed persistent memory file system. In *Proc. ATC*, 2017.

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

[42] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proc. SIGCOMM*, 2017.

[43] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and network in the cell distributed B-tree store. In *Proc. ATC*, 2016.

[44] M. Poke and T. Hoefler. Dare: High-performance state machine replication on RDMA networks. In *Proc. HPDC*, 2015.

[45] Y. Qiu, J. Xing, K.-F. Hsu, Q. Kang, M. Liu, S. Narayana, and A. Chen. Automated SmartNIC offloading insights for network functions. In *Proc. SOSP*, 2021.

[46] B. Rothenberger, K. Taranov, A. Perrig, and T. Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *Proc. USENIX Security*, 2021.

[47] M. Sabbagh, Y. Fei, T. Wahl, and A. A. Ding. Scadet: a side-channel attack detection tool for tracking prime+probe. In *Proc. ICCAD*, 2018.

[48] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *Proc. OSDI*, 2016.

[49] A. K. Simpson, A. Szekeres, J. Nelson, and I. Zhang. Securing RDMA for high-performance datacenter storage systems. In *Proc. HotCloud*, 2020.

[50] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *Proc. ATC*, 2018.

[51] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas. Crail: A high-performance I/O architecture for distributed data processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.

[52] K. Taranov, B. Rothenberger, A. Perrig, and T. Hoefler. sRDMA: efficient NIC-based authentication and encryption for remote direct memory access. In *Proc. USENIX ATC*, 2020.

[53] B. Tian, J. Gao, M. Liu, E. Zhai, Y. Chen, Y. Zhou, L. Dai, F. Yan, M. Ma, M. Tang, J. Lu, X. Wei, H. H. Liu, M. Zhang, C. Tian, and M. Yu. Aquila: A practical usable verification system for production-scale programmable data planes. In *Proc. SIGCOMM*, 2021.

[54] S.-Y. Tsai, M. Payer, and Y. Zhang. Pythia: remote oracles for the masses. In *Proc. USENIX Security*, 2019.

[55] S.-Y. Tsai and Y. Zhang. Lite kernel RDMA support for datacenter applications. In *Proc. SOSP*, 2017.

[56] S.-Y. Tsai and Y. Zhang. A double-edged sword: Security threats and opportunities in one-sided network communication. In *Proc. HotCloud*, 2019.

[57] F. Vano-Garcia and H. Marco-Gisbert. Kaslr-mt: Kernel address space layout randomization for multi-tenant cloud systems. *Journal of Parallel and Distributed Computing*, 137:77–90, 2020.

[58] L. Wang, H. Kim, P. Mittal, and J. Rexford. Programmable in-network obfuscation of traffic. *arXiv preprint arXiv:2006.00097*, 2020.

[59] X. Wei, R. Chen, and H. Chen. Fast RDMA-based ordered key-value store using remote learned cache. In *Proc. OSDI*, 2020.

[60] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proc. OSDI*, 2018.

[61] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proc. SOSP*, 2015.

[62] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: Scaling graph computation to the trillions. In *Proc. SoCC*, 2015.

[63] J. Xing, Q. Kang, and A. Chen. NetWarden: Mitigating network covert channels while preserving performance. In *Proc. USENIX Security*, 2020.

[64] J. Xing, W. Wu, and A. Chen. Architecting programmable data plane defenses into the network with FastFlex. In *Proc. HotNets*, 2019.

[65] J. Xing, W. Wu, and A. Chen. Ripple: A programmable, decentralized link-flooding defense against adaptive adversaries. In *Proc. USENIX Security*, 2021.

[66] J. Yang, Y. Yue, and K. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *Proc. OSDI*, 2020.

[67] N. Yaseen, J. Sonchack, and V. Liu. Synchronized network snapshots. In *Proc. SIGCOMM*, 2018.

[68] Y. Zhu, D. Firestone, C. Guo, J. Padhye, S. Raindel, M. Zhang, Y. Liron, H. Eran, M. H. Yahia, and M. Lipshteyn. Congestion control for large-scale RDMA deployments. In *Proc. SIGCOMM*, 2015.