Antikernel: A Decentralized Secure Hardware-Software Operating System Architecture

Andrew Zonenberg[1] and Bülent Yener[2]

[1] IOActive Inc., Seattle WA 98105, USA,
andrew.zonenberg@ioactive.com
[2] Rensselaer Polytechnic Institute, Troy NY 12180, USA,
yener@cs.rpi.edu

Abstract. The "kernel" model has been part of operating system architecture for decades, but upon closer inspection it clearly violates the principle of least required privilege. The kernel is a single entity which provides many services (memory management, interfacing to drivers, context switching, IPC) having no real relation to each other, and has the ability to observe or tamper with all state of the system. This work presents *Antikernel*, a novel operating system architecture consisting of both hardware and software components and designed to be fundamentally more secure than the state of the art. To make formal verification easier, and improve parallelism, the Antikernel system is highly modular and consists of many independent hardware state machines (one or more of which may be a general-purpose CPU running application or systems software) connected by a packet-switched network-on-chip (NoC). We create and verify an FPGA-based prototype of the system.

Keywords: network on chip · system on chip · security · operating systems · hardware accelerators

1 Introduction

The Antikernel architecture is intended to be more, yet less, than simply a "kernel in hardware". By breaking up functionality and decentralizing as much as possible we aim to create a platform that allows applications to pick and choose the OS features they wish to use, thus reducing their attack surface dramatically compared to a conventional OS (and potentially experiencing significant performance gains, as in an exokernel).[3]

Antikernel is a decentralized architecture with no system calls; all OS functionality is accessed through message passing directly to the relevant service. To create a process, the user sends a message to the CPU core he wishes to run it on. To allocate memory, he sends a message to the RAM controller. Each

[3] This paper is based on author 1's doctoral dissertation research [1].

of these nodes is self-contained and manages its own state internally (although nodes are free to, and many will, request services from other nodes).

There is no “all-powerful” software; all functionality normally implemented by a kernel is handled by unprivileged software or hardware. Even the hardware is limited in capability; for example the flash controller has no access to RAM owned by the CPU. By formally verifying the isolation and interprocess communication, we can achieve a level of security which exceeds even that of a conventional separation kernel: even arbitrary code execution on a CPU grants no privileges beyond those normally available to userspace software. Escalation to “ring 0” or “kernel mode” is made impossible due to the complete lack of such privileges; unprivileged userspace runs directly on “bare metal”.

Thus Antikernel architecture unifies two previously orthogonal fields - hardware accelerators and operating system (OS) security - in order to create a new OS architecture which can enforce OS security policy at a much lower level than previously possible. In contrast to the classical OS model, our system blurs or eliminates many of the typical boundaries between software, hardware, kernels, and drivers. Most uniquely, there is no single piece of software or hardware in our architecture which corresponds to the kernel in a classical OS. The operating system is instead an emergent entity arising out of the collective behavior of a series of distinct hardware modules connected via message passing, which together provide all of the services normally provided by a kernel and drivers. Each hardware device includes state machines which implement low-level resource management and security for that particular device, and provides an API via message passing *directly to userspace*. Applications software may either access this API directly (as in an exokernel [2]) or through server software providing additional abstractions (as in a microkernel).

By decentralizing to this extent, and creating natural chokepoints for dataflow between functional subsystems (as in a separation kernel [3], [4]), we significantly reduce the portion of the system which is potentially compromised in the event of a vulnerability in any one part, and render API-hooking rootkits impossible (since there is no syscall table to tamper with). In order to avoid difficult-to-analyze side channels between multiple modules accessing shared memory, we require that all communication between modules take place via message passing (as in a multikernel [5]). This modular structure allows piecewise formal verification of the system since the dataflow between all components is constrained to a single well-defined interface.

Unlike virtualization-based separation platforms (such as Qubes [6]), our architecture does not require massive processing and memory overhead for each security domain, and is thus well suited to running many security domains on an embedded system with limited resources. Our architecture also scales to a large number of mutually untrusting security domains, unlike platforms such as ARM TrustZone ([7]) which provide one-way protection of a single domain.

We have tested the feasibility of the architecture by creating a proof-of-concept implementation targeting a Xilinx FPGA, and report experimental results including formal correctness proofs for several key components. The prototype is open source [8] to encourage verification of our results and further research.

2 Related Work

There are many examples in the literature of operating system components being moved into hardware[4] however the majority of these systems are focused on performance and do not touch on the security implications of their designs at all.

Fundamentally, any hard-wired OS component has an intrinsic *local* security benefit over an equivalent software version - it is physically impossible for software to tamper with it. This brings an unfortunate corollary - it cannot be patched if a design error, possibly with security implications, is discovered. Extremely careful testing and validation of both the design and implementation is thus required. Furthermore, hardware OSes may not provide any *global* benefits to security: If the hardware component does not perform adequate validation or authentication on commands passed to it from software, compromised or malicious software can simply coerce the hardware into doing its bidding. Next we briefly review some of the related work in this domain.

2.1 Security Agnostic Hardware Accelerations

Several researchers implemented hardware accelerators for various RTOS functions: [9] proposes a distributed OS built into a network-on-chip, or NoC; [10] proposes a basic RTOS which contains a simple hardware microkernel implementing a scheduler, semaphores, and timers; [11] describes a microkernel-based OS using a 2D mesh NoC. Each node is a CPU with a microkernel on it, running user processes and/or servers. [12] proposes a "hardware OS kernel", or HOSK, which is connected to a conventional (unmodified) RISC processor and functions as an accelerator. [13] describes BORPH, an operating system for a reconfigurable platform containing one or more CPUs and one or more reconfigurable components such as FPGAs. It introduces the concept of a "hardware process", which is functionally equivalent to a conventional OS process.

While these approaches provide significant performance benefits compared to a software-only implementation, there are no authentication or protection capabilities built into them, thus they provide no security benefits.

2.2 Security-Focused Designs

[14] presents an FPGA-based implementation of a separation kernel. It describes a distributed OS based on a "time-triggered network on chip" (TTNoC) connect-

[4] This paper uses the term "hardware OS" to refer to a series of state machines implemented in silicon which provide operating system services to a computer. Some other authors use the same term to refer to a very different concept: a component of an operating system (which is typically implemented in software) responsible for managing partitions of an FPGA or other reconfigurable computing device.

ing a series of IP cores, each considered a separate partition within the system. While the TTNoC provides complete and deterministic isolation between hosts (i.e., no traffic sent by any other host can ever impact the ability of another to communicate and thus there are no timing / resource exhaustion side channels) it suffers from the lack of burst capabilities and does not scale well to systems involving a large number of hosts (in a system with N nodes each one can only use 1/N of the available bandwidth).

[15] describes a "zero-kernel operating system" or ZKOS. The general guiding principles of "no all-powerful component", "hardware-software codesign", and "safe design" are very similar to our work, as well as the conclusion that privilege rings are an archaic and far too coarse-grained concept. The main difference is that their system relies on "streams" (point-to-point one-way communications links) and "gates" (similar to a syscall vector, allows one security domain to call into another) for IPC and does not support arbitrary point-to-point communication. Furthermore, while threading and message passing are implemented in hardware, the ZKOS architecture appears to be primarily software based with minimal hardware support and does not support hardware processes/drivers. Finally, BiiN [16] was the result of a joint Intel-Siemens project to develop a fault-tolerant computer, which could be configured in several fault-tolerant modes including paired lock-step CPUs. A capability-based security system is used to control access to particular objects in memory or disk. The system architecture advocates heavy compartmentalization with each program divided up as much as possible, and using protected memory between compartments (although the goal was reliability against hardware faults through means such as error correcting codes and lock-stepped CPUs, not security against tampering). No mention of formal verification could be found in any published documentation.

3 Antikernel Network Architecture

At the highest logical level, an Antikernel-based system consists of a series of nodes (userspace processes or hardware peripherals) organized in a quadtree[5] and connected by a packet-switched NoC with 16-bit addressing.[6] Hardware and software components are indistinguishable to developers and are addressed using the same message passing interface.

Each bottom-level leaf node is assigned a /16 subnet (a single address) and corresponds to a single hardware module. The next level nodes are routers for /14

[5] The choice of a quadtree was made purely for convenience of prototyping. Other implementations of the Antikernel architecture could use an octree, 2D grid, add direct sibling-to-sibling links to reduce load on the root, or use more esoteric topologies depending on system requirements.

[6] For the remainder of this paper, NoC routing addresses are written in IPv6-style hexadecimal CIDR notation. For example the subnet consisting of all possible addresses is denoted 0000/0, 8002/16 is a single host, etc. The architecture can be scaled to larger address sizes in the future if needed, however it is unlikely that more than 65536 unique IP cores will be present in any SoC in the near future and smaller addresses require less FPGA resources.

subnets, followed by routers for /12 subnets, and so on. Routers are instantiated as needed to cover active subnets only; if there are only four nodes in the system the network will consist of a single top-level router with four children rather than an eight-level tree. Nodes may also be allocated a subnet larger than a /16 if they require multiple addresses: perhaps a CPU with support for four hardware threads, with each thread as its own security domain, would use a /14 sized subnet so that the remainder of the system can distinguish between the threads.

"The network" is actually two parallel networks specialized for different purposes, as shown in Fig. 1. The RPC network transports fixed-size datagrams consisting of one header word and three data words, and is optimized for low-latency control-plane traffic. The DMA network transports variable size datagrams, and is optimized for high-throughput data-plane traffic. Each node uses the same address on both networks to ensure consistency, although individual nodes are free to only use one network and disable their associated port on the other (for example, node "n8002/16"). Entire routers for one network or the other may be optimized out by the code generator if they have no children (for example, there is no RPC router for the subnet 8004/14 as all nodes in that subnet are DMA-only).

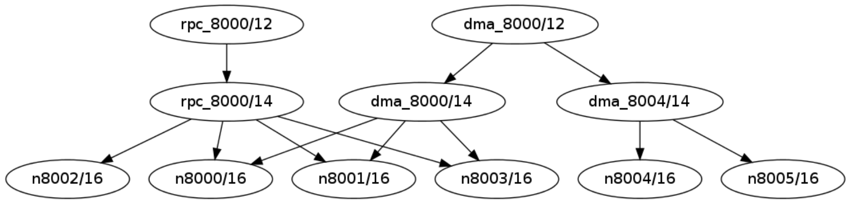

Fig. 1. Example routing topology showing RPC and DMA

A full link for either network contains two independent unidirectional links, each consisting of a 32-bit data bus[7] and several status flags. Since the prototype is FPGA-based all network links are system-synchronous, however they could fairly easily be converted to source-synchronous for a globally-asynchronous/locally synchronous (GALS) ASIC clocking structure.

Packets for both networks begin with a single-word layer-2 routing header containing the 16-bit source and destination node addresses, followed by protocol-specific layer-3 headers.[8] Each network guarantees strict FIFO ordering, as well as reliable delivery, for any two endpoints.

[7] Links could potentially scale to 64, 128, or larger multiples of 32 bits if higher bandwidth is needed, however our prototype does not implement this.

[8] There is no layer-2 header field to distinguish RPC and DMA traffic; since the networks are physically distinct the protocol can be trivially determined from context. Alternate implementations of the architecture could potentially merge both protocols into a single network with an additional header to specify the protocol.

3.1 Remote Procedure Call (RPC)

An RPC message consists of the standard layer-2 routing header followed by an ID indicating the operation to be performed ("call"), a type field ("op"), and the message payload.

Any RPC transaction involves two nodes. The node which initiates the transaction is designated the *master*; the other is designated the *slave*. These roles are not fixed and any node may choose to act as a master or slave at any time.

Interrupts The simplest kind of RPC transaction is an *interrupt*[9]: a unidirectional notification from master to slave. RPC interrupts are typically sent to inform the slave that some long-running operation (such as a backgrounded DMA write to slow flash memory) has completed at the master, or that an external event took place (such as an Ethernet frame arriving, or a button pressed). The `call` field of an interrupt packet is set to a value chosen by the master describing the specific type of event; the data fields may or may not be significant depending on the specific master's application-layer protocol.

Function Calls The second major kind of RPC transaction is a function call: a request by the master that the slave take some action, followed by a result from the slave. This result may be either a success/fail return value indicating that the remote procedure call completed, or a retry request indicating that the slave is too busy to accept new requests and that the call should be repeated later.

The `call` field of a function call packet is set to a slave-dependent value describing one of 256 functions the master wishes the slave to perform. The meaning of the data fields is dependent on the slave's application-layer protocol.

A return packet (including a retry) must have the same `call` value as the incoming function call request to allow matching of requests to responses. The meaning of the data fields is dependent on the slave's application-layer protocol.

Although not implemented by any current slaves, the RPC call protocol allows out-of-order (OoO) transaction processing (handling multiple requests in the most efficient order, rather than that in which they were received).

Flow Control / Routing The RPC protocol will function over links with arbitrary latency (and thus register stages may be added at any point on a long link to improve timing), however a round-trip delay of more than one packet time will reduce throughput since the transmitter must block until an ACK arrives from the next-hop router before it can send the next packet. We plan to solve this issue with credit-based flow control in a future revision.

The RPC router is a full crossbar which allows any of the five ports to send to any other, with multiple packets in flight simultaneously. Each exit queue

[9] Note that the term "interrupt" was chosen because these messages convey roughly the same information that IRQs do in classical computer architecture. While the slave node is free to interrupt its processing and act on the incoming message immediately, it may also choose to buffer the incoming message and handle it later.

maintains a round-robin counter which increments mod 5 each time a packet is sent. In the event that two ports wish to send out the same port simultaneously, the port identified by the counter is given max priority; otherwise the lowest numbered source port wishing to send wins. This ensures baseline quality of service (each port is guaranteed 20% of the available bandwidth) while still permitting bursting (a port can use up to 100% of available bandwidth if all others are idle).

3.2 Direct Memory Access (DMA)

Packet Structure and Semantics A DMA packet consists of the standard layer-2 routing header followed by a type field, data length, and a 32-bit address indicating the target of the DMA operation. This is then followed by message content (up to 512 32-bit words in our current prototype) for "read data" and "write data" packets. The "read request" packet has no data field.

All write operations must be to an integral number of 32-bit words; byte masking is not supported (although it could potentially be added in the future by using some of the reserved bits in the DMA header). If byte-level write granularity is required this is typically implemented with a read-modify-write.

Since the DMA address field is 32 bits, a maximum of 4 GB may be addressed within a single device (/16 subnet). Nodes requiring $>$ 4 GB of address space may be assigned to larger subnets; for example a SD card controller might use a /14 subnet (4 routing addresses) to permit use of cards up to 16 GB (4x 4 GB). When sending pointers between nodes it is necessary to send both the 16-bit routing address and the 32-bit pointer. The resulting 48-bit physical address uniquely identifies a single byte of data within the system.

Memory Read/Write A memory read transaction consists of one packet from master to slave, with the type field set to "read request", the address set to the location of the data being requested, and the length set to the number of words being read. If the request is permitted by the slave's security policy, it responds with a packet of type "read data". Address and length are set as in the requested packet, and the data field contains the data being returned. If the request is not permitted by policy, the slave returns an error code so that the master knows no response is forthcoming.

A memory write consists of one packet from master to slave, with the type field set to "write data" and length/address set appropriately. If the request is permitted by the slave's security policy, it responds with a "write complete" RPC interrupt. This allows the master to implement memory-fencing semantics for interprocess communication: to avoid potential race conditions, one cannot send the pointer to another node (or change access controls on it) until the in-flight write has completed. If the request is not permitted, the slave returns an RPC error interrupt so that the master knows the underlying physical memory has not been modified.

Flow control and routing The current DMA flow control scheme expects a fixed single-cycle latency between routers with lock-step acknowledgement[10]. The DMA router uses the same arbiter and crossbar modules as the RPC router, although the buffers are somewhat larger.

4 Memory Management

One of the most critical services an operating system must provide is allowing applications to allocate, free, and manipulate RAM. In the minimalistic environment of an exokernel there is no need for an OS to provide sub-page allocation granularity, so we require nodes to allocate full pages of memory and manage sub-page regions (such as for C's `malloc()` function) in a userspace heap. If a block larger than a page is required, the node must allocate multiple single pages and map them sequentially to its internal address space.

Antikernel's memory management enforces a "one page, one owner" model. Shared memory is intentionally not supported, however data may be transferred from one node to another in a zero-copy fashion by changing ownership of the page(s) containing the data to the new user.

The Antikernel memory management API is extremely simple, in keeping with the exokernel design philosophy. It consists of four RPC calls for manipulating pages ("get free page count", "allocate page", "free page", and "change ownership of page") as well as DMA reads and writes. A "write complete" RPC interrupt is provided to allow nodes to implement memory fencing semantics before `chown()`ing a page.

The data structures required to implement this API are extremely simple, and thus easy to formally verify: a FIFO queue of free pages and an array mapping page IDs to owner IDs. When the memory subsystem initializes, the FIFO is filled with the IDs of all pages not used for the internal metadata and the ownership array records all pages as owned by the memory manager.

Requesting the free page count simply returns the size of the free list FIFO. Allocating a page fails if the free list is empty. If not, the first page address on the FIFO is popped and returned to the caller; the ownership records are also updated to record the caller as the new owner of the page. Freeing a page is essentially the allocation procedure run in reverse. After checking that the caller is the owner of the page, it is zeroized to prevent any data leakage between nodes, then pushed onto the free list and the ownership records updated to record the memory manager as the new owner of the page. Changing page ownership does not touch the free list at all; after verifying that the caller is the owner of the page the ownership records are simply updated with the new owner.

DMA reads and writes perform ownership checks and, if successful, return or update the contents of the requested range. The current memory controller API requires that all DMA transactions be aligned to 128-bit (4 word) boundaries, be a multiple of 4 words in size, and not cross page boundaries.

[10] We plan to extend this in the future in order to support variable latency for long-range cross-chip links, as was done for RPC.

The current prototype codebase contains two compatible implementations of the Antikernel memory management API: `BlockRamAllocator` (backed by on-die block RAM, parameterizable size) and `NetworkedDDR2Controller` (backed by DDR2, currently fixed at 128MB capacity with a 16-bit bus). A parameterizable-depth allocator backed by DDR3 or QDR-II+ is planned, but has not yet been implemented.

Since Antikernel's architecture is inherently NUMA, multiple memory controllers may be instantiated without causing problems as long as full 48-bit pointers are used to avoid ambiguity.

5 SARATOGA Processor and Threading

The prototype CPU for Antikernel (named SARATOGA) is a high-performance dual-issue in-order barrel processor using a modified version of the the MIPS-1 instruction set with an 8-stage pipeline[11] and a parameterizable number of hardware threads.[12] This produces the net effect of 8 virtual CPUs at 1/8 the core clock rate, time-sharing the same two execution units.

The CPU can easily reach around 180 MHz on a Xilinx Artix-7 FPGA (-2 speed), and can be pushed to 200 with careful floorplanning of the L1 caches and register file. Area is 5700 flipflops, 6400 LUTs, 2570 slices, and 44 RAMs for the CPU itself. The reference system is 12800 flipflops, 15100 LUTs, 5900 slices, and 77 RAMs.[13] This includes the standard Antikernel infrastructure (name server, JTAG debug bridge, system info core), as well as a packet sniffer observing the CPU's RPC uplink to aid in system bring-up.

In addition to allocating one NoC address per hardware thread, the CPU has a dedicated management address used for out-of-band control functionality. This allows applications to request services from the CPU (for example starting a new process, quitting, or modifying their page table). The first address of the CPU's subnet is used as the OoB management address. This ensures that any node which queries the name server for the hostname of the processor will get the management address. The high half of the subnet is used for thread addresses; all other addresses in the low half of the subnet are unused and incoming packets are dropped.

5.1 Thread Scheduler

The processor begins in the idle state; the run queue (a circular linked list) is empty, with no threads running. A free-list of thread IDs is initialized to contain every valid thread ID, and a bitmap of thread IDs is initialized to the "unallocated" state. When a "create new process" message is received by the

[11] The pipeline has two stages of instruction fetch, two of decode/register fetch, and four of execution.

[12] Any power of two ≥ 8 is legal; the default for synthesis is 32.

[13] These numbers are for the default configuration with 32 threads, 2-way cache associativity, 16 lines of 8 words per cache bank.

CPU on its management address the free list is popped, the bitmap is updated to reflect that this thread ID is allocated, and the thread ID is now available for use (but is not yet scheduled). A simple hardware state machine loads the statically linked ELF executable at the provided physical address, initializes the thread, and requests that the scheduler append it to the run queue.

During execution, the CPU reads the current thread from the linked list and schedules it for execution if possible, then goes on to the next thread in the linked list the following cycle. If the thread is already in the pipeline (which may be true if less than 8 threads are currently runnable) then it waits for one cycle and tries again. If the thread is not in the run queue at all (which may be true if the thread was just canceled, or if no threads are currently runnable), then the CPU goes to the next thread and tries again the next cycle.

To delete a thread, it is removed from the linked list and pushed into the free list, and the bitmaps are updated to reflect its state as free. The linked-list pointers for the deleted thread are not changed; this ensures that if the CPU is about to execute the thread being deleted it will correctly read the "next" pointer and continue to a runnable thread the next clock cycle. (There are no use-after-free problems possible due to the multi-cycle latency of the allocate and free routines; by the time the freshly deleted thread can be reallocated the CPU is guaranteed to have continued to a runnable thread.)

The architecture allows for a thread to very quickly remove itself from the run queue without terminating (although the thread management API does not currently provide a means for doing this). This will allow threads blocking on IO or an L1 cache miss to be placed in a "sleep" state from which they can quickly awake, but which does not waste CPU time.

5.2 Execution Units

SARATOGA has two execution units connected to separate ports on the register file. Both are copies of the same Verilog module however some functionality is left unconnected (and thus optimized out) in execution unit 1. Each execution unit takes in two values from the register file during EXEC0, and outputs one value to the register file during EXEC3.

During EXEC0, unit 0 may dispatch an RPC send or receive, or a memory transaction. Results from RPC receives (if data is available), as well as memory operations (in case of an L1 cache hit) are available during EXEC2. All ALU operations other than integer division complete by EXEC1.

5.3 L1 Cache

The L1 cache for SARATOGA is split into independent I- and D-side banks, and is fully parameterizable for levels of associativity, words per line, and lines per thread. The default configuration is 2-way set associative and 16 lines of 8 32-bit words, for a total size of 1KB instruction and 1KB data cache per thread (plus tag bits) and 32KB overall. The cache is virtually addressed and there is no coherency between the I- and D-side caches.

The current cache is quite small per thread, which is likely to lead to a high miss rate, but this is somewhat made up for by the ability of multithreading to hide latency - if all 32 threads are active, a 31-cycle miss latency can be tolerated with a penalty of only one skipped instruction. We have not yet implemented performance counters for measuring cache performance; after this is added there are likely to be numerous optimizations to the cache structure.

5.4 MMU

In order to speed prototyping a very simple MMU was created, consisting of a software-controlled TLB with no external page tables. It supports a parameterizable number (the default is 32) of 2KB pages of virtual memory per thread, mapped consecutively starting at virtual address 0x40000000. Each thread has a fully independent virtual address space, meaning that the total amount of virtual memory addressable by all threads combined in the default configuration is 32*32 pages, or 2 MB. This has been sufficient for initial prototyping; a full MMU with a TLB and external page tables in RAM is planned for the future. Since software accesses the MMU using an abstracted API via the CPU management port, it is possible to make arbitrary changes to the internal MMU and TLB structure without breaking software compatibility.

Each page table entry consists of a valid bit, R/W/X permission flags, a 16-bit node ID, and a 21-bit upper address within that node (low bits are implicit zero). This allows the full 48 bits of physical address space to be used.

5.5 RPC Network Interface

In order to send an RPC message, the high half of the `a0` register is loaded with the "send" opcode; the low half of `a0`, as well as `a1`, `a2`, and `a3`, store the RPC message. This is identical to the standard C calling convention for MIPS, which makes implementation of the `syscall()` library function trivial. (The high half of `a0` is used as the opcode since this would normally be the source address of the packet, but this is added by hardware). A `syscall` instruction then actually sends the message.

Receiving an RPC message is essentially the same process in reverse. The high half of `a0` is loaded with the "receive" opcode and a `syscall` instruction is executed. When a message is ready, it is written to `v1`, `v0`, `k0`, and `k1`. This places the success/fail code and the first half-word of the return data in `v0`, typically used for integer results in the MIPS C calling convention.

Since a new application starting up on a SARATOGA core does not necessarily know the management address of its host CPU, we provide a means for doing so through the `syscall` instruction. At any time, an application may perform a `syscall` with the high half of `a0` set to "get management address" to set the `v0` register to the current CPU's management address. All other CPU management operations are accessed via RPCs to the management address.

5.6 ELF Loader with Code Signature Checking

To create a new process, a node sends a “create process” call to the CPU’s OoB management port, specifying the physical address of the executable to run. The management system begins by allocating a new thread context, returning failure if all are currently in use.

If a thread ID was successfully obtained, the ELF loader then issues a DMA read for `sizeof(Elf32_Ehdr)` bytes to the supplied physical address, expecting to find a well formed ELF executable header. If the header is invalid (wrong magic numbers, incorrect version, or not a big-endian MIPS executable file) an error is returned.

If the header is well formed, the loader then looks at the `e_entry` field to find the address of the program’s entry point. This is fed into a FIFO of data to be processed by the signature engine.[14] It is important to hash headers, as well as the contents of all executable pages, in order to ensure that a signed application cannot be modified to start at a different address within the code, potentially performing undesired actions.

The loader then checks the `e_phoff` field to find the address of the program header table, which stores the addresses of all segments in the program’s memory image. It loops over the program header table and checks the `p_type` field for each entry. If the type is `PT_LOAD` (meaning the segment is part of the loadable memory image) then the loader reads the contents of the segment and feeds them into the hashing engine and stores the virtual and physical addresses in a buffer for future mapping. If the type is 0x70000005 (an unused value in the processor-defined region of the ELF program header type specification) then the segment is read into a buffer holding the expected signature. After all loadable segments have been hashed, the signature is compared to the expected value. If they do not match an error is returned and the allocated thread context is freed.

If the signature is valid, the list of address mappings is then fed to the MMU. Note that the ELF loader is the only part of the processor which has permission to set the `PAGE_EXECUTE` permission on a memory page; permissions for pages mapped by software through the OoB interface are ANDed with `PAGE_READ_WRITE` before being applied. This means that it is impossible by design for any unsigned code to ever execute as long as the physical memory backing the executable cannot be modified externally (for example, by modifying the contents of an external flash chip while the program is executing). With appropriate choices of access controls for on-chip memory, and use of encryption to prevent tampering with off-chip memory, this risk can be mitigated. After the initial memory mappings are created the program counter for the newly created thread is set to the entry point address from the ELF header and the thread is added to the run queue.

[14] We used HMAC-SHA256 in the prototype due to FPGA capacity limitations, as well as difficulty finding a suitable open source public key signature core. An actual ASIC implementation would presumably use RSA or ECC signatures.

5.7 Remote Attestation

SARATOGA supports a simple form of remote attestation. When an application is loaded by the ELF loader, the signature is stored in a buffer associated with the thread ID. At any time in the future, any NoC node may ask the CPU (via RPC to the management interface) to return the signature associated with a given thread context.

6 Security Analysis

6.1 Threat Model

Antikernel's primary goal is to enforce compartmentalization between user-space processes, and between user-space and the operating system. The focus is on damage control, rather than preventing initial penetration. The attacker is assumed to be remote so physical attacks are not considered. Existing antitemper techniques can, of course, be used along with the Antikernel architecture to produce a system with some degree of robustness against physical tampering; but it is important to note that no physical security is perfect and an attacker with unrestricted physical access to the system is likely to be able to penetrate any security given sufficient time and budget.

Antikernel is designed to ensure that following are not possible given that an attacker has gained unprivileged code execution within the context of a user-space application or service: (i) download a backdoor payload and configure it to run after system restart, (ii) modify executable code in memory or persistent storage, intercept/spoof/modify system calls or IPC of another process, (iii) read or write private state of another process, or (iv) gain access to handles belonging to another process by any means.

We consider an abstract RTL-level model of the system with ideal digital signals in which it is not possible for the state of one register or input pin to observe or modify the state of another except if they are connected through combinatorial logic in the RTL netlist.[15]

6.2 Methodology and Goals

We have performed fairly extensive verification on the current prototype system using a mix of simulation, hardware-in-loop (HiL) testing on our test cluster, and formal methods. All tests are fully automated and re-run before every commit. The general verification methodology begins by creating at least one HiL test

[15] In practice it is sometimes possible for this property to be violated (for example by DRAM read disturbance, as described in [17], [18]). Such attacks exploit subtle layout-vs-schematic (LVS) mismatches which are not picked up by automated tools. While detecting these bugs is certainly an important task, it requires a level of solid-state physics better suited to a journal of electrical engineering so we leave it as an open research problem and focus on the computer science problem: ensuring safety of the pre-layout netlist.

for each module or subsystem being verified, supplemented by simulation tests in some cases to speed the design cycle. The focus of this level of verification is catching obvious bugs that occur when the module is used as intended. 100% of the modules in the project receive at least this level of verification.

In addition, the most critical subsystems are provably verified against a formal model of the desired behavior. The choice of modules to verify is determined by several factors including their importance to the security model (the worse the impact of a bug, the more important provable correctness is) and their complexity (simpler modules are easier to prove correct).[16] The Verilog source for formal/simulation testbenches, as well as the C++ test cases and top-level modules for HiL testing, is included under the "tests" directory of the source distribution.

6.3 Assumptions

All of the low-level proofs of correctness were performed on post-synthesis RTL netlists using `yosys` ([19]). We assume correctness of the temporal induction proof system in `yosys` and the SAT solver.[17] In other words, we assume that if the post-synthesis netlist is inconsistent and one or more of the assertions in the netlist are violated, that the solver will correctly detect the error and declare the proof to not hold.

These proofs are only valid down to the RTL level for the current prototype. The actual synthesis and place-and-route (PAR) of the prototype systems were performed using Xilinx's proprietary tools; correctness of these tools and the FPGA silicon is assumed.[18]

It is assumed that the RPC network consists of a series of `RPCv2Router` objects connected in a quadtree, with nodes under the routers. All of these nodes must connect to the RPC network with either an `RPCv2RouterTransceiver` or an `RPCv2Transceiver` object, configured as a leaf node (with the exception of the multithreaded CPUs such as SARATOGA, which are treated as multiple nodes under one router for the scope of the network-level proofs).

The DMA network is assumed to consist of a series of `DMARouter` objects connected in a quadtree, and nodes under the routers. Each node must connect to the DMA network with a `DMATransceiver` or `DMARouterTransceiver` object, configured as a leaf node. As with the RPC network, multithreaded CPUs are a special case and handled separately.

[16] While full verification of the entire implementation is of course desirable, and a goal we are working toward, it would require many man-years of additional effort. Additionally, several components of the design are still being optimized and improved, making a correctnesss proof of the current code a waste of time.

[17] MiniSAT by default, although different solvers can be configured at run time.

[18] Since the FPGA microarchitecture is undocumented, equivalence checking on the actual FPGA bitstream would not be possible without extensive reverse engineering of the silicon. While an interesting problem, and one that researchers including author 1 are actively working on [20], it is beyond the scope of this paper.

Furthermore, it is assumed that if any node connects to both networks it uses the same address on each, and that there is no information flow between nodes outside of the NoC (for example, by wires that do not pass through a NoC router, or off-die paths on the printed circuit board).

All of the test SoCs created as part of this paper were generated by our `nocgen` tool, which is intended to enforce these requirements for the top-level module, however its correctness has not yet been proven. Verifying that any particular generated source file (and the instantiated modules) meet these requirements is relatively easy to do by inspection. In the future we intend to create a DRC tool which uses `yosys` to parse the actual RTL source for a particular SoC and verifies that all of the on-chip topology requirements are met.

6.4 Networks

All four combinations of RPC and DMA transceivers (node or router at each end) for a layer-2 link were formally verified using `yosys`.[19]

Each test case instantiates one transmitter and one receiver of the appropriate types, as well as testbench code. `yosys` is then run on each testbench to synthesize to RTLIL intermediate representation, followed by invoking the SAT solver to prove the assertions in the testbenches. If the solver declares that all assertions pass, the proof is considered to hold.

While the testbenches are all slightly different due to the differences in interface between router/client transceivers and RPC/DMA network protocols, their basic operation is the same. When the test starts, all outputs are in the idle state and remain so in the absence of external stimuli. When a transmit is requested, the test logic stores the signals at the transceiver's inputs and asserts that the same data exits the receiver a fixed time later. The test also verifies that attempts to transmit while the receiver is busy block until the receiver is free (thus preventing dropped packets) and that the transceiver fully resets to its original state after sending a packet.

It is also necessary to prove that packets are correctly forwarded to the desired layer-3 destination by routers. We can map the quadtree directly to routing addresses by allocating two bits of the address to each level of the tree. Each router simply checks if the high bits match its subnet, forwards out the downstream port identified by the next two bits if so, and otherwise forwards out the upstream port. It is easy to see by inspection that this algorithm will always lead to a correct tree traversal.

[19] It is important to note that due to the large maximum packet size (512 words) it was not possible to run the DMA network proofs to a steady state, thus the proof is not complete. The current proof is artificially limited to examining state for the first 64 cycles and shows that no assertions are violated during this time. Running the solver on each proof takes about ten minutes on a single CPU core and uses between three and ten gigabytes of RAM; given a sufficiently large amount of CPU time and RAM there is no reason why the proof cannot be extended until a steady state is reached.

Since correct routing at the hop level combined with a valid quadtree topology implies correct routing at the network level, and the previous proofs show that link-layer forwarding is correct, the proof for correct end-to-end forwarding thus reduces to showing that the router correctly implements the routing algorithm, which is shown by another of our proofs (for the RPC network only). [20]

6.5 Name Server

Even if the layer-3 links between nodes are secure and packet misrouting is impossible, if a rogue node can trick a target node into sending its traffic to the wrong address, by causing the name server to report incorrect data, then MITM attacks can still occur.

Avoiding this requires proving two properties: First, the name server must always return the correct entry (if one exists) from its table when queried, or an error if none exists. Second, the name server must only insert names into the table, or remove them, if authorized by system security policy.

The top-level `NOCNameServer` module consists of several RAM blocks, an RPC transceiver, an HMAC-SHA256 engine, a "target matching" system (which compares outputs of the RAMs against a value being searched for), a mutex, and the main control state machine.

We assume correctness of the RAM and prove correctness of the transceiver separately. The mutex, target matching logic, and HMAC cores have undergone conventional validation but do not have correctness proofs as of this writing. Several correctness properties have been proven on the control state machine, as described below.

We currently have a partial liveness proof on the name server, which shows that two of the RPC calls will always terminate in constant time and return the name server to the idle condition. It also shows that these two calls will always behave as specified by the formal model, and will never modify any other state. A liveness proof for the remaining opcodes in progress however it was not complete as of this writing.

We have verified correct operation of the name server's registration and lookup functionality via conventional verification techniques, including automated unit testing, but have not yet completed a formal correctness proof.

Names for hardware nodes that are "baked" into the name table at logic synthesis time require no further authentication since the source code of the SoC

[20] Aside from the transceivers, the majority of the DMA network router is identical to that of RPC, instantiating the same modules with the same configuration. The only changes were adding an additional SRAM buffer and multiplexer for each port since the DMA transceiver has separate memory channels for headers and packet bodies, as opposed to the single channel for RPC. We believe that these changes are sufficiently non-intrusive that the probability of them containing a security-critical bug is very low. Although a full part-wise verification of the router (as was done for RPC) is certainly possible, and should be performed before an Antikernel system is actually deployed in a critical application, we believe that doing so at this stage of development would be an inefficient use of research time.

is trusted implicitly (and an attacker has no way to modify these addresses short of an invasive silicon attack). Names being registered by a random NoC node at run time, however, are not inherently trusted. In order to prevent malicious name registrations, the name server requires a cryptographic signature to be presented and validated before the name can be registered.

6.6 RAM controller

There are two implementations of the RAM controller API, `BlockRAMAllocator` and `NetworkedDDR2Controller`. Both are covered by an extensive conventional (non-formal) verification suite in the current codebase. In order to ensure interoperability, the same compiled test binary is run on bitstreams containing both RAM controller implementations and is verified to work properly with both.

7 Conclusions and Future Work

The overall goal of this research was to determine whether moving operating system functionality into hardware is a practical means for improving operating system security. We define a high-level architecture, Antikernel, for an operating system which freely mixes hardware and software components as equal peers connected by a packet-switched network. The architecture takes the ideal of "least required privilege" to the extreme by having each node in the network be a fully encapsulated system which manages its own security policy, and only allows access to its internal state through a well-defined API.

The architecture draws inspiration from numerous existing operating system architectures, such as the microkernel (minimal privileged functionality with most services in userspace), the exokernel (drivers as very thin wrappers around hardware providing nothing but security and sharing), and the separation kernel (enforcing strong isolation between processes except through a defined interface).

Additionally, the modular structure of an Antikernel system is highly amenable to piecewise formal verification. If we define security of the entire system as the condition where all security properties of each node are upheld, we can then prove security by proving security of the interconnect, as well as proving that every node's security policy is internally consistent (in other words, policy cannot be violated by sending arbitrary messages to the NoC interface or any external communications interfaces).

We hope that this work will serve to inspire future research at the intersection of computer architecture and security, and lead to more convergent full-stack design of critical systems. Blurring the lines between hardware and software appears to be a promising architectural model and one warranting further study. By releasing all of our source code we hope to encourage future work building on our design. We intend to continue actively developing the project.

While the current prototype does show that hardware-based operating systems are practical and can be highly secure, it is far from usable in real-world applications. Many features which are necessary in a real-world operating system could not be implemented due to limited manpower so effort was focused on the most critical core features such as memory and process management.

The current prototype relies on the initialization code starting all software applications in the same order (and thus receiving the same thread ID since these are allocated in FIFO order every boot). A more stable system for binding processes to IDs is, of course, desirable.

As of this writing, neither of the memory controller implementations have been formally verified. No part of the CPU (other than the NoC transceivers) has been formally verified to date. While SARATOGA's architecture was designed to minimize the risk of accidental data leakage between thread contexts, until full verification is completed we cannot rule out the possibility that such a bug exists. Eventually we would like to verify that the CPUs themselves correctly implement the semantics of our reduced MIPS-1 instruction set. If we then compiled our application code with a formally verified C compiler (such as CompCert C [21], [22]) we could have full equivalency proofs from C down to RTL.[21] This could then be combined with verification of the C source code, resulting in fully verified correct execution from application software all the way down to RTL.

Finally, our prototype is intended to be a proof of concept for hardware-based compartmentalization at the OS level. As a result, we do not incorporate any of the numerous defensive techniques in the literature for guarding against physical tampering, hardware faults, or software-based exploits targeting userland. Currently, implementation of many useful subsystems (such as the networking stack and filesystem) are missing major features or entirely absent. Although many of the core components (such as the NoC) have been formally verified, many higher-level components and peripherals have received basic functional testing only and the full system should be considered research-grade. Further work could explore integrating existing software-based mitigations with Antikernel.

The prototype prioritizes ease of verification and implementation over performance: for example, the SARATOGA CPU uses a simple barrel scheduler which has poor single-threaded performance, lacks support for out-of-order execution, and has very unoptimized logic for handling L1 cache misses. Although these factors combine to cause a significant (order of magnitude) performance reduction compared to a legacy system running the same ISA, these are due to implementation choices rather than any inherent limitations of the architecture. We conjecture that a more optimized Antikernel implementation could match or even exceed the performance of existing OS/hardware combinations due to the streamlined, exokernel-esque design.

Additionally, although backward compatibility with existing operating systems was explicitly *not* a design goal, we have done a small amount of work on a POSIX compatibility layer. This is unlikely to ever reach "recompile and run" compatibility with legacy software due to inherent architecture differences, but we hope that it will help minimize porting effort.

[21] The current CompCert compiler does not support the MIPS instruction set - only x86, ARM, and PowerPC. We plan to explore adding formally verified MIPS code generation to this or another verified C compiler in the future.

References

1. A. D. Zonenberg, "Antikernel: A decentralized secure hardware-software operating system architecture," Ph.D. dissertation, Rensselaer Polytechnic Institute, 2015.
2. D. R. Engler *et al.*, "Exokernel: an operating system architecture for application-level resource management," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, pp. 251–266, Dec. 1995.
3. J. M. Rushby, "Design and verification of secure systems," in *Proc. 8th ACM Symp. Operating Sys. Principles*, 1981, pp. 12–21.
4. W. Martin, P. White, F. S. Taylor, and A. Goldberg, "Formal construction of the mathematically analyzed separation kernel," in *Automated Software Eng., 2000. Proc. ASE 2000. 15th IEEE Int. Conf.*, 2000, pp. 133–141.
5. A. Baumann *et al.*, "The multikernel: A new os architecture for scalable multicore systems," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Principles*, New York, NY, USA, 2009, pp. 29–44.
6. J. Rutkowska and R. Wojtczuk. (2010, Jan.) *Qubes OS Architecture.* [Online]. Available: http://files.qubes-os.org/files/doc/arch-spec-0.3.pdf
7. ARM Ltd. (2014) *TrustZone Technology.* [Online]. Available: http://www.arm.com/products/processors/technologies/trustzone.php (Accessed 2015-04-09).
8. A. Zonenberg. (2016, Mar. 18) *Antikernel source repository.* [Online]. Available: http://redmine.drawersteak.com/projects/achd-soc/repository (Accessed 2016-03-18).
9. M. Engel and O. Spinczyk, "A radical approach to network-on-chip operating systems," in *System Sciences, 2009. HICSS '09. 42nd Hawaii Int. Conf.*, Jan. 2009, pp. 1–10.
10. S. Nordstrom *et al.*, "Application specific real-time microkernel in hardware," in *Real Time Conference, 2005. 14th IEEE-NPSS*, Jun. 2005, p. 4.
11. W. Hu, J. Ma, B. Wu, L. Ju, and T. Chan, "Distributed on-chip operating system for network on chip," in *Computer and Information Technology (CIT), 2010 IEEE 10th Int. Conference on*, Jul. 1 2010, pp. 2760–2767.
12. S. Park *et al.*, "A hardware operating system kernel for multi-processor systems," *IEICE Electron. Express*, vol. 5, no. 9, pp. 296–302, 2008.
13. H. Kwok-Hay So *et al.*, "A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph," in *CODES+ISSS '06: Proc. 4th Int. Conf. Hardware/Software Codesign Syst. Synthesis*, 2006, pp. 259–264.
14. A. Wasicek *et al.*, "A system-on-a-chip platform for mixed-criticality applications," in *Object/Component/Service-Oriented Real-Time Distributed Comput. (ISORC), 2010 13th IEEE Int. Symp.*, May 2010, pp. 210–216.
15. A. Thomas *et al.* (2013, Jan 10) *Towards a Zero-Kernel Operating System.* [Online]. Available: http://www.infsec.cs.uni-saarland.de/ hritcu/publications/zkos_draft_jan10_2013.pdf (Accessed 2015-04-09).
16. BiiN Corporation. (1988, Jul) *BiiN Systems Overview.* Portland, OR. [Online]. Available: http://bitsavers.informatik.uni-stuttgart.de/pdf/biin/BiiN_Systems_Overview.pdf (Accessed 2015-04-09).
17. Y. Kim *et al*, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Comput. Architecture (ISCA), 2014 ACM/IEEE 41st Int. Symp.*, Jun 2014, pp. 361–372.

18. C. Evans. (2015, Mar. 9) *Project Zero: Exploiting the DRAM rowhammer bug to gain kernel privileges.* [Online]. Available: http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html (Accessed 2015-04-09).
19. C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/.
20. A. Zonenberg. (2015, Jul. 22) *From Silicon to Compiler: Reverse-Engineering the Xilinx XC2C32A*. [Online]. Available: https://recon.cx/2015/slides/recon2015-18-andrew-zonenberg-From-Silicon-to-Compiler.pdf (Accessed 2016-03-02).
21. S. Blazy *et al.*, "Formal verification of a C compiler front-end," in *FM 2006: Int. Symp. Formal Methods*, vol. 4085, 2006, pp. 460–475.
22. S. Boldo *et al.*, "A formally-verified C compiler supporting floating-point arithmetic," in *ARITH, 21st IEEE Int. Symp. Comput. Arithmetic.* IEEE Computer Society Press, 2013, pp. 107–115.