DASICS-Security Processor Design White Paper Abstract This white paper systematically elaborates on the DASICS security processor architecture, a hardware-assisted security extension technology designed to defend against memory safety vulnerabilities and control-flow hijacking attacks. By introdu...

Authors: Jin Yue, Yang Chengyuan, Xu Yibin, Lu Tianyue, Chen Mingyu, Chen Mingyu

Date: 2023-04-18T20:27:27+00:00

Abstract

The open-source, shared, and collaborative software development model has spurred the flourishing development of fields such as the Internet and artificial intelligence. However, this model has also led to increasing complexity in software development, manifested in reliance on numerous developers jointly developing a single software system, frequent invocation of third-party code libraries, and management and maintenance of massive codebases. This complex development model introduces a high probability of security vulnerabilities at the development level. For instance, software developers inevitably need to invoke third-party code libraries yet lack guarantees regarding their security, resulting in exploitable vulnerabilities that pose risks of information leakage and tampering. Once vulnerabilities are discovered in a widely-used third-party library, numerous software systems built upon it are typically affected. Among software security vulnerabilities, memory access vulnerabilities constitute the most critical category. In response to the challenges posed by these vulnerabilities, academia and industry have proposed a series of software and hardware memory protection methods. These approaches employ, on one hand, Data Flow Integrity (DFI) technology to strictly inspect and restrict the data flow of untrusted code, preventing illegal memory operations through out-of-bounds checks or data source compliance verification. Representative works include Intel's MPX and MPK technologies, ARM's MTE technology, and the University of Cambridge's CHERI security architecture. On the other hand, Control Flow

Integrity (CFI) technology prevents malicious control flow hijacking, exemplified by Intel's CET, ARM's BTI, and PA technologies. However, these methods suffer from various limitations, including overly coarse isolation granularity, vulnerability of security metadata to attacks, excessive hardware implementation and performance overhead, and requirements for significant modifications and recompilation of existing third-party code. We propose the DASICS secure processor design to address these issues of coarse isolation granularity, low metadata security, and high performance overhead, while also focusing on dynamic permission partitioning, intra-address-space memory protection, and cross-layer call checking—areas previously underexplored. Our design implements dynamic permission partitioning at the code snippet granularity, providing hardware-assisted efficient software memory protection, ensuring secure invocation and execution of third-party code, and offering solid security guarantees for open-source software development.

Full Text

Preamble

DASICS - Secure Processor Design White Paper
Version: v1.0.0
Institute of Computing Technology, Chinese Academy of Sciences
Advanced Computer Systems Laboratory, System Architecture Group

6 DASICS Application Scenario Examples

6.3 Use Case 3: Single-Level OS (LibraryOS)

The open-source, shared, and collaborative software development model has fueled the prosperity of fields such as the Internet and artificial intelligence. However, this model has also introduced increasing complexity in software development, manifested in reliance on large numbers of developers working on a single software project, frequent invocation of third-party code libraries, and the management and maintenance of massive codebases. This complex development model inevitably introduces security vulnerabilities at the development level. For instance, software developers must inevitably call third-party code libraries but lack guarantees about their security, leading to exploitable vulnerabilities introduced through reliance on untrusted libraries and creating risks of information leakage and tampering. Once a vulnerability is discovered in a widely-used third-party library, the impact often extends to numerous software products built upon it.

Memory access vulnerabilities constitute the most critical category of software security flaws. To address these challenges, academia and industry have proposed a series of hardware-software memory protection methods. These approaches employ Data Flow Integrity (DFI) techniques to strictly check and

restrict the data flow of untrusted code, preventing illegal memory operations through boundary checks or compliance verification of data sources. Representative works include Intel's MPX and MPK technologies, ARM's MTE technology, and the CHERI security architecture led by the University of Cambridge. On the other hand, Control Flow Integrity (CFI) techniques prevent malicious control flow hijacking, such as Intel's CET, ARM's BTI, and PA technologies. However, these memory protection methods suffer from various limitations, including overly coarse-grained isolation objects, vulnerability of security metadata to attacks, excessive hardware implementation or performance overhead, and the need for substantial modifications and recompilation of existing third-party code.

We propose the DASICS secure processor design to address the problems of coarse-grained isolation, insecure metadata, and excessive performance overhead in existing protection technologies. DASICS focuses on aspects previously overlooked: dynamic privilege partitioning, intra-address-space memory protection, and cross-layer call checking. It implements a security processor design based on dynamic privilege partitioning by code segments, providing hardware-assisted efficient software memory protection to ensure secure invocation and execution of third-party code, thereby offering solid security guarantees for open-source software development.

2 Untrusted Third-Party Code Libraries

Many low-level software components (such as operating systems, network protocol stacks) and high-performance third-party libraries (such as SQLite, ASL, Crypto++, Qt) are developed in memory-unsafe C/C++ languages, with numerous third-party libraries containing potential memory access vulnerabilities. Examples include the libxml2 XML parsing library with heap buffer overflow vulnerabilities, the htmlawed HTML filtering library with remote code execution vulnerabilities, and the ClamAV open-source anti-malware toolkit with buffer overflow vulnerabilities. When using these libraries, developers often overlook their unsafe characteristics (such as lack of input boundary checking), resulting in exploitable vulnerabilities in their software that cause severe security issues like information leakage and tampering.

Memory access vulnerabilities represent the primary threat to contemporary software security. These vulnerabilities occur when programs access or modify unauthorized memory regions due to programming errors or malicious attacks, potentially disrupting normal execution, causing system crashes, or enabling exploitation. Common memory access vulnerabilities include buffer overflows, use of uninitialized memory, null pointer dereferences, and heap overflows, which can lead to program crashes, data loss, system failures, and information leakage. Figure 1 [Figure 1: see original paper] presents Google's statistics on difficult-to-fix security vulnerabilities in the Chrome codebase, showing that 36.1% of threats stem from Use-after-Free attacks (primarily caused by incorrect memory management), and 32.9% from overflow attacks and other issues. Approximately

96% of these vulnerabilities are memory access vulnerabilities.

The primary cause of typical memory access vulnerabilities in third-party libraries is the lack of boundary checking on accessible data and control flow verification for untrusted code. For example, the Heartbleed vulnerability (CVE-2014-0160) in OpenSSL versions prior to 1.0.1 occurred when implementing the TLS heartbeat mechanism without validating input lengths (i.e., missing boundary checks), enabling attackers to perform overread buffer attacks against clients/servers and causing widespread information leakage. More severe memory vulnerabilities allow attackers to rewrite instruction execution addresses, transferring program control flow to attacker-controlled code for more flexible and threatening attacks, potentially achieving Turing-complete capabilities.

Since third-party libraries and developer code typically run within the same process address space, vulnerabilities in third-party code can easily compromise overall program security. For programs written in languages like C/C++ without built-in memory safety checks, theoretically every line of code can directly access any location within the process address space without restriction. Once a third-party code vulnerability is exploited, attackers can easily access arbitrary data and code within the process space to steal or modify information or hijack the program for privilege escalation attempts. Furthermore, vulnerabilities in third-party drivers within the operating system kernel can grant access to entire system memory.

Existing research on memory access vulnerability protection falls into two categories: DFI approaches that focus on efficiently partitioning software regions and performing boundary checks to restrict certain code from accessing other code's data (e.g., preventing function A from modifying its caller's local variables via stack overflow), and CFI approaches that restrict jumps, such as detecting when function A's return address differs from the original call site to prevent illegal jumps. Different protection methods exhibit varying characteristics in terms of completeness, usability, and performance overhead, requiring careful trade-offs in practical system design.

3 Related Work Comparison

Defense mechanisms for memory safety issues can be categorized into pure software approaches and hardware-software collaborative methods. Pure software approaches provide protection at programming language, compiler, and loader levels, while hardware-software collaborative methods include constructing trusted execution environments and implementing intra-address-space isolation and control flow protection.

3.1 Pure Software Approaches

Pure software defense methods operate at multiple levels. At the programming language level, rewriting untrusted code in memory-safe languages like Rust

could resolve most memory access vulnerabilities. However, this faces significant practical difficulties, as porting numerous large third-party libraries from C/C++ to safe languages represents a massive engineering effort.

At the compiler level, runtime boundary checking can be achieved by inserting checks into source code. Examples include GCC and clang's Stack Protector technology, which prevents stack attacks by setting a stack guard variable during function calls and checking for tampering upon return, and Microsoft's AddressSanitizer (ASan) compiler plugin, which injects detection code at compile-time to catch memory errors at runtime.

At program loading, Address Space Layout Randomization (ASLR) defends against code-relocation attacks by randomizing the layout of heap, stack, and shared library mappings, making it difficult for attackers to predict target code locations.

Control flow protection primarily relies on CFI technology, which statically analyzes source or binary code to construct a Control-Flow Graph (CFG) and monitors at runtime whether program control flow remains within the CFG. A CFG consists of basic blocks (continuous code without control flow instructions) and directed edges representing control flow transfers.

While pure software methods provide certain security defenses, code instrumentation (e.g., ASan) significantly degrades performance by adding checks before and after memory accesses. Additionally, mechanisms protecting security metadata (e.g., CFG-based control flow checking) must ensure this metadata cannot be tampered with or forged by attackers.

3.2 Trusted Execution Environment

Trusted Execution Environment (TEE) is a hardware-software collaborative security architecture that constructs an isolated secure computing environment through CPU time-sharing or partitioning memory address space. It uses hardware isolation, encryption, or integrity verification to prevent malicious programs from reading or tampering with internal data and control flow from outside, enhancing system security and privacy. TEE has been widely applied in mobile devices and cloud computing, with representative technologies including Intel's SGX, AMD's SEV-SNP, ARM's CCA and TrustZone, and RISC-V's Penglai and Keystone.

Specific TEE technologies depend on particular architectures or platforms, increasing device replacement and software porting costs. To maintain security, TEE requires substantial hardware resources (e.g., Merkle trees for memory integrity), and the overhead of data interaction between TEE-internal and external programs, as well as secure environment creation and switching, is considerable. Therefore, TEE is suitable for relatively independent small-scale security-critical code but less appropriate for scenarios requiring frequent interaction. While theoretically feasible to run large applications entirely within

TEE, isolation between multiple modules within large programs, particularly for third-party libraries, is not addressed by TEE functionality.

3.3 Address Space Security Isolation

Beyond external isolation protection like TEE, internal software threats (e.g., untrusted third-party libraries) also require protection. Without internal code isolation where all code shares identical privileges and can access any data and jump to any code within the address space, untrusted code could steal private data (e.g., user passwords) or jump to malicious attack code. Therefore, software internals also require privilege partitioning and isolation of access and jump regions based on different code components.

Software internal protection technologies must partition privileges for internal code segments while restricting memory access and jump targets for untrusted third-party libraries to prevent out-of-bounds access to private data and control flow hijacking to malicious code. These technologies can be classified into data access isolation and control flow protection.

Figure 2 [Figure 2: see original paper] illustrates threats from untrusted libraries within software.

3.3.1 DFI Methods Data Flow Integrity (DFI) techniques partition and isolate data accessible to different code segments within software, performing access checks based on isolation rules to prevent out-of-bounds access and illegal operations, thereby blocking theft and tampering of sensitive data. DFI methods can be further divided into pointer-based permission attachment and memory data-based permission attachment.

Pointer-based permission techniques like MPX and CHERI define each pointer's access range and permitted operations on pointed-to data, performing checks during memory access via pointers.

MPX (Memory Protection Extensions) is Intel's hardware extension for preventing memory access vulnerabilities. It adds four 128-bit bound registers, four boundary-setting/comparison instructions, four bound register move instructions, two configuration registers, and one status register. MPX checks pointer references at runtime to determine if they exceed preset ranges. Due to limited bound registers, additional pointer bound metadata must be stored in a memory table. This technology requires recompiling all third-party libraries, incurs substantial software modification costs, and suffers severe performance degradation from code instrumentation for every memory access instruction. Consequently, Intel removed MPX from x86 extensions in 2019.

CHERI (Capability Hardware Enhanced RISC Instructions) is another hardware-software collaborative memory protection technology that extends pointer semantics from address-length to 128-bit capabilities containing valid range, permissions, and validity information, with dedicated instructions for

modifying capability information. CHERI enables pointer-level memory safety protection, theoretically defending against most overflow attacks. However, its extended pointers incur significant memory access overhead and require extensive compiler modifications and third-party library recompilation.

Memory data-based permission techniques like MTE and PKU isolate data by tagging memory regions, with code holding these tags to define access permissions for corresponding memory areas.

MTE (Memory Tagging Extension), introduced in ARMv8.5-A, uses tags to divide memory into 64-byte chunks with unique tags for each. These tags are used with pointers to verify correct memory locations. MTE performs memory protection using hardware tags without code modifications and provides software tagging for custom tag schemes, offering multiple tag width options to balance tag quantity and overhead.

PKU (Protection Key Unit) is a lightweight memory isolation mechanism represented by Intel's Memory Protection Key (MPK) technology, which uses 4 extra bits in page table entries to "color" memory regions at page granularity. MPK adds a special register (PKRU) to record the current program's key, detecting illegal access when a page doesn't permit the current key. MPK also adds special instructions for PKRU manipulation, achieving memory isolation with minimal hardware overhead. However, MPK lacks systematic security mechanisms to protect PKRU from tampering, requiring binary inspection of library functions to ensure no PKRU-modifying instructions exist. Additionally, MPK supports only 16 keys, enabling only 16 colorings simultaneously, which is insufficient for complex programs with extensive isolation requirements.

3.3.2 CFI Methods Control Flow Integrity (CFI) originally required constructing a complete program control flow representation (e.g., CFG), but practical implementations adopt concise and efficient methods to protect critical control flow data (e.g., function pointers, return addresses on stack) and restrict indirect jump targets (e.g., to function entry points only). Major technologies include CET, BTI, and PA.

CET (Control-flow Enforcement Technology) is Intel's hardware-level security technology that effectively prevents control flow hijacking and tampering through two mechanisms: - Indirect Branch Tracking (IBT): Defends against JOP (Jump-Oriented Programming) and ROP (Return-Oriented Programming) attacks using indirect jumps by setting landing point instructions to define legitimate indirect jump targets (e.g., function entry addresses). - Shadow Stack (SS): Maintains an additional stack to track function call return addresses, pushing return addresses during calls and comparing them with actual return targets upon function return to detect tampering.

BTI (Branch Target Identification), added in ARMv8.5, similarly defends against JOP attacks by setting BTI instructions at specific positions during

compilation to define indirect jump targets, increasing the difficulty of chaining program fragments through indirect jumps.

PA (Pointer Authentication) is ARM's pointer integrity verification technology that uses cryptographic integrity to protect return addresses, leveraging unused high bits in virtual address pointers. With compiler support, called functions compute authentication codes for return addresses, attach them to pointer high bits (called PAC, Pointer Authentication Code), and push the authenticated return addresses onto the stack. Verification instructions inserted before function returns detect tampering by comparing computed authentication results with PACs, triggering exceptions on mismatch.

3.3.3 Security Metadata Protection Existing address space isolation methods share a common problem: while using security metadata (e.g., MTE tags, PKU keys) for protection, they lack protection for the metadata itself, allowing malicious programs to elevate privileges or disable isolation by modifying security metadata. For example, programs can arbitrarily manipulate PKRU registers using instructions to disable coloring protection in MPK.

Numerous studies have attempted to improve security metadata protection from different perspectives, broadly classified into: - Binary scanning methods: Inspect and rewrite application or untrusted library binaries to ensure no metadata-modifying instructions exist (typically specially designed instructions). For example, the academic ERIM method uses binary scanning to prevent third-party library code from containing WRPKRU instructions that modify PKRU registers. However, this approach cannot support dynamically generated code. - User-mode privileged function methods: Construct a privileged function in user mode dedicated to security metadata management and operations, with any metadata operations outside this function triggering exceptions. The academic Donky system exemplifies this approach, using user-mode interrupts to establish a Domain Monitor for PKRU management, where WRPKRU instructions outside the monitor trigger exceptions. Donky also restricts jumps to the Domain Monitor to prevent control flow hijacking for PKRU modification. This approach resembles further privilege set partitioning within user mode but may incur performance overhead from frequent privilege switching.

Security metadata protection often requires combining DFI and CFI: DFI methods must isolate metadata as private data to prevent untrusted code access and tampering, while CFI methods must ensure that code capable of modifying this private data cannot be hijacked for malicious modification.

3.4 Common Issues in Security Method Design

We summarize the aforementioned protection mechanisms and identify the following practical challenges that memory vulnerability protection mechanisms face when entering practical deployment:

3.4.1 Trade-off between Protection Capability and Overhead Widely-applicable processor security solutions inevitably face trade-offs between performance/hardware overhead and protection capability. On one hand, mechanisms with strong protection capabilities may incur substantial performance overhead from frequent checks (e.g., MPX requires permission table lookups for every pointer access). Protection based on privilege states may require numerous privilege switches, also introducing considerable overhead. On the other hand, additional security metadata in hardware-software collaborative mechanisms may cause substantial hardware overhead (e.g., CHERI's pointer extension increases pointer storage overhead several-fold, causing significant storage and access overhead for pointer-intensive applications). Therefore, security mechanism design must consider not only security but also the cost of achieving it.

3.4.2 Portability Issues Some security mechanisms face rewriting and compilation difficulties when protecting existing third-party code libraries, leading to portability problems. Many academic works achieve this by designing new instructions or extending pointers, which changes program instruction semantics and pointer semantics, causing incompatibility with legacy binary code libraries. Applying these mechanisms requires recompiling all unsafe third-party libraries, which is not only labor-intensive but often impossible for closed-source libraries. Thus, portability must be considered in security mechanism design.

3.4.3 Security Checks for Privilege Switching DFI and CFI generally add fine-grained protection within the same privilege level. However, once privilege switching occurs, higher-privilege code often has permission to modify lower-privilege code and data. Without restrictions on privilege switching, DFI and CFI protection mechanisms could be bypassed through privilege calls. Typical privilege calls include system calls and virtual machine traps. System calls are important interfaces for applications to use OS functionality but can also be exploited by untrusted code to bypass security mechanisms. For example, under MPK protection, code can use the madvise system call (originally for providing kernel memory access information for performance optimization) to clear contents of specific memory pages even when MPK protects them from modification. Code can also use brk and sbrk system calls (originally for heap management) to reclaim and reallocate private heap data.

Therefore, beyond data isolation, boundary checking, and control flow restrictions, security mechanisms should also intercept and filter system calls and other privilege switches to prevent attacks exploiting privilege transitions.

4 DASICS Design Philosophy

Based on the aforementioned challenges in security method design, we aim to design an efficient, low-overhead, portable processor security enhancement technology for intra-address-space applications that simultaneously addresses data flow integrity, control flow integrity, and system call security.

Attack Model: Our secure processor design assumes that software can be divided into code fragments from different sources, including developer-written code (which developers can control and recompile as needed) and third-party library code (which may come from closed-source binaries or dynamically generated code that developers cannot analyze or modify). We assume this latter portion is untrusted, likely containing exploitable memory access vulnerabilities and potentially dangerous operations like privilege escalation through system calls. Attackers can exploit these vulnerabilities for control flow hijacking, data tampering, and theft of critical data.

Code as Subject: To achieve intra-address-space security isolation, we must first address how to distinguish the security subject. Within an address space, traditional processes and threads are unsuitable as security subjects; function libraries as subjects are too coarse-grained; pointers or instructions as subjects are too fine-grained. We observe that code fragments (continuous address ranges of code) within a program serve as appropriate subjects for privilege partitioning. For example, a library function contains many continuous code fragments that typically implement the same functionality. The same code fragment accesses a determined data range and performs determined operations during a specific time period, meaning we can achieve temporal security isolation by spatially partitioning data accessed by different code fragments (e.g., limiting pointer access ranges) and restricting operations (e.g., limiting read/write to specific addresses). We also need to dynamically adjust partitioning and restrictions over time to ensure security isolation for the same code fragment at different times (e.g., when the same function is called multiple times to process data at different locations). Additionally, we must restrict entry and exit points of these code fragments (e.g., functions can only be entered via calls and exited via returns, cannot jump to the middle or exit to non-caller code) to limit control flow.

Therefore, we select code fragments as security subjects—the principle of "Code as Subject."

Dynamic Permissions: Based on the code-as-subject principle, our security mechanism's basic approach is to dynamically set permissible data boundaries and jump targets according to preset or inferred code data access ranges before invoking a code fragment. The code fragment can freely access and jump within allowed data ranges, but any out-of-bounds access or illegal jump triggers an exception to block the operation. Permissions are automatically revoked after the code fragment returns.

Dynamic permission partitioning requirements manifest in several aspects: - Dynamic permission setting and allocation are needed because security resources/metadata are limited while complex programs require isolation and partitioning of numerous code fragments. - Support for different permission settings for the same code fragment at different times is needed (e.g., function A may access private data when called by trusted code but must be prohibited from such access when called by untrusted code). - Support for scenarios that

cannot be statically partitioned is needed (e.g., function pointers where the caller cannot determine the function's specific behavior and accessible data range before invocation).

Built-in Trusted Base: As previously discussed regarding security metadata issues, security resource management requires a trusted core. We need a trusted code base (TCB) to uniformly manage permissions and maintain metadata, which must satisfy basic characteristics: the TCB itself must be guaranteed not to be (or be extremely difficult to) attacked, and efficient transfer mechanisms from TCB to untrusted code are needed to minimize performance overhead. The TCB must: - Allocate and reclaim security resources to support security protection for different code fragments or the same fragment at different times, enabling dynamic partitioning. - Set data access regions and operations for untrusted zones before control flow transfers, ensuring data flow integrity. - Restrict its own entry points to prevent untrusted zones from hijacking control flow to modify security metadata. - Trigger exceptions when untrusted code fragments violate permissions, with options to exit or correct the operation. - Provide a series of functions similar to system calls, offering trusted invocations for operations and data beyond the untrusted code's accessible range.

Regarding TCB implementation, we do not adopt privilege level partitioning within the same address space. We observe that the TCB itself can be managed as a special code fragment. Transfer between TCB and untrusted code does not require complex privilege switching but can directly jump under security rule control. Therefore, the TCB is essentially a code fragment built into the application.

We call this security processor design based on dynamic privilege partitioning by code fragments **Dynamic in-Address-Space Isolation by Code Segments (DASICS)**. Specifically, DASICS comprises several components: region partitioning, code privilege configuration, control flow restriction, permission checking, permission exception handling, main function calls, and system call interception.

4.1 Region Partitioning

DASICS performs privilege partitioning by code fragment, where a code fragment is defined as a continuous executable code set in address space (e.g., code within a non-inline function). Code fragment ranges are defined by start and end addresses. DASICS follows the principle that higher-privilege code fragments can restrict lower-privilege code fragments. As shown in Figure 3 [Figure 3: see original paper], these restrictions operate in two dimensions: - **Vertical restriction**: DASICS supports higher-privilege levels setting restrictions on lower-privilege levels. For example, in RISC-V architecture, M-mode trusted zones can restrict S-mode trusted zones, which can further restrict U-mode trusted zones. - **Horizontal restriction**: DASICS partitions program code within the same privilege state into trusted and untrusted zones, with trusted zones re-

stricting untrusted zones' access permissions. For instance, in user mode, the program's trusted zone is the main function while untrusted zones are third-party library functions, with the main function setting permissions before calling library functions. In kernel mode, the trusted zone is kernel scheduling code while untrusted third-party driver code is placed in untrusted zones, with the kernel setting permissions before using these drivers.

Trusted/untrusted zone partitioning is currently determined through manual address space annotation. DASICS places trusted code fragments into statically designated address spaces during linking to construct trusted zones. All other address space code is considered untrusted. Trusted zones can access all data at the same privilege level, while untrusted zones cannot access trusted zone data, thereby completing access region isolation between different privilege codes.

4.2 Code Permission Partitioning

In DASICS, code fragment permissions are divided into access permissions and jump permissions. Access permissions are defined using data access boundaries, with DASICS adding several sets of boundary registers in hardware, each describing an accessible region's range and permitted operations (read/write). Jump permissions are similarly defined by jump range registers specifying allowed jump target address ranges for library functions.

Before trusted zones call untrusted zone code, they first allocate boundary registers and configure untrusted zone permissions, including memory access and jump permissions, to restrict untrusted zone data flow and control flow. Boundary register resources are reclaimed after untrusted zone returns. For control flow restriction, beyond defining accessible regions via jump range registers, DASICS also limits entry points for jumps from untrusted to trusted zones. Untrusted zones can only jump to trusted zone code through function returns, main function call entries, or same-privilege-level interrupts, preventing trusted zone code from being intercepted and exploited.

4.3 Permission Checking

DASICS access permission checking is implemented in the processor's instruction fetch and memory access units. When untrusted zone code performs memory access (including instruction fetch, read, or write), the access address is checked against boundary registers to verify whether it accesses addresses outside allowed ranges or performs disallowed operations (execute/read/write). Any permission violation (address out-of-bounds or no operation permission) triggers a same-privilege-level interrupt for further handling (e.g., error exit).

Control flow restriction checking occurs during jump instruction execution, primarily verifying control flow when trusted zones call untrusted zone functions. When program control flow transfers from trusted to untrusted zone code via function call, the return address in trusted zone code is recorded. Upon untrusted zone return, the actual return address is compared with the recorded

address; inconsistency indicates potential tampering by untrusted code, triggering an interrupt for error exit.

Based on these permission checks, DASICS can prevent attacks such as buffer overflow and control-flow hijacking through return address modification.

4.4 Same-Privilege-Level Exception Handling

DASICS supports trusted zones registering exception handlers for access or control flow exceptions. When exceptions occur as described above, they trigger same-privilege-level interrupts (e.g., user-mode interrupts in user mode) that jump to exception handlers, which read hardware privilege registers for exception causes and perform different handling such as error exit or dynamic permission correction before resuming original instruction execution. Exception handlers also support dynamic registration to enable different handling for unified exceptions across scenarios.

4.5 Trusted Calls

Similar to OS system calls, DASICS provides trusted calls for untrusted zones to perform trusted operations beyond their accessible range. This addresses cases where pre-configured data ranges are insufficient for untrusted zone functionality. For example, when a library function in user mode needs expanded data access permissions or larger data regions, it can return to the main function via main function call to update boundary registers before resuming library function execution.

To support trusted calls, trusted zones must first register trusted call entry addresses to add legitimate trusted zone entry points. Untrusted zone code can set trusted call parameters and directly jump to these entries, which is faster than system calls as it avoids time-consuming privilege switching.

4.6 System Call Interception

To prevent attacks exploiting cross-privilege calls, DASICS includes system call interception. Hardware monitoring logic captures system call instructions (e.g., RISC-V ecall instructions) and triggers user-mode interrupts to jump to trusted zone exception handlers. Based on exception cause (user-mode system call exception), handlers then jump to system call exception handling functions that can perform security checks, such as verifying whether the code fragment can perform the system call, checking whether system call parameters exceed the code fragment's permission range, and proxying unsafe system calls (copying parameters and having the main function perform the system call on behalf). Performing system call checks within applications enables more targeted fine-grained management of various file and device resources.

4.7 Advantages of DASICS

Compared with previous memory vulnerability protection mechanisms, DASICS offers several advantages: - **Comprehensive protection**: DASICS ensures data flow integrity (DFI) through memory isolation between different privilege subjects and control flow integrity (CFI) through paired checking of function calls and returns. It provides both data flow and control flow protection with mutual support, minimizing the risk of security metadata modification. - **Efficient protection**: Transfers between trusted and untrusted zones in DASICS are direct jumps, avoiding the interrupt-based switching used in mechanisms like Donky and eliminating frequent kernel/monitor mode transitions for complete security protection. - **Fine-grained protection**: DASICS achieves pointer access restrictions through dynamic boundary register-based partitioning, providing finer protection granularity than page-granularity mechanisms like MPK. DASICS code fragments enable finer-grained privilege partitioning without relying on hardware IDs or tags. - **Dynamic protection**: DASICS enables dynamic region partitioning through trusted zone dynamic configuration and dynamic privilege partitioning through allocation and reclamation of security resources (access boundary registers and jump range registers), supporting flexible, scenario-based protection with time-sensitive security rules. - **Multi-privilege-level protection**: DASICS supports multi-level privilege state protection (kernel mode and user mode), applicable to both user-mode code protection (see Use Case 1) and kernel code protection (see Use Case 2). - **Good portability**: Under DASICS protection, users only need to partition third-party libraries into security zones without recompiling them, reducing third-party library recompilation difficulties. When source code modification is possible, simple API additions at function entry/exit points enable further security enhancements.

5 Prototype 1.0 Implementation

Based on the above design philosophy, we implemented DASICS Prototype 1.0, including hardware and software components. The hardware is based on the domestic open-source RISC-V processor NutShell, while the software modifies BBL (Berkeley Boot Loader) and Linux 4.18.2.

5.1 Hardware Implementation

The DASICS hardware component is built on the open-source RISC-V processor NutShell. In NutShell's sequential six-stage pipeline, we added support for DASICS CSRs, DASICS exceptions, and DASICS new instructions, primarily involving the decode module (IDU), CSR execution module, and load-store unit (LSU), with total changes not exceeding 500 lines of Chisel code.

For DASICS CSRs, we added 43 logical 64-bit CSR registers in the CSR module to support DASICS security functions:

DASICS CSR Name	CSR Number	Description
dsmcfg & dumcfg	0xBC0 & 0x5C0	S-mode/U-mode main function configuration registers
dsmbound0 & dsmbound1	0xBC1 & 0xBC2	S-mode main function boundary registers
dumbound0 & dumbound1	0x5C1 & 0x5C2	U-mode main function boundary registers
dlcfg0 & dlcfg1	0x881 & 0x882	Library function configuration registers
dlbound0 … dlbound31	0x883 … 0x8A2	Library function boundary registers
dmaincall	0x8A3	Main function call entry address
dretpc	0x8A4	Library function return address
dretpcfz	0x8A5	Active zone return address

- dsmcfg and dumcfg share one physical CSR register to enable DASICS protection for S-mode/U-mode and set clear bits to reduce software context switch overhead.
- dsmbound0/1 and dumbound0/1 define S-mode and U-mode trusted zone boundaries, with all code outside these ranges considered untrusted.
- dlcfg registers contain eight 4-bit tiny configs per register (odd-indexed tiny configs are all-zero and unused), with each tiny config corresponding to a set of dlbound registers indicating permissions (read/write/execute) for code fragments within that library function boundary. dlcfg and dlbound together form a DASICS library function lookup table for DASICS exception detection.
- dmaincall stores the main function call entry address, to which library function jumps are permitted.
- dretpc records the target address allowed for library function returns. When the main function jumps to a library function via jump or branch instructions, the address of the next instruction is automatically filled into dretpc by hardware.
- dretpcfz is the active zone return address, where an active zone refers to code within untrusted zones that external code can jump to arbitrarily, but which can only return to external code via function returns. This mechanism provides further restrictions for nested untrusted zone calls.

To prevent untrusted zone code from modifying these CSR registers to disable DASICS protection, DASICS imposes new restrictions on CSR access, allowing only trusted zone code to access privilege-state-permitted DASICS CSRs. For

example, dsmbound0/1 can only be read/written by M-mode and S-mode main functions, while dlcfg can only be accessed by M-mode, S-mode main functions, and U-mode main functions.

To enable exceptions upon DASICS access or jump violations and perform system call hijacking, we added eight DASICS exceptions:

DASICS Exception Name	Exception Code	Description
DASICS_U/S_{{{INST}}}_{{{FAULT}}}	0x18 / 0x19	U-mode/S-mode jump exception
DASICS_U/S_{{{LOAD}}}_{{{FAULT}}}	0x1a / 0x1b	U-mode/S-mode no-read permission
DASICS_U/S_{{{STORE}}}_{{{FAULT}}}	0x1c / 0x1d	U-mode/S-mode no-write permission
DASICS_U/S_{{{ECALL}}}_{{{FAULT}}}	0x1e / 0x1f	U-mode/S-mode system call exception

- DASICS S-mode/U-mode instruction exceptions occur when untrusted zone functions jump to trusted zone targets not equal to dretpc or dmaincall, when untrusted active zone functions attempt to jump to addresses beyond return addresses, or when untrusted zone code jumps outside allowed ranges defined by boundary registers.
- DASICS S-mode/U-mode memory access exceptions occur when untrusted zones attempt to access addresses outside permitted ranges (i.e., no matching entry in the DASICS library function lookup table for the access address and read/write flags), triggering corresponding privilege-level read/write exceptions based on whether the instruction is load or store. When such exceptions occur, the memory request being sent to the AXI bus in the LSU module must be canceled.
- DASICS S-mode/U-mode system call exceptions are triggered when the current instruction is a RISC-V system call instruction (ecall), enabling system call interception checks in the exception handler.

In NutShell's instruction decode module, we added decode logic for the new dasicsret instruction. This instruction is designed similarly to JALR but differs in that dasicsret only permits main function calls.

Additionally, to accelerate DASICS exception handling, we implemented the RISC-V N extension draft (version 1.1) on NutShell, enabling delegation of DASICS exceptions to user mode via medeleg/sedeleg configuration. The exception handling flow when DASICS exceptions are delegated to user mode is illustrated in Figure 4 [Figure 4: see original paper].

5.2 Software Implementation

In BBL, we added DASICS mechanism initialization, including setting DASICS exception delegation and configuring dsmbound to cover the entire memory space. The latter is done because: (1) BBL cannot determine S-mode trusted zone size, and (2) after initialization, BBL jumps to S-mode trusted zone, which modifies dsmbound via new SBI calls. We also added SBI calls in BBL for dsmbound modification, which can only be invoked by S-mode trusted zones, otherwise returning an error directly.

In Linux, we added DASICS exception handling logic and applied DASICS to user program ELF loading. For exception handling, we added logic to save and restore DASICS context in Linux's original exception handling flow and delegated DASICS U-mode exceptions to user mode. Meanwhile, we provided default main function call entry functions and default DASICS user-mode exception handlers. For user program ELF loading, we primarily modified fself.c to identify dasics partitions in user programs and fill dumbound registers to set user program main function regions.

6 DASICS Application Scenario Examples

This chapter introduces several possible application examples that efficiently improve system security through DASICS.

6.1 Use Case 1: Trusted Rust + Untrusted C Code Libraries

As previously mentioned, using safe programming languages like Rust can resolve most memory vulnerabilities, but such languages are complex to develop and difficult to port, and rewriting all existing program code in safe languages would be a massive undertaking. With DASICS, developers can build trusted zone code using safe languages while restricting other untrusted library code through DASICS mechanisms. This ensures trusted zone code remains free of memory vulnerabilities while significantly reducing rewriting workload. DASICS mechanisms restrict untrusted zone code permissions within the trusted zone, greatly reducing the harm from potential security vulnerabilities in untrusted zones. DASICS configures security permissions through hardware register operations, independent of programming language, thus supporting various languages and providing flexible security configuration.

6.2 Use Case 2: Kernel Protection + Third-Party Drivers

Many existing security mechanisms rely on a secure kernel to set security permission data for user-mode program checking and restriction. However, kernel mode itself may run large amounts of third-party code, including file systems, device drivers, and network protocol stacks, which are not trusted by OS developers and thus pose security risks. Both third-party modules and kernel code run at privileged levels—once kernel security is compromised, numerous secu-

rity protection mechanisms become ineffective. Through DASICS, we can also introduce trusted/untrusted zone isolation within the operating system. System developers can place third-party module code in kernel-space untrusted zones and restrict their execution access permissions, thereby avoiding OS security risks from third-party kernel modules. DASICS implements security protection mechanisms in hardware, supporting kernel address isolation and protection without breaking original kernel-user isolation while adding new isolation layers within kernel mode.

6.3 Use Case 3: Single-Level OS (LibraryOS)

In cloud computing scenarios, applications often do not require complex OS functionality, while complex operating systems reduce overall system efficiency. Therefore, the LibraryOS or Unikernel concept was proposed, transforming required OS functionality into user-mode libraries for applications, thereby reducing complex OS overhead and improving platform adaptability. However, this approach sacrifices traditional OS security—the isolation between kernel and user modes. To achieve sufficient security, additional lightweight isolation mechanisms must be introduced. If LibraryOS incorporates DASICS functionality, it can restrict user program memory access permissions at function granularity, isolating application code from kernel functionality code within LibraryOS to avoid certain security risks. Compared with protection mechanisms based on multiple protection state switches, DASICS incurs smaller performance overhead, preserving LibraryOS's original performance advantages.

Note: Figure translations are in progress. See original paper for figures.