Orthrus: Efficient Software Integrity Protection on Multi-Cores

Ruirui Huang, Daniel Y. Deng, and G. Edward Suh
Computer Systems Laboratory, Cornell University, Ithaca, NY 14853, USA
{rh335,dyd2}@cornell.edu, suh@csl.cornell.edu

Abstract

This paper proposes an efficient hardware/software system that significantly enhances software security through diversified replication on multi-cores. Recent studies show that a large class of software attacks can be detected by running multiple versions of a program simultaneously and checking the consistency of their behaviors. However, execution of multiple replicas incurs significant overheads on today's computing platforms, especially with fine-grained comparisons necessary for high security. Orthrus exploits similarities in automatically generated replicas to enable simultaneous execution of those replicas with minimal overheads; the architecture reduces memory and bandwidth overheads by compressing multiple memory spaces together, and additional power consumption and silicon area by eliminating redundant computations. Utilizing the hardware architecture, Orthrus implements a fine-grained memory layout diversification with the LLVM compiler and can detect corruptions in both pointers and critical data. Experiments indicate that the Orthrus architecture incurs minimal overheads and provides a protection against a broad range of attacks.

Categories and Subject Descriptors C.0 [*GENERAL*]: Hardware/software interfaces; D.4 [*OPERATING SYSTEMS*]: Security and Protection

General Terms Design, Performance, Security

Keywords Memory protection, Multi-core architecture, Software diversity and redundancy, Replication-aware architecture, Software security

1. Introduction

As we place more responsibilities into computing devices, the secure operation of computers is becoming critical. Yet, today's software programs are far from perfect, often containing various types of vulnerabilities that may be exploited to maliciously alter the program behavior. Moreover, programs are likely to have more types of vulnerabilities that are unknown today. As an example, the format string exploit was first reported in 1999 even though the vulnerability existed for years. The Orthrus system aims to enhance the security of future computing systems by enabling efficient and fine-grained software diversity and redundancy on multi-cores.

In recent years, researchers have shown that the security of a system can be greatly enhanced by introducing automatically generated diversity and redundancy to the system even when the system contains various types of exploitable vulnerabilities. In this approach, a program is automatically transformed into multiple versions with the identical functionality (”in-specification” behavior) but diverse implementation details (”out-of-specification” behavior). At run-time, multiple replicas execute together with the same input and their behaviors are checked to be consistent. An adversary needs to compromise all replicas without causing detectable divergence in order to compromise the system. This approach is particularly attractive because it provides general protection against a large class of known or even unknown attacks while being applicable to legacy code. The recent trend towards large-scale multicore systems also provides an opportunity to deploy the replication techniques.

However, there exists one major obstacle for the replication techniques to be widely deployed in real-world systems. Today, executing N diverse replicas incurs N times overheads in all resources including memory space, on-chip area, power consumption, and bandwidth consumption. In fact, the overheads are far greater if the replicas are compared at a fine granularity for high security. While the exponential growth in the number of transistors with advances in fabrication technologies may alleviate the overheads in on-chip area and main memory space, power consumption and off-chip bandwidth cannot scale with the number of transistors and will continue to be major bottlenecks in modern and future computing systems. As a result, applying a technique with two replicas to a system can reduce the system throughput by half or more, which is often too high a price to pay for systems in the field.

This paper proposes a multi-core extension named Orthrus that supports efficient execution of diversified replicas exploiting the close relationship between replicas. Replicas often share many identical properties such as a control flow and most data because they are automatically generated from one program. As a result, most memory operations and computations of one replica can be quite accurately obtained from those of another replica. In this paper, we introduce a replication-aware memory system that eliminates unnecessary bandwidth overheads by efficiently compressing multiple replicas' memory spaces, and a tightly-coupled core architecture that removes redundant computations along with their energy and on-chip area overheads. In addition to lowering the overheads, the proposed architecture also enables fine-grained comparisons between replicas to enhance security.

To utilize the Orthrus architecture, this paper proposes and studies a memory layout diversification scheme. With the fine-grained checks that are enabled by the architecture, the layout diversification can provide a comprehensive detection for spatial memory errors and can also detect some temporal errors. Experiments with real-world vulnerabilities and test programs demonstrate that the layout diversification on Orthrus can detect a broad range of attacks exploiting vulnerabilities such as buffer overflows, format string bugs, and dangling pointers. The layout diversification with replication can detect attacks that overwrite memory no matter which

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.
Copyright © 2010 ACM 978-1-60558-839-1/10/03... $10.00

vulnerability allows the overwrite as long as they incur discrepancies between two replicas.

The experimental results also demonstrate that the proposed architecture features are quite effective in reducing the replication overheads. The replication-aware memory system reduces the second replica's off-chip bandwidth usage by over 77% on average, and the selective instruction processing eliminates 70% of instructions on average. In fact, with the reduced overheads, the experiments show that an asymmetric core configuration with a wide (4-issue) superscalar core and a single-issue in-order core can execute two replicas with different memory layouts with a 2.7% slowdown on average while maintaining the same protection as the traditional dual-core configuration with two large cores.

The following summarizes the main contributions of this paper:

- *Replication-aware memory system* (Section 3.3): The memory architecture reduces bandwidth overheads by compressing multiple replicas' memory spaces while enabling fine-grained checks.
- *Selective instruction processing* (Section 3.4): Orthrus removes redundant computations between replicas. Experiments show that even a simple single-issue in-order core can match the throughput of a wide (4-issue) superscalar processor, saving energy and on-chip area.
- *Fine-grained memory layout diversity* (Section 4): The paper introduces a diversity technique that can detect both temporal and spatial memory errors while applicable to legacy code.

The rest of the paper is organized as follows. Section 2 describes the diversified replication in general, and Section 3 presents our architecture design to enable efficient replication. Section 4 explains how a fine-grained memory layout diversification scheme can run on our architecture. Section 5 evaluates the proposed architecture, and Section 6 discusses related work. Finally, Section 7 concludes the paper.

2. Software Security Through Diversity and Redundancy

2.1 Security Attack Scope

The main focus of this paper is to detect software attacks that tamper with the program integrity by overwriting program state. More specifically, attacks may tamper with data pointers either by directly injecting a pointer or exploiting bugs in the pointer manipulation such as buffer overflows, overwrite code pointers to change the control flow, or overwrite critical non-control data. While we do believe that the diversity and redundancy can also enhance other aspects of security such as confidentiality and availability, this paper focuses on the integrity of an individual program execution.

We note that the attacks on data pointers, code pointers, and data values can be largely independent even though many known attacks use errors in a data pointer such as a buffer overflow in order to tamper with code pointers or data values. As an example, a format string attack may overwrite a pointer or a data value using an existing data pointer in the stack. Therefore, only protecting data pointers is not sufficient in general.

This paper mainly studies programs written in C or C++ where low-level vulnerabilities are most common. Currently, our system has not been tested with self-modifying code. While we believe that the general technique can be applied to self-modifying code, this paper does not discuss issues related to self-modifying code.

2.2 Diversification for Software Security

Traditionally, most computer systems share identical executable binaries and configurations, which allows one software exploit to work for all instances of a vulnerable program. Software diversity addresses this weakness by randomizing implementation details on each system. Because a successful exploit needs to match the implementation details, this randomization prevents a single exploit to be applied to many systems (see Figure 1(a)). For example, in the stack smashing attack, if each program instance has a different stack layout, an adversary needs to customize an attack in order to successfully overwrite a return address. Many approaches have been proposed in this direction, including randomizing the memory layout [3, 4, 16, 34, 44], relocating code [16], randomizing instruction opcode [2, 23], and changing the name of system calls [8].

However, diversification alone is often not enough to stop a determined attacker from compromising a system. Even when implementation details are obfuscated and initially kept secret, studies show that an adversary, especially for 32-bit machines, can circumvent the randomization through trial and error; the address space randomization can be broken in hundreds of seconds [36] and the instruction set randomization can be compromised in several minutes [39]. The main benefit of the diversification is a slowdown of attacks, but not prevention of attacks.

2.3 Diversified Replication

The security of a computer system can be greatly enhanced by introducing redundancy (replication) within a system in addition to diversity. In this approach, one system executes multiple diversified versions of a program simultaneously as shown in Figure 1(b). Externally, the replicas appear as one program instance; an input to the program is duplicated and fed to all instances, and outputs from multiple replicas are combined into one. During the execution, a system compares the behaviors of replicas such as system calls, memory accesses, and control flows to be consistent; divergence beyond the built-in diversity indicates an error or an attack.

To circumvent this protection method, an attacker must compromise all replicas without incurring detectable divergence in their behaviors. This proposition can be made infeasible for a class of attacks given that *all replicas operate on the same input*. As an example, consider an attack to overwrite a function pointer exploiting a format string vulnerability on the two replicas in Figure 1(c). To compromise Replica 1, an attack can overwrite a pointer at Address `0x200`. However, this malicious input also causes Replica 2 to overwrite a different object at Address `0x200`. An attack cannot overwrite the pointer in both replicas at the same time. Therefore, this attack can be detected if the discrepancy in memory writes can be checked. Note that fine-grained checks at each memory access are critical to ensure the security. Otherwise, an attacker can compromise both replicas in sequence.

The diversified replication can be applied to many program characteristics and provide protection from a broad range of software attacks. The following examples show how different program components can be protected.

- *Data Pointers*: As shown in the above example, the data layout diversity can detect injected pointers or out-of-bound pointers. DieHard [3] and the N-variant system [10] show examples of such layout diversification. A previous study shows that the layout diversity provides protection comparable to that of a type system with bound checks [32].
- *Code Pointers*: Program pointers can be protected by diversifying the code layout. Overwriting a program pointer results in discrepancy in a control flow given that an attack injects the same pointer to all replicas.
- *Critical Data*: Diversification of data can detect attacks that overwrite critical data values. For example, the N-variant system [31] protects user IDs from data corruption attacks by having replicas with different UID representations. Similarly, di-

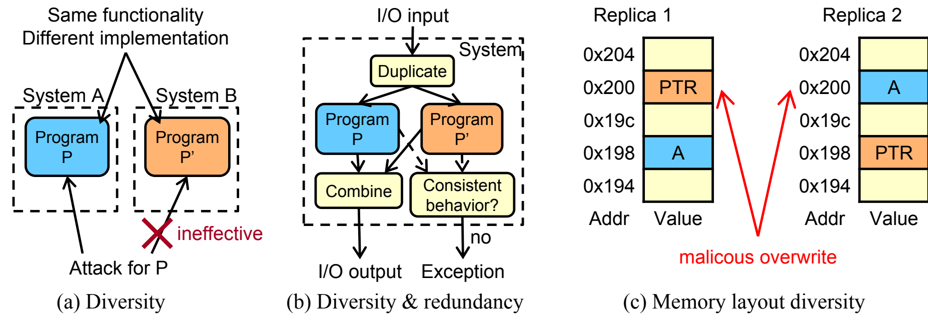


Figure 1. Diversity and redundancy techniques to enhance software security.

versifying system call names prevents attacks that corrupt system calls [8].

2.4 Strengths and Limitations

The diversified replication provides a promising way to protect the integrity of computing systems thanks to its broad attack coverage and applicability to legacy code. The approach checks discrepancies in pointers or data values instead of preventing particular types of vulnerabilities such as buffer overflows or format string errors. This property is particularly attractive because it implies that the diversified replication can detect attacks even if they exploit new types of vulnerabilities that are unknown today as long as the attacks overwrite pointers or data.

In fact, the studies in this paper show that the layout diversification can detect both spatial errors such as buffer overflows and format string vulnerabilities and temporal errors such as certain dangling pointers. Unlike most memory protection schemes such as array bound checks, the diversified replication can even potentially detect memory overwrites by malicious code. For example, a third party library may contain an intentional backdoor that allows an attacker to overwrite a specified address with a specified value. Traditional bound checks cannot prevent such attacks because malicious code can arbitrarily set the bound information and may not even use an array pointer. In the diversified replication, however, the malicious code still cannot compromise multiple replicas at the same time unless the malicious code in each replica can be customized to overwrite a different memory location. Because an attacker can only provide one copy of malicious code, such customization for each replica cannot be done. Effectively, the diversified replication provides fine-grained memory protection in this scenario.

At the same time, the diversification can often be carried out automatically without human intervention or sometimes even without source code. Code and heap layouts can be changed by the linker [10] or the `malloc()` function [3]. Stack layouts or system call names can be changed by a simple re-compilation. Therefore, the diversification approach can often be applied to legacy code or libraries without significant human efforts. Finally, while this paper focuses on detection of attacks, we also note that the replication can potentially enable recovery from attacks because multiple replicas are unlikely to be corrupted in the same way [3].

The main limitation of today's diversified replication approach lies in its overheads. Because each instance of the program essentially acts as if it is a stand-alone process, each replica occupies its own memory space and requires its own processing core unless replicas run in sequence with a two times slowdown. Therefore, the replication scheme with N copies incurs N times overheads in all resources including memory, bandwidth, caches, processing cores, and power consumption. While advances in fabrication technologies will help, the overheads in off-chip bandwidth and power consumption significantly hinder wide deployment of these replication schemes. Moreover, fine-grained comparisons in software often result in prohibitive overheads. As a result, today's replication techniques often rely on checks at system calls, which limits their security - an attacker may be able to compromise one replica at a time with multiple attacks.

3. Orthrus Replication-Aware Architecture

This section describes our multi-core architecture that enables efficient execution of diversified replicas with fine-grained monitoring. The architecture is designed to run two replicas while the same approach may be extended to more replicas. The discussion uses a memory layout diversification as the main example because the scheme is important for a large class of memory errors and incurs significant difference between replicas; arguably, the scheme is the most difficult one to execute efficiently. Section 4 describes the layout diversification in detail.

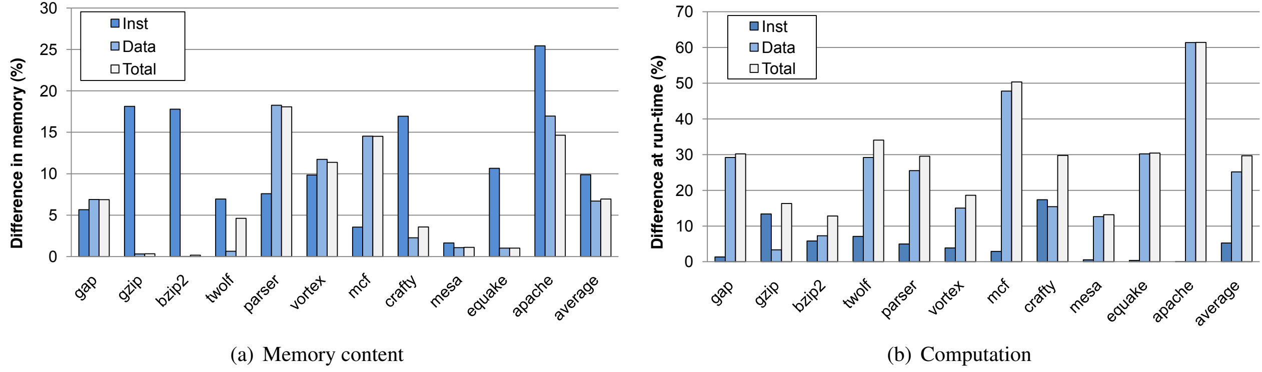
In a high level, the proposed architecture consists of two components. A replication-aware memory subsystem compresses the two replicas' memory spaces into one so that the second copy is used only if it is different from the first one. In this way, the off-chip bandwidth overhead is minimized. Selective instruction processing reduces redundant computations for the second replica and allows the use of a simple in-order core instead of an out-of-order core to minimize the on-chip area and power consumption.

3.1 Opportunity: Similarities in Replicas

Diversified replicas often share lots of common properties together because they are generated from one program. For example, replicas with diversified memory layouts differ only in the memory addresses that they use. All replicas have the same control flow, the same instruction opcodes, and identical data values in memory.

Figure 2 illustrates the similarities between two replicas when the memory layout diversification is applied to Apache web server and a set of SPEC CPU2000 benchmarks [18]. As shown in Figure 2(a), the corresponding memory locations of two replicas[1] often contain an identical value; 9.9% of instructions are different because their immediate field represents a memory address, and 6.7% of data are different because they are pointers. Overall, most corresponding memory locations hold the same value. In the figure, Total shows the differences in the entire memory space whereas Inst and Data only consider code and data, respectively. As a result, the total percentage is not a simple sum of the percentages for Inst and Data. Similarly, Figure 2(b) shows that both replicas perform

[1] Two locations (one from each replica) that contain the same variable or array element of a program.



(a) Memory content (b) Computation

Figure 2. Differences between two replicas in the memory layout diversification.

	Properties	Replica 1	Replica 2	Example: Layout Diversity
Memory	Address	A_{R1}	$MAP[A_{R1}]$	$A_{R2} = MAP[A_{R1}]$
Memory	Value	V_{R1}	Any	V_{R2}= V_{R1} or $MAP[V_{R1}]$
Computation	Control Flow	PC_{R1}	$MAP[PC_{R1}]$	PC_{R2}= $MAP[PC_{R1}]$
Computation	Operation	Op_{R1}	Op_{R1}	$Op_{R2} = Op_{R1}$
Computation	Value	V_{R1}	Any	V_{R2} = V_{R1} or $MAP[V_{R1}]$

(a) SIMDA Model

Diversification Techniques	Diversity Requirement: Memory		Diversity Requirement: Computation			SIMDA enough?
	Addr	Value	Control	Op	Value	
Data Layout	Y	Y	N	N	Y	Y
Code Location	Y	Y	N	N	Y	Y
System Call	N	Y	N	N	Y	Y
Data Value	N	Y	N	N	Y	Y
Instruction	N	N	N	Y	N	N

(b) SIMDA applicability in diverse replication schemes

Table 1. Single Instruction Multiple Data and Address (SIMDA) computation model and its applicability to diversified replication schemes. $MAP[]$ is a mapping between the two address spaces.

an identical operation with identical input operand values in most cases as they are different only for pointer arithmetic operations and jump targets; 5.2% of instructions use different immediate values and 25.1% of instructions have different source register values.

The above experimental results show the similarities between replicas with diverse memory layouts. In general, the same observation can be made for most, if not all, automatic diversification schemes. In fact, we believe that other diversification schemes result in even less differences between replicas. For example, replicas in the data diversity scheme will only be different for the parts that process the protected data, such as UIDs and system call names, which are often very small portions of the entire program.

3.2 Computation Model

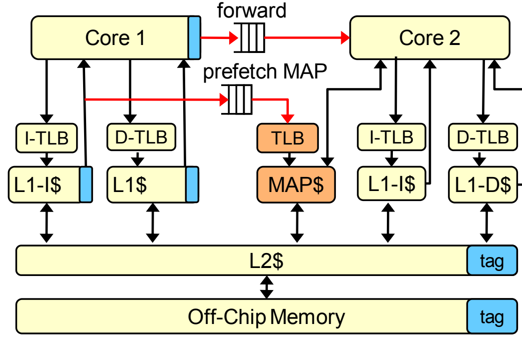
The large amount of similarities in automatically-generated replicas suggest that there is an opportunity to significantly reduce overheads of replication by eliminating unnecessary redundancy. However, dynamically detecting such similarities is quite difficult without explicit annotations if replicas are allowed to diverge in an arbitrary fashion. To effectively exploit the similarities, it is critical to carefully limit the extent of divergence between replicas while allowing enough flexibility to support various diversification techniques.

Fortunately, replicas in our diversification schemes satisfy simple relations because only pointers or data values are changed for integrity protection. Essentially, the replicas share one control flow and perform the same operation instruction-by-instruction. On the other hand, the replicas can diverge in their data values in an arbitrary way and use different memory addresses as long as there exist a one-to-one mapping between their addresses. We call this diversification model as SIMDA (Single Instruction Multiple Data and Address). Essentially, the model is the same as SIMD with additional flexibility in the memory addresses.

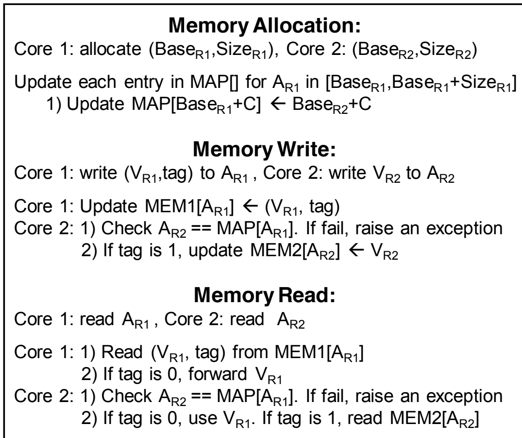
Table 1(a) summarizes the SIMDA model and shows the relationship between two replicas in the memory layout diversification as an example. The model requires that both replicas perform identical operations (opcode) on each instruction ($Op_{R1} = Op_{R2}$). The memory layouts can be different between replicas, but there needs to be a one-to-one mapping $A_{R2} = MAP[A_{R1}]$ between two replicas' memory spaces at any given point of time[2]. Here, A_{R1} and A_{R2} represent the addresses in Replica 1 and Replica 2, respectively. Therefore, data accesses and program counters should satisfy the following conditions: $A_{R2} = MAP[A_{R1}]$ and $PC_{R2} = MAP[PC_{R1}]$. Finally, there is no restriction on memory and register content (values).

As shown in Table 1(b), the SIMDA model can support multiple diversity techniques as long as they do not require divergence in the control flow or the operations. For example, any memory layout diversification scheme [3, 4, 10, 16, 44], system call randomization [8], and data diversification to protect integrity [31] are all supported. We do not support the instruction set randomization [2, 23] that detects injected code because instructions are usually read-only except for self-modifying code. Also, injected instruc-

[2] The mapping can change over time.



(a) Memory system architecture



(b) Memory operation summary

Figure 3. The replication-aware memory system. Dark colors (blue and orange) indicate the additional modules compared to a standard memory system.

tions are harmful only if a control flow can be changed to execute them, which can be prevented with other diversity schemes.

3.3 Replication-Aware Memory System

The replication-aware memory system aims to support two replicas with minimal overheads, especially in the off-chip bandwidth consumption. In the SIMDA model, the memory system must allow the replicas to read and write different values to different addresses as if each replica has its own memory space, `MEM1` for Replica 1 and `MEM2` for Replica 2. The memory system, however, can assume that both replicas always perform the same type of memory operation (read or write) to the same object[3]. In the common case, the corresponding memory operations from the two replicas are likely to use an identical value (see Figure 2(a)). The replication-aware memory system exploits this redundancy and effectively compresses the second memory space (`MEM2`).

Figure 3 illustrates our replication-aware memory system design and its high-level operation. To remove redundant copies between the two replicas, the memory system adds a 1-bit tag for each word in Core 1's memory hierarchy including both L1 caches, the L2 cache, and the main memory. The tag indicates whether both replicas have the same value (tag 0) or not (tag 1). Core 1 also propagates this value tag during its computation by combining (OR) the source tags on each operation. On a read, Replica 1 on Core 1 reads a value and the corresponding tag from its memory space (`MEM1`). These values are forwarded to Core 2. Replica 2 on Core 2 uses the forwarded value from Core 1 for its read operation if the tag is zero, and reads from its own memory space (`MEM2`) if the tag is one. Similarly, on a write, Core 1 writes a value-tag pair to its memory (`MEM1`), but Core 2 performs a write to `MEM2` only if the tag is one.

Because the tags are only kept for each word, the current architecture does not allow store byte instructions, which only update a byte in memory, from diverging in their store values. This constraint is not an issue for typical diversification techniques that protect pointers and critical integer data. If necessary, however, the architecture can be modified to maintain per-byte tags and support value diversification at any granularity. A previous work on Dynamic Information Flow Tracking (DIFT) [40] showed that a similar tag scheme at a byte-granularity incurs a slowdown less than 0.1% in the worst case even when tags share the memory bus with regular data. The DIFT implementation uses a simple compression scheme to reduce the size of tags. The same technique can also be applied to Orthrus. For simplicity, this paper uses per-word tags and dedicated tag bits in the off-chip bus and memory as in Raksha [12]. Therefore, the tag bits do not incur any slowdown in our implementation.

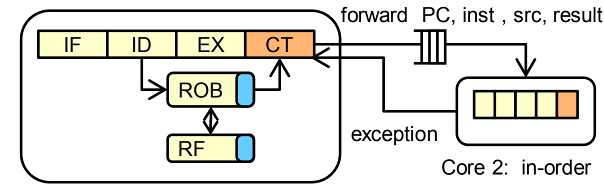
The replication-aware memory system is based on the assumption that accesses from Core 2 always match accesses from Core 1, which is true in the SIMDA model. However, this assumption can be violated if there is an error. To check the consistency between two cores' accesses, the memory system dynamically constructs and checks the address map between the two cores. Conceptually, the address map $MAP[]$ is a table that takes an address of Replica 1 and provides the corresponding address in Replica 2. The mapping is stored as a linear array in its own virtual memory space, and our architecture provides a dedicated L1 cache and a TLB for the address map. Both the map cache and TLB are attached to Core 2.

To construct this map table, the architecture relies on hints from a compiler. On each memory allocation such as a function prolog (stack) or a `malloc()` function (heap), a compiler inserts a special instruction **`mem-alloc`** that indicates an allocation of memory space. The instruction provides the base address $Base$ and the size $Size$ of an allocated memory object. This instruction simply provides hints to set the address map, and does not have any impact on the execution. At run-time, Core 1 forwards the allocation information from Replica 1 to Core 2. Core 2 combines the hints from both replicas and enqueues them into the map store queue, which updates the address map in the background in a way similar to store queues. For a memory allocation, the map store queue updates each map entry that represents the allocated object: $MAP[Base_{R1}+C] = Base_{R2}+C$ where $0 \leq C < Size_{R1}$. Our current implementation requires the size of corresponding objects to be the same for both replicas ($Size_{R1} = Size_{R2}$). As in a typical load bypass for store queues, following reads from the map cache takes the value from the map store queue if there is a matching entry.

During the execution, Core 2 checks the consistency of data memory access from the two replicas by comparing the address from Replica 2 (A_{R2}) with the address from the map ($MAP[A_{R1}]$). If the address does not match the map, an exception is raised indicating an error. For instruction fetches, the mapping between the two PCs is checked on indirect jump instructions when they can diverge. Because such checks are infrequent, the architecture shares one cache for MAP between instruction fetches and data accesses.

In our design, the memory system keeps the mapping for each 64-byte memory block rather than each word. This design choice was made to reduce overheads and also because even the most aggressive layout diversification scheme only needs to change the lay-

[3] The same scalar variable or the same array element.



(a) Asymmetric core architecture

Core 1:
1) Maintain a tag for each instruction and register/ROB value
2) At the commit stage, forward instructions to Core 2 if the tag is one (needs execution on Core 2)

Core 2:
1) Re-execute forwarded instructions (tag of 1)
2) Update/check MAP[A_{R1}] with A_{R2}

(b) Core operation summary

Figure 4. Heterogeneous core architecture for efficient execution of two replicas.

out at the granularity of an array or a heap object, which is typically much larger than 64 bytes. As a result, the memory layout between two replicas can only differ at a 64-byte granularity. All addresses within a 64-byte block in one replica must map to one 64-byte block in another replica, and the six LSBs (least significant bits) of the corresponding addresses in two replicas must be identical. Section 4.2 discusses how the compiler infrastructure can ensure these restrictions in the memory layout diversification.

3.4 Selective Instruction Processing

The processing part of the Orthrus architecture dynamically removes redundant operations from Replica 2 to further reduce unnecessary overheads. As shown in Figure 4, Core 1 executes Replica 1 ahead of Core 2. During its execution, Core 1 maintains a tag for each instruction and data that indicates whether it is different for Replica 2. The tag is read from the memory system on a load and gets propagated on every operation. At the commit stage, Core 1 forwards a completed instruction to Core 2 if either source operands are tagged to be different ($tag == 1$) or the operation needs a consistency check (memory accesses and indirect jumps). Core 2 only executes a small portion of Replica 2, which are identified and forwarded by Core 1. We note that some of Core 2's state such as register values and a program counter (PC) can become stale because some instructions are skipped. To address this issue, Core 2 uses a source operand from Core 1 if it is tagged zero and keeps an offset register between two PCs so that its PC can be obtained from Core 1's PC. To properly update Core 2's PC when the offset between two replicas' PCs may change, a taken branch whose target offset is different from Core 1 and an indirect jump are re-executed on Core 2.

To minimize the impact on Core 1's performance and also enable low-overhead forwarding paths, our design chooses to have two cores loosely coupled instead of executing in lock step. Core 1 can continue committing its instructions without waiting for Core 2. Because the only option on an attack detection is to abort an application, there is no need to support a precise exception. Core 1 stalls its commit stage only if the FIFO between the two cores is full or if there is a system call.

The above optimizations significantly reduce the number of instructions that need to be processed by Core 2. As shown in Figure 2(b), only 30% of instructions on average are different between two replicas. At the same time, because Core 1 resolves control and many data dependencies ahead of time, Core 2 runs faster than by itself. This reduced computation requirement for Core 2 indicates that an asymmetric core configuration, where Replica 1 runs on a fast but large out-of-order superscalar core, and Replica 2 runs on a simple in-order core, can further reduce the silicon area and energy consumption. In fact, the experimental results in Section 5 show that a single-issue in-order core is sufficient for Replica 2 and can match the throughput of a 4-issue superscalar core for all benchmarks that we tried. While the asymmetric cores are beneficial on average cases, we note that a small core cannot guarantee to match the throughput of a large core in extremely memory intensive applications. The Orthrus architecture can support both symmetric and asymmetric configurations.

3.5 Operating System Support

The Orthrus architecture requires the operating system (OS) to be aware of the replicas as well as the hardware features. For example, as discussed in Section 2, the OS must provide the same inputs to all replicas on a system call and remove redundant outputs. The OS must also allocate memory for the address map, properly initialize the meta-data such as value tags and the address map at a start-up, and manage context switches and paging. For this purpose, the processor supports new supervisor instructions to read and write meta-data.

In our architecture, both cores behave as one entity from the OS's perspective. On an exception or a system call, both cores synchronize and trap into one OS instance on Core 1 instead of executing two separate OS instances. In this way, the execution of the replicas can be precisely synchronized. For example, on an interrupt for a context switch, Core 1 waits for Core 2 to complete all instructions and finishes checks before it starts executing the OS code. The OS needs to treat Core 2 as a co-processor of Core 1 and handles both cores together.

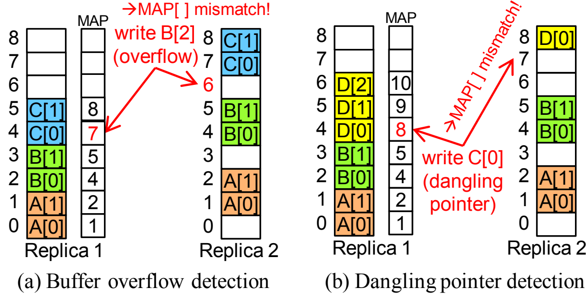
4. Fine-Grained Memory Layout Diversity

This section describes the memory layout diversification scheme that Orthrus uses to protect memory integrity, and shows how the proposed layout diversification together with the address map (MAP) can detect both spatial and temporal memory errors. We first describe the technique based on a fine-grained address map and discuss the changes necessary to accommodate the coarse-grained MAP that is used in the Orthrus architecture.

4.1 Fine-Grained Address Map

A memory layout diversification scheme executes multiple replicas of a program, which are identical except for their memory layouts. Assuming that the program is indifferent to its memory layout, all replicas must access exactly the same variable or array element on each memory access even though they use different addresses. For example, if the first replica reads an array element `a[0]`, the second replica must also read `a[0]` in its memory space. Layout diversification schemes aim to carefully design the differences in the memory layout so that a memory error violates the above invariant and causes a detectable divergence between replicas. In our architecture, the detectable divergence implies that a check on the address map fails.

In order to provide a comprehensive detection of spatial memory errors, we use two simple transformations similar to previous proposals [3, 17]. First, Replica 2's memory space is shifted by a constant to ensure that each object is located at different address



(a) Buffer overflow detection (b) Dangling pointer detection

Figure 5. Memory layout diversification to detect spatial and temporal memory errors.

in two replicas. This transformation ensures that an error that uses data as a pointer, such as a format string error, results in an access to different memory objects in two replicas. As an example, let us consider a simple example in Figure 5(a), which shows three arrays A, B, and C. If a format string attack overwrites memory using 2 as an address, Replica 1 will access $B[0]$ whereas Replica 2 will access $A[1]$. Second, in Replica 2, a pad is added before and after each array and a heap allocated object so that buffer overflows make the replicas access different memory objects. In Figure 5(a), an overflow in B (write to $B[2]$) will overwrite $C[0]$ in Replica 1 whereas overwrite a pad in Replica 2.

As described in Section 3, the proposed architecture uses a dynamically-constructed address map ($MAP[]$) to detect such divergence between memory accesses from two replicas. As an example, Figure 5(a) illustrates how a buffer overflow in Array B gets detected with the address map. In the example, a program erroneously overwrites $B[2]$, which is beyond the array bound. The out-of-bound access results in a write to Address 4 in Replica 1 and a write to Address 6 in Replica 2. Thus, the overflow gets detected by a mismatch between the address map ($MAP[4] = 7$) and the address from Replica 2 (6).

In general, let us consider an overflow in Array A, which either erroneously reads or writes Variable X in Replica 1. Because both replicas use the same array index, Replica 2 accesses $addr_{R2}(A[0]) + C$ when Replica 1 accesses $addr_{R1}(A[0]) + C$. Here, $addr_{R1}(A[0])$ and $addr_{R2}(A[0])$ represent the address of $A[0]$ in Replica 1 and Replica 2, respectively. C is a constant. In order for an out-of-bound access to go undetected, both replicas must access an identical object with that particular memory operation. If the out-of-bound operation accesses Variable X in Replica 1, the following condition must hold: $addr_{R1}(A[0]) + C = addr_{R1}(X)$ and $addr_{R2}(A[0]) + C = addr_{R2}(X)$. In other words, $addr_{R1}(X) - addr_{R1}(A[0])$ must be equal to $addr_{R2}(X) - addr_{R2}(A[0])$. However, because Replica 2 has an extra pad on each array, this condition cannot be true, and an out-of-bound access will get detected.

The proposed layout diversity scheme also detects a memory error where a data value gets used as a pointer. Say that there is a bug such as the format string vulnerability, which causes a data value V to be used as a memory address to access Variable X in Replica 1: $V = addr_{R1}(X)$. Because only the memory layout is different, data values including external inputs are identical for both replicas, and the error implies that both replicas access the same address V. However, our layout transformation ensures that X is located at two different addresses in each memory space, $addr_{R1}(X) \neq addr_{R2}(X)$, which means that the two replicas cannot both access X with an erroneous memory operation. On the other hand, the address map of X will point to the corresponding address in Replica 2 ($MAP[addr_{R1}(X)] = addr_{R2}(X)$). Therefore, the erroneous accesses to X will find a discrepancy in the map. Overall, the proposed diversity together with the address map checks provides a comprehensive protection against spatial memory errors. The experimental results in the next section also demonstrate the error detection capability.

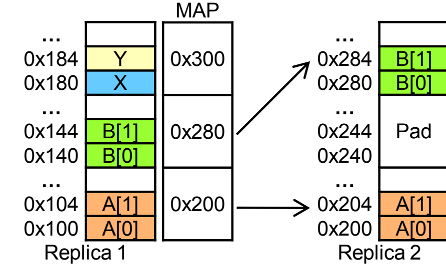


Figure 6. Memory layout diversification for 64-B granularity MAP.

While not as comprehensive, the address map can also detect temporal memory errors in some cases. Dangling pointer errors can be detected if the address map is different before and after the re-allocation. As an example, Figure 5(b) illustrates a dangling pointer error when Array C is freed from Figure 5(a) and Array D is allocated in its place. If a dangling pointer for C ($C[0]$) is used and overwrites $D[0]$ in Replica 1, the updated address map ($MAP[4] = 8$) does not match the old address of $C[0]$ in Replica 2 (7), detecting an error. Similarly, an access to an unallocated memory location can be detected because the address map is not set. A read from an uninitialized location does not incur an inconsistency in the address map, but may be detected eventually if two replicas read different values and diverge in future control flows or memory accesses.

We note that the temporal error detection in Orthrus is not comprehensive. As an example, a use of a dangling pointer may not be detected if the address maps for the old and new objects happen to be identical.

4.2 Coarse-Grained Address Map

As an optimization to reduce overheads, Orthrus maintains the address map at a coarse granularity (64-B blocks) instead of keeping the map for each byte or word. Therefore, the compiler is restricted to change the memory layout between two replicas only at the granularity of the map. If not, two objects that share the same map entry may have different mapping, resulting in a false positive. Our compiler infrastructure ensures that the layout is changed at the 64-B granularity by properly aligning memory objects and injecting pads.

Figure 6 illustrates the layout diversification considering the coarse address map. In addition to adding pads to Replica 2, the compiler places objects that are separated by a pad in two different 64-B blocks. In other words, these objects are aligned at 64-B boundaries. As an example, Array A and B in the figure are placed in two separate 64-B blocks by inserting extra space between them even though the arrays are smaller than 64 bytes. For Replica 2, the compiler also adds an additional pad whose size is a multiple of 64 bytes. These constraints ensure that all objects that share one map entry have identical offsets between the address in Replica 1 and the address in Replica 2. They also ensure that the 6 LSBs (least significant bits) of corresponding addresses in two replica are identical. Therefore, one map entry provides an accurate address translation for all objects within a 64-B block.

We note that not all objects need to be aligned at 64-B boundaries. Our current layout diversity scheme injects a pad before and

after an array, but not between scalar variables because there cannot be an overflow or an underflow from them. Therefore, multiple scalar variables may share the same address map as illustrated by X and Y in the figure.

The additional space for alignment has no practical impact on the detection capability. The layout diversification with a coarse-grained map still detects spatial errors where a data value is used as a pointer or where an out-of-bound access from an array reads or writes another object. The technique can also detect some temporal errors if they result in inconsistent address maps or inconsistent read values. On the other hand, the coarse-grained scheme does not detect an out-of-bound access to the extra space after an array that is added for alignment. For example, in Figure 6, an overflow from Array A to the space between A and B will not be detected. However, such out-of-bound accesses have no impact in practice because the added space is not used by a program.

5. Evaluation

5.1 Experimental Methods

We use detailed simulations to evaluate the effectiveness and performance overheads of the proposed architecture with the layout diversity technique. The simulation infrastructure is built with the Pin binary instrumentation tool [26] and the TAXI performance simulator [42]. In the front-end, the Pin tool runs two versions of a program and generates execution traces. To check the functionality, our back-end uses these traces to construct the address map and check for an error. To evaluate the performance, the back-end performs a cycle-by-cycle micro-architecture simulation that models two processing cores and the memory hierarchy. We used x86 ISA in our simulation. The transformation for the layout diversification is performed with the LLVM compiler infrastructure [24].

The performance simulations use benchmarks from the SPEC CPU2000 suite [18] and the Apache web server. Each benchmark is fast forwarded 2 billion instructions and then a cycle-by-cycle micro-architecture simulation is performed to 200 million instructions. While the SPEC benchmarks do not represent typical network workloads, which are most likely targets of software attacks, they allow us to study the impact of the proposed architecture on a diverse set of workloads. From a processor's perspective, network applications will look similar to bandwidth-limited benchmarks such as `mcf` because they often process a large stream of data. We note that our Apache simulations are currently limited to the user space due to limitations of Pin. We expect Apache to benefit more from Orthrus optimizations in practice because the data transfers in the OS will further stress the off-chip bandwidth. In the experiments, we only used C and C++ programs from the benchmarks because our LLVM extension currently only supports C and C++.

Table 2 summarizes the simulation parameters for four configurations that we compare. The baseline represents a case without any protection. The dual-core represents a traditional method where two large cores without a special memory system run replicas. Finally, there are two configurations for the Orthrus architecture, both with the replication-aware memory system and fine-grained checks; one has symmetric cores and the other has asymmetric cores. We assume a 32-entry FIFO between the two cores. Simulation studies showed that there is not much benefit of having more than 32 entries in the FIFO. The parameters for the complex core are based on Intel's Core2 microarchitecture.

5.2 Error Detection Capability

To validate the error detection capability of Orthrus, we tested a range of real-world attacks and attack test suites. The experiments confirmed that the layout diversification scheme on Orthrus can indeed detect all tested attacks without any false positive. Table 3 summarizes the attacks that we tested with description of the program and the vulnerability. The exploit ID is also presented if applicable. First, we tested 15 buffer overflows in three open source programs (bind, sendmail, and wu-ftpd) using the Zitser test suite [47]. We have also tested additional overflow attacks in various utilities using the BugBench suite [25]. Furthermore, we constructed additional bug test programs for format string attacks, and dangling pointer exploits. These test programs are written based on examples in [35].

Previous studies have also shown that the diversified replication is effective for a large class of real-world attacks. For example, N-variant systems studied the Apache web server to demonstrate that the address space partitioning is effective against pointer injections [10] and the data diversity can detect critical data (UID) corruption [31]. DieHard [3] also demonstrated that diversified heap layouts can detect errors in the Squid web cache and the Mozilla web browser.

5.3 False Positives

To test possible false positives, we functionally simulated the SPEC and Apache benchmarks used in this paper for 2 billion instructions each. We did not encounter any false positive cases.

5.4 Die Area and Power Consumption

The main on-chip hardware overheads for the Orthrus architecture consists of the MAP cache and additional tag bits for each level of the caches. We note that the MAP TLB and FIFO queue structures add an insignificant amount of hardware in comparison to the MAP cache and the tag bits. To evaluate the additional hardware overhead of Orthrus, we estimated the area and power consumption of the MAP cache as well as other caches with and without additional tag bits using Cacti [38] for the 45nm technology node. For the 32-KB 2-way MAP cache and 1-bit tag per 32-bit word, which are used in our simulations, the area overhead is estimated to be $0.24mm^2$ for the MAP cache and $0.08mm^2$ for tag bits. The MAP cache and the tag bits also add 0.23 Watts of total power. Compared to a 45nm Intel Core 2 Duo processor [20], these overheads are less than 0.5% of the total area and less than 1% of the total power dissipation.

The asymmetric Orthrus architecture uses a simple in-order core instead of another out-of-order core to execute the second replica. While we do not have numbers from a detailed implementation, previous studies on checker cores [1, 43] suggest that a simple core will require about 10 times less area and power compared to the baseline core. For example, Intel's 45nm Core 2 Duo processor has an area of $107mm^2$ [20] with the power budget of 35 Watts whereas a much simpler and in-order Intel Atom processor running at the same frequency has an area of $25mm^2$ and consumes only 2.5 Watts. Therefore, the Atom processor, if used as the checker core, will occupy less than a quarter of the area and consume less than 8% of the power compared to having another complex core. Overall, the use of asymmetric cores can significantly reduce the overheads for the additional core in Orthrus.

5.5 Performance Overheads

Figure 7 illustrates the performance of Orthrus for SPEC2000 benchmarks and Apache when two replicas with different memory layouts execute simultaneously. The performance is shown as execution time that is normalized to the baseline. On average, because most benchmarks only utilize a small portion of the off-chip bandwidth, both traditional dual-core configuration and the Orthrus architecture closely match the performance of the baseline: 12.0% slowdown for the dual-core, 2.2% slowdown for Orthrus with symmetric cores, and 2.7% slowdown for Orthrus with asymmetric cores. However, for a bandwidth-sensitive benchmark such as `mcf`,

Parameter	Baseline	Dual-Core	Orthrus Symmetric	Orthrus Asymmetric
Frequency	3 GHz			
Processing Cores	1 4-issue OoO	2 4-issue OoO	2 4-issue OoO	1 4-issue OoO & 1 1-issue in-order
Checks	None	Coarse-grained	Fine-grained	Fine-grained
L1 caches (I$/D$) (32B Block)	4-way 64KB/64KB	Core1: 4-way 64KB/64KB Core2: 4-way 64KB/64KB	Core1: 4-way 64KB/64KB Core2: 4-way 64KB/64KB MAP: 2-way 32KB	Core1: 4-way 64KB/64KB Core2: 2-way 32KB/32KB MAP: 2-way 32KB
L1 latency	2 cycles			
L2 cache (64B Block)	2-MB unified, 8-way, 14 cycle latency			
DRAM	Latency: 50 ns, Bandwidth: 8 GB/s			

Table 2. Simulation parameters.

Program	Ref	Description	Error
bind-8.2	CA-1999-14	DNS server	Size arg of `memcpy` not checked
bind-8.2	CA-1999-14	DNS server	`memcpy` underflows to large positive int
bind-8.2	CVE-1999-0009	DNS server	Size arg of `memcpy` not checked
bind-8.2	CVE-2001-0013	DNS server	Use of sprintf() without proper bounds checking
sendmail-8.12	CA-2003-07	mail transfer agent	`sprintf`() without bounds checking
sendmail-8.12	CVE-1999-0131	mail transfer agent	Upper bound increment for a > char not decrement for <
sendmail-8.12	CVE-1999-0206	mail transfer agent	gecos field copied into fixed-size buffer without size check
sendmail-8.12	CVE-1999-0047	mail transfer agent	Pointer to buffer not reset after line read
sendmail-8.12	CA-2003-12	mail transfer agent	Size check not performed
sendmail-8.12	CVE-2001-0653	mail transfer agent	Input byte set to `0xff` cast to minus one error code
sendmail-8.12	CVE-2002-0906	mail transfer agent	Unchecked `strncpy`
wu-ftpd-2.6.2	CVE-1999-0878	FTP server	Unchecked `strcpy`
wu-ftpd-2.6.2	CAN-2003-0466	FTP server	Wrong size check inside a if statement
wu-ftpd-2.6.2	CVE-1999-0368	FTP server	Unchecked `strcpy` and `strcat`
ncompress-4.2.4	CVE-2001-1413	file (de)compression	Stack smash
polymorph-0.4.0	BID-7663	file system "unixer" (WIN32 to Unix filename converter)	Stack smash & Global buffer overflow
gzip-1.2.4	CVE-2001-1228	file (de)compression	Global buffer overflow
man-1.5h1	CVE-2001-0641	documentation tools	Global buffer overflow
bc-1.06	AccMon [46]	interactive algebraic language	Heap buffer overflow
bug1	Seacord 2005 [35]	Bug test program	Dangling pointer
bug2	Seacord 2005 [35]	Bug test program	Format string

Table 3. Attacks used for error detection validation.

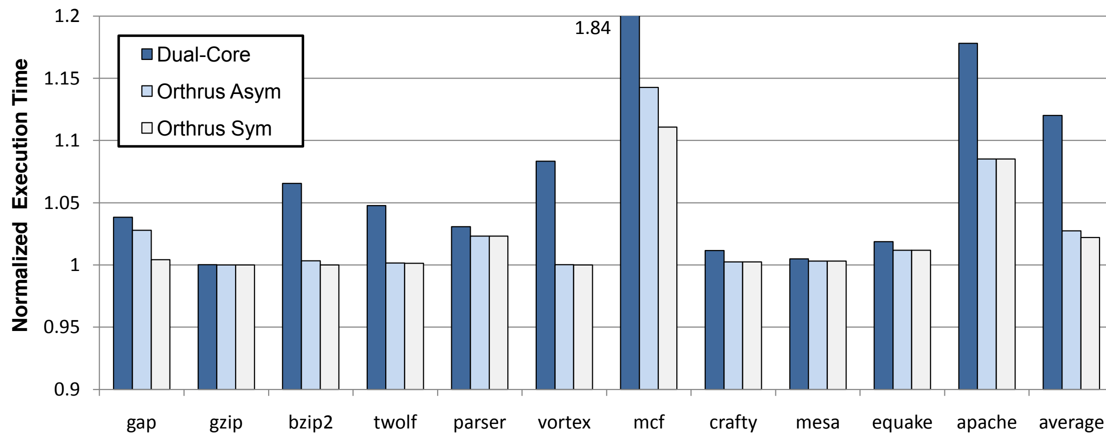


Figure 7. Slowdown for layout diversity.

the traditional dual-core incurs a significant slowdown (84%) by increasing the off-chip traffic. In this case, the replication-aware asymmetric architecture reduces the overhead to 14.3% by effectively reducing the memory traffic. For Apache, Orthrus reduces the performance overhead from 17.8% to 8.5%. We note that this performance improvement is even without considering data transfers inside the operating system where most web page data are moved. Pin only allows us to instrument user-level instructions, therefore we could not monitor OS activity. We expect the performance improvement to be greater if we include OS-level memory

accesses. The data transfers for large web pages will stress the off-chip bandwidth but be common for both replicas. The Orthrus memory system will combine off-chip accesses from two replicas for such common data transfers. The results also show that the asymmetric configuration has a negligible performance degradation compared to the symmetric configuration at least for the benchmarks here.

In order to understand the sources of the performance overhead in Orthrus, we performed a detailed analysis of the overhead for the asymmetric core configuration. The breakdown is estimated by incrementally adding each source of the overhead in simulations: Core1-to-Core2 FIFO stalls, `mem-alloc` instructions, $MAP[]$ accesses, shadow memory ($MEM2$) accesses. As shown in Figure 8, the performance overheads in `mcf` and Apache are largely due to shadow memory accesses and partly due to address map accesses. The shadow memory accesses and address map accesses pollute the shared L2 cache and the off-chip bandwidth, and may slow down the main core (Core 1) by increasing its L2 misses and memory latency. Given that `mcf` and Apache are the two benchmarks with the highest performance overheads for Orthrus, the shadow memory access appears as the main source of the performance slowdowns.

Some benchmarks such as `gzip`, `parser`, `vortex`, and `equake` show slowdowns mainly due to additional `mem-alloc` instructions. For example, we found that `parser` has significantly more `mem-alloc` instructions than other benchmarks. When the map store queue is full due to a large number of memory allocations, Core 2 needs to stall on the next `mem-alloc` instruction. Such overheads may be alleviated by increasing the size of the map store queue. For `equake`, we also found that the injection of the `mem-alloc` instructions slightly degrades the branch prediction accuracy and the instruction cache hit-rate. However, we note that the absolute performance overheads due to the memory allocation instructions are insignificant. As an example, `gzip` and `vortex` do not show any noticeable slowdown (less than 0.02%) as shown in Figure 7 even if the `mem-alloc` instruction is the main source of the slowdown.

If the FIFO queue between the two cores gets full, it slows down the main core by delaying instruction commits. Such stalls due to the throughput discrepancy in the asymmetric configuration contribute significantly to several benchmarks' slowdown. However, again, the absolute slowdown due to the FIFO stalls was rather insignificant, even in the worst case in `gap`. The results indicate that the small core can keep up with the large core in general, and the 32-entry FIFO queue is large enough to hide any temporary throughput differences between the two cores. In fact, our simulation studies (not shown here) indicate that there is not much benefit of having more than 32 entries in the FIFO.

In order to see the effects of architectural parameters, we also simulated the performance with varying L1 and L2 cache sizes and parameters (not shown here). We found that the average performance overheads for various L2 sizes remain almost the same at 2.7% because the baseline performance also changes with different L2 size. L1 cache parameters have small effects because Core 2's execution does not directly stall the main core.

5.6 Off-Chip Bandwidth Overheads

Table 4 summarizes the off-chip bandwidth usage of the traditional dual-core configuration and our architecture with two replicas. The table only shows one set of numbers for Orthrus because both symmetric and asymmetric configurations use the same replication-aware memory system and show almost identical memory bandwidth usage. As shown in the table, the dual-core configuration more than doubles the bandwidth consumption as both cores make the same number of memory accesses to two disjoint memory spaces, which increases both the total number of memory accesses as well as L2 cache miss-rates. On the other hand, Core 2 and the address map (MAP) together in the Orthrus architecture consume only 22.3% more bandwidth compared to the Core 1's bandwidth consumption. While the savings in the bandwidth may not translate into increased performance for applications with low bandwidth usage, the lower off-chip traffic should result in a lower energy consumption. Because the off-chip bandwidth cannot scale with the number of transistors on a die, saving the bandwidth is likely to become even more important for future many-core processors.

5.7 Memory Space Overheads

Figure 9 shows the memory usage of Orthrus's layout diversity scheme normalized to the baseline without any protection. The traditional dual-core configuration incurs a 2 times overhead in memory space in order to run two replicas. On the other hand, the replication-aware memory system allocates a page for Replica 2 only if its content is different from Replica 1's. As a result, in our memory system, Replica 2 consumes much less memory (17.9%) than Replica 1. Our current implementation cannot take a full advantage of the fact that only about 7% of memory values are different between replicas (see Figure 2(a)), because shadow memory allocation is done at a page granularity. This is also the reason why gap shows close to 2x memory usage overhead. The memory access pattern of gap is spread across different pages in the memory space.

In addition to the memory space for applications, our memory system also uses meta-data such as a 1-bit tag for each 32-bit word and an address map (32 bits) for each 64-byte block, which result in an additional 9.375% overhead. Overall, on average, our memory system requires 27.3% more memory than the baseline, which is much less than the dual-core configuration.

6. Related Work

There exists a large body of work on protecting computing systems from various types of software exploits. Here, we discuss most closely related work that either provides a broad attack coverage or has similarities in the approach. Due to the space limit, we could not include all software protection techniques.

Software Diversity and Redundancy. Section 2 describes the previous work on software diversification and replication techniques for security [2–4, 8, 10, 16, 23, 31, 34, 44]. Overall, the proposed Orthrus architecture implements such replication techniques efficiently and enables fine-grained checks for enhanced security.

Diversification and replication techniques are also being used beyond protecting the software integrity. TightLip [45] protects the confidentiality by using a replica with diversified sensitive data. The outputs of the two processes are then checked for divergence at system call level to determine whether sensitive information could be leaked. Bressoud and Schneider proposed a software approach for fault-tolerance using hypervisor based replicas [6]. The Orthrus architecture can potentially be extended to support such replication techniques more efficiently.

Safe Languages. This paper studies the layout diversification, which provides a comprehensive detection of spatial memory errors as well as some temporal errors, as the main example of our proposed architecture. Alternatively, safe languages such as Java, and safe dialects of C such as CCured [30] and Cyclone [21] can provide strong memory safety using type systems and run-time checks. While safe languages provide more comprehensive memory safety then the layout diversification, they also require non-trivial programmer efforts to port legacy code and incur significant overheads. On the other hand, the layout transformation is completely automatic.

Bound Checks. Recent studies show that compilers can automatically add array bound checks to legacy C code [15, 29, 33].

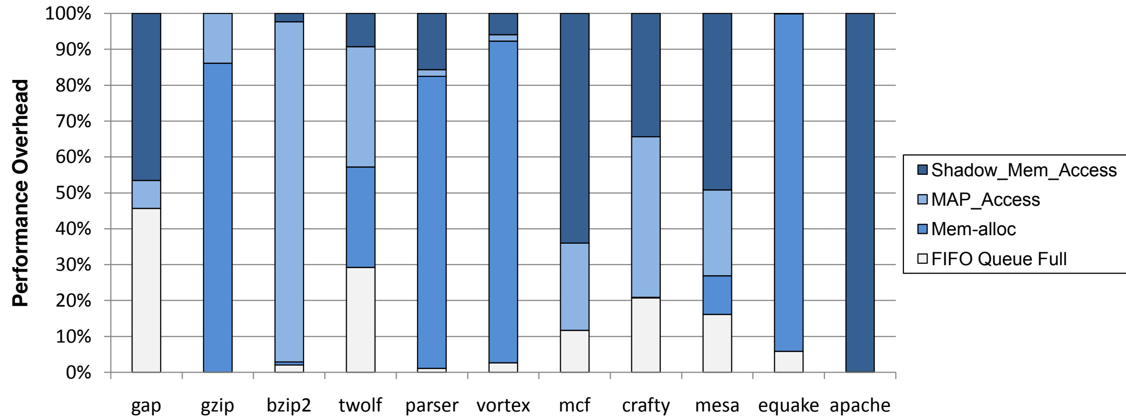


Figure 8. The breakdown of the performance overheads for the asymmetric Orthrus configuration. The breakdown is estimated by incrementally adding each source of the overhead in simulations: Core1-to-Core2 FIFO stalls, `mem-alloc` instructions, $MAP[]$ accesses, shadow memory ($MEM2$) accesses.

Benchmark	Baseline	Dual-Core		Orthrus		
	Base Core	Core 1	Core 2	Core 1	Core 2	MAP
gap	18.4454	19.4485	19.4415	19.4091	9.2265	0.9744
gzip	0.3098	0.3098	0.3098	0.3097	0.0005	0.0035
bzip2	12.4595	29.3977	29.1489	13.6530	0.0073	0.4352
twolf	1.0427	13.9297	13.5913	1.0427	0.0018	0.0327
parser	1.4385	1.7522	1.9459	1.5825	0.3493	0.0367
vortex	0.4558	0.5309	0.5382	0.4710	0.0927	0.0125
mcf	807.2247	1193.9716	1184.2169	883.0118	302.9616	47.8127
crafty	1.1673	1.7559	1.7613	1.1682	0.0053	0.0049
mesa	0.1773	0.1772	0.1772	0.1773	0.0212	0.0022
equake	5.4999	7.3787	7.3673	5.4993	0.0040	0.1793
apache	0.2344	0.2344	0.2344	0.2344	0.1807	0.0094

Table 4. Off-chip bandwidth usage (MB per 200 million instructions).

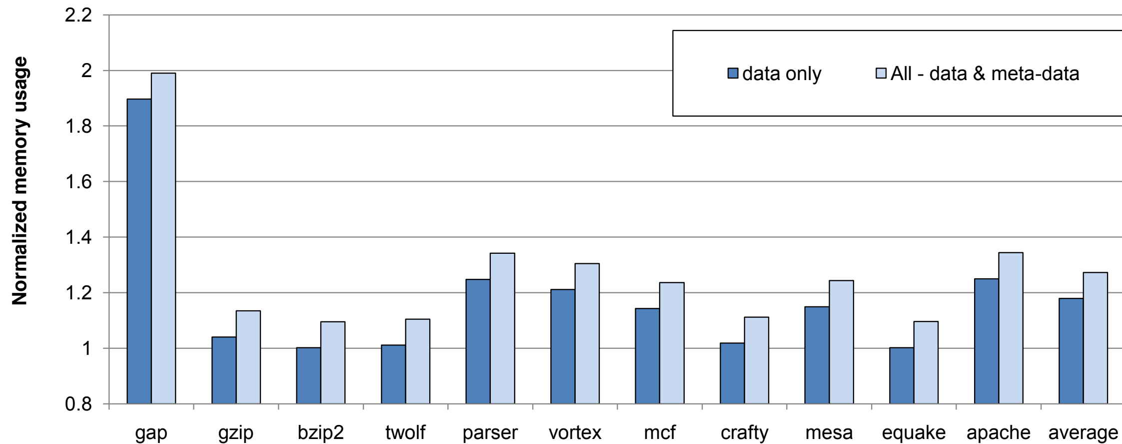


Figure 9. Memory usage for layout diversity.

However, bound checks in software can still incur a significant slowdown in memory intensive programs (up to 1.69x in [15]). Hardware support for bound checks [9, 13] can reduce the performance overhead to 5-20% and provide comprehensive protection for buffer overflows. Compared to Orthrus, these hardware bound checking schemes show comparable but slightly higher performance overheads. The additional hardware requirements are also comparable to Orthrus if we do not consider the second core that is used by Orthrus. As an example, HardBound [13] requires a tag bit per word to mark pointers as well as base and bound addresses

for each pointer, which has similar overheads as the MAP and the shadow memory in Orthrus. However, the bound checks only target to address one type of spatial memory errors, namely buffer overflows. The proposed architecture with the diversified replication is capable of detecting corruption of pointers and data from other vulnerabilities such as format string errors, temporal memory errors, etc.

Hardware Support for Dynamic Taint Analysis. For security attacks, dynamic taint analysis provides a general technique to detect a large class of software attacks including high-level exploits such as SQL injections with minimal hardware and performance overheads [11, 12, 40]. However, the taint analysis can only detect a subset of memory errors where I/O inputs corrupt code pointers or certain critical data that are checked. The proposed architecture provides a comprehensive memory protection from both malicious attacks and unintended software errors.

Multi-Core for Security. INDRA [37] and LBA [7] propose to use multi-core architecture to inspect run-time program behavior. Nagarajan et al. also studied dynamic taint analysis on multi-cores [28]. While the idea of using multi-cores is similar to the proposed architecture, these techniques are designed to explicitly check properties such as illegal control flows or use of tainted values, and rely on symmetric multi-cores. The proposed architecture supports software diversity and redundancy, and investigates both symmetric and asymmetric cores.

Efficient Execution of Software Variants. Tucek et al. proposed delta execution as an efficient way to validate software patches by executing both patched and unpatched versions together [41]. To reduce overheads, delta execution only executes parts of the patched program that are different from the original and have both versions share common pages. Biswas et al. [5] proposed to improve the performance by exploiting the similarities in data when many instances of a program with slightly different data need to be run. In a high-level, these approaches are similar to the proposed architecture optimizations. However, they target different applications and do not support diversity in memory layouts. The Orthrus architecture can potentially be used to extend the delta execution for more complex patches that change memory layouts.

7. Conclusion and Future Directions

This paper presents the Orthrus architecture that enables efficient software diversity and redundancy with fine-grained checks. The proposed architecture significantly reduces the overheads associated with executing multiple replicas by removing unnecessary redundancy. The experiments demonstrate that a small single-issue in-order core is sufficient to match the throughput of a wide out-of-order core. Utilizing this hardware support, the paper also introduces and evaluates a memory layout diversification scheme that can detect a wide range of memory errors.

The current Orthrus design does not support multithreaded applications. However, we believe that the proposed approach to efficiently execute multiple replicas can be generalized to multithreaded applications. One challenge in replicating multithreaded applications lies in ensuring identical execution of replicas. Recent techniques from deterministic multithreading and replay present viable solutions to this problem [14, 19, 27]. Moreover, we believe that the memory compression in Orthrus reduces the sources of non-determinism in multithreading because synchronization operations are typically done using data, which are identical between replicas. For multithreaded applications, Orthrus must also ensure the coherence between data and their tags. Fortunately, recent studies in Dynamic Information Flow Tracking handle the same issue [22, 28].

In the future, we also plan to extend Orthrus to protect confidentiality in addition to software integrity, to automatically recover from attack techniques exploiting multiple replicas, and to handle hardware and design errors in addition to the software security attacks. We believe that such extensions can be implemented with minimal increase in the overheads because Orthrus already has the redundancy that is often required to detect other types of errors.

Acknowledgments

We thank anonymous reviewers for their feedback and Professor David Lie for his suggestions to improve the final version of the paper. This work was partially supported by the National Science Foundation under grants CNS-0746913 and SA4897-10808PG, by the Air Force Office of Scientific Research under Grant FA9550-09-1-0131, and an equipment donation from Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF, AFOSR, or Intel.

References

[1] T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the* 32th *International Symposium on Microarchitecture*, November 1999.

[2] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the* 10th *ACM Conference on Computer and Communications Security (CCS03)*, 2003.

[3] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, 2006.

[4] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploitsth. In *Proceedings of* 12th *USENIX Security Symposium*, 2003.

[5] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong. Multi-execution: multicore caching for data-similar executions. In *Proceeding of the* 36th *International Symposium on Computer Architecture*, June 2009.

[6] T. Bressoud and F. Schneider. Hypervisor-based fault tolerance. In 15th *ACM Symposium on Operating Systems Principles*, 1995.

[7] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. Gibbons, T. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the* 35th *International Symposium on Computer Architecture*, June 2008.

[8] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. In *Technical Report CMU-CS-02-197*, 2002.

[9] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic. Effective memory protection using dynamic tainting. In *Proceedings of the* 22nd *International Conference on Automated Software Engineering*, 2007.

[10] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the* 15th *USENUX Security Symposium*, August 2006.

[11] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the* 37th *International Conference on Microarchitecture*, December 2004.

[12] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the* 34th *International Symposium on Computer Architecture*, June 2007.

[13] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. In *ASPLOS XIII: Proceedings of the* 13th *international conference on Architectural support for programming languages and operating systems*, pages 103–114, 2008.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *ASPLOS XIV: Proceeding of the*

14th international conference on Architectural support for programming languages and operating systems, 2009.

[15] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceeding of the* 28th *International Conference on Software Engineering*, May 2006.

[16] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proceedings of* 6th *Workshop on Hot Topics in Operating Systems*, 1997.

[17] M. Franz. Understanding and countering insider threats in software development. In *Proceedings of the 2008 International Conference on e-Technologies*, January 2008.

[18] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.

[19] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the* 35th *International Symposium on Computer Architecture*, June 2008.

[20] Intel Inc. Intel Details Upcoming New Processor Generations, 2007. http://www.intel.com/pressroom/archive/releases/20070328fact.html.

[21] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, 2002.

[22] H. Kannan. Ordering decoupled metadata accesses in multiprocessors. In *ACM/IEEE* 42nd *International Symposium on Microarchitecture (MICRO-42)*, December 2009.

[23] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the* 10th *ACM Conference on Computer and Communications Security (CCS03)*, 2003.

[24] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, March 2004.

[25] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools(PLDI'05)*, 2005.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the* 2005 *Conference on Programming Language Design and Implementation International (PLDI)*, June 2005.

[27] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution effciently. In *Proceedings of the* 35th *International Symposium on Computer Architecture*, June 2008.

[28] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta. Dynamic information flow tracking on multicores. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architectures*, 2008.

[29] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 245–258, 2009.

[30] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the* 29th *ACM Symposium on Principles of Programming Languages*, 2002.

[31] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson. Security through redundant data diversity. In *Proceedings of the* 38th *IEEE/IFPF International Conference on Dependable Systems and Networks, Dependable Computing and Communications Symposium*, 2008.

[32] R. Pucella and R. B. Schneider. Independence from obfuscation: A semantic framework for diversity. In *Proceedings of the 2006 Computer Security Foundations Workshop*, 2006.

[33] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the* 11th *Annual Network and Distributed System Security Symposium*, 2004.

[34] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, 2009.

[35] R. C. Seacord. *Secure Coding in C and C++ (SEI Series in Software Engineering)*. Addison-Wesley Professional, 2005. ISBN 0321335724.

[36] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the* $11th$ *ACM Conference on Computer and Communications Security*, 2004.

[37] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. INDRA: An integrated framework for dependable and revivable architectures using multicore processors. In *Proceedings of the* 33rd *International Symposium on Computer Architecture*, 2006.

[38] P. Shivakumar and N. J. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical report, WRL Research Report, Feb. 2001.

[39] A. N. Sovarel, D. Evans, and N. Paul. Wheres the FEEB? the effectiveness of instruction set randomizationth. In *Proceedings of the* 14th *USENIX Security Symposium*, 2005.

[40] G. E. Suh, J. Lee, D. X. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the* 11th *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[41] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *Proceedings of the* 14th *International Conference on Architecture Support for Programming Languages and Operating Systems*, 2009.

[42] S. Vlaovic. TAXI: Trace analysis for x86 interpretation. In *Proceedings of the 2002 IEEE International Conference on Computer Design*, pages 508–514, 2002.

[43] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *IEEE International Conference on Dependable Systems and Networks (DSN-2001)*, June 2001.

[44] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of* 22nd *International Symposium on Reliable Distributed Systems (SRDS03)*, 2003.

[45] A. R. Yumerefendi, B. Mickle, and L. P. Cox. Tightlip: Keeping applications from spilling the beans. In *NSDI*, 2007.

[46] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *37th International Symposium on Microarchitecture (MICRO)*, pages 269–280, 2004.

[47] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT Softw. Eng. Notes*, 29(6):97–106, 2004.