The Rise of Memory-Safe Languages: Building a Fast, Elastic, Secure Software & Hardware Architecture

Anjo Vahldiek-Oberwagner (Intel Labs)

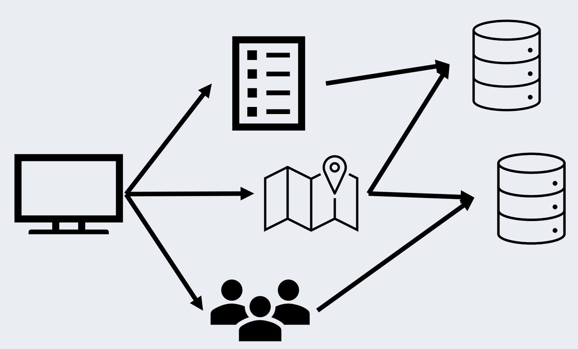
2022 DARPA RISER forward.darpa.mil

Cloud Services waste 25% of Compute

- Monolithic architectures (Google, Facebook, etc.) incur high overheads
  - CPU wastes 25% of cycles on preparing communication
  - High tail latencies due to dependencies
- Moving to logically decomposed services improves:
  - Developer productivity by 10x
  - Isolation of security vulnerabilities
  - Scaling service elastically with load

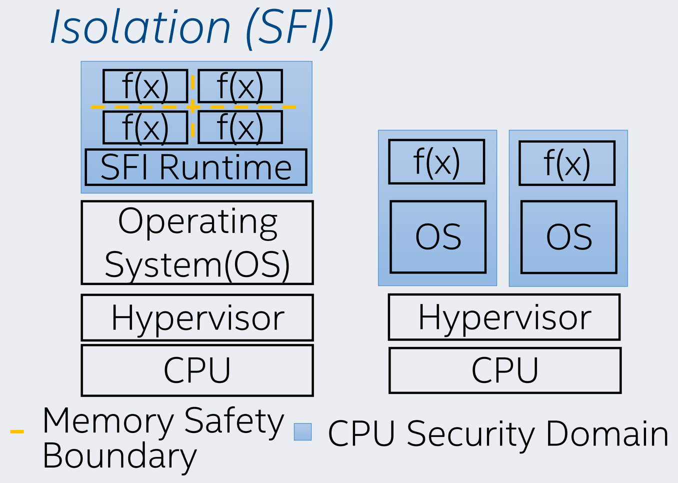
Distributed Cloud Service:

Collections of communicating Services



Existing architectures cannot simultaneously be fast, elastic and secure

	SFI	MicroVM	multiPIE
Execution	20-40% slower	Native	Native
Creation	5 ms	20x slower	<10ms
Domains	10,000	100x fewer	100,000s
Switch time	2 ns	> 1,000 ns	<10ns
Security	HW Attacks	Strong	Strong
Sharing	5-10% wasted	25% wasted	instant



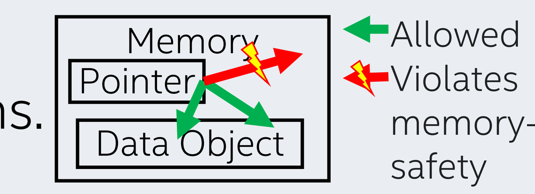
Research improving SFI:
- Swivel (USESec'21) → Harden SFI against HW, 5-240% slower
- Hardware Fault Isolation → Special hardware extension
- Wasm compiler optimizations → 10% perf. improvement

Research improving MicroVM:
- µSwitch (IEEE S&P'23) → Faster + avoid unnecessary switches
- LittleMac → Secure, Programmable Isolation + Sharing

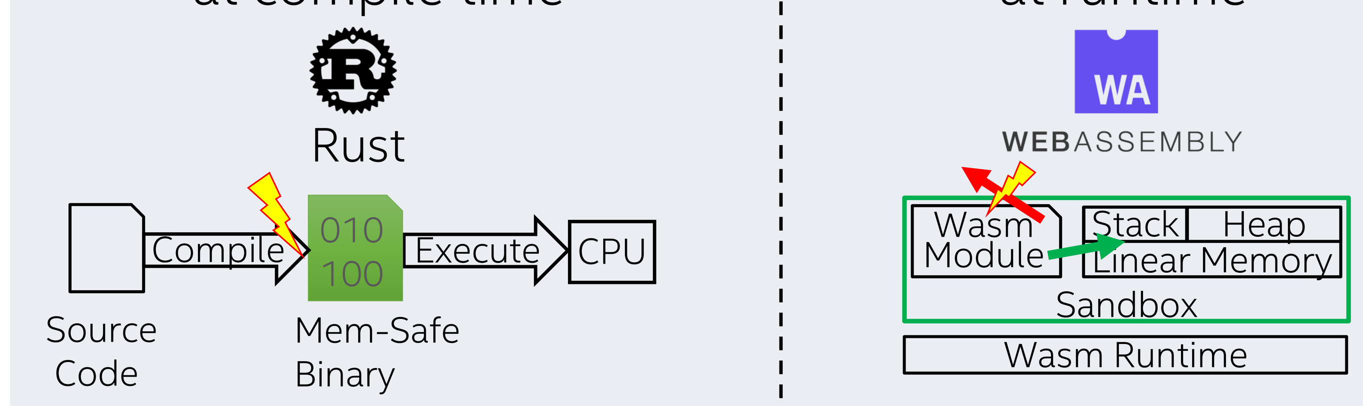
UC San Diego · Purdue University · Rice

The Rise of Memory-Safe Languages

70% of security vulnerabilities are caused by memory-safety violations.



Memory-Safe Languages prevent violations



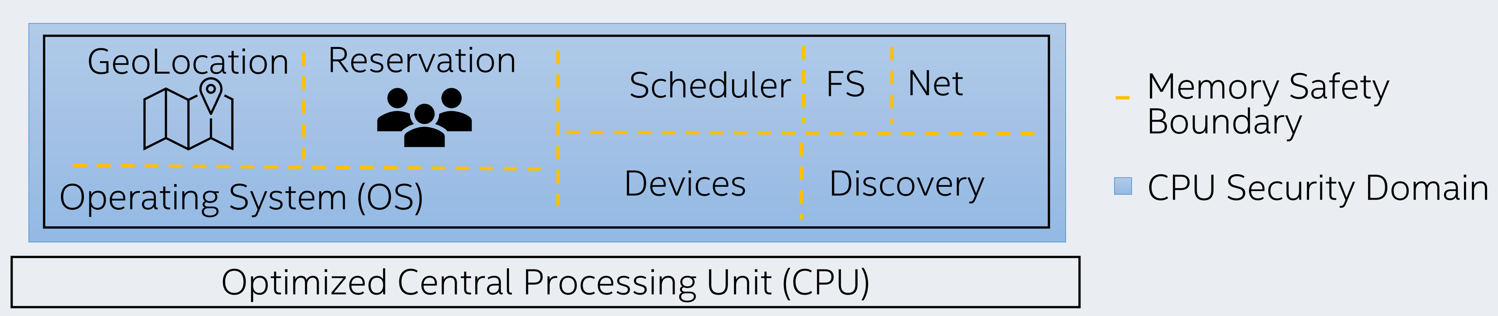
At compile time:
- Ownership
- Comparable performance

At runtime:
- Virtual Machine
- 20-40% performance overhead

Vision: Fast, Elastic, Secure Memory-Safe Software & Hardware Architecture

Key Insight:

Collapse services and protect with memory-safe languages



- Memory-Safe Languages restrict
  - Access to service's memory
  - Execution to predefined entry points in services
  - ➢ Eliminates the use of CPU security domains
    - ➢ 20x faster creation
    - ➢ 100x more domains
- Sharing between OS and service is a function call
  - No marshalling of complex structures
  - No copy to/from service, all memory is shared
  - No synchronization via thread migration
  - Kernel bypass for remote communication
  - ➢ Regain 25% of CPU cycles wasted for communication
  - ➢ Improve tail-latency due to complex synchronization chains

Challenges and Opportunities: Safe & Efficient SW/HW Abstraction with Legacy Support

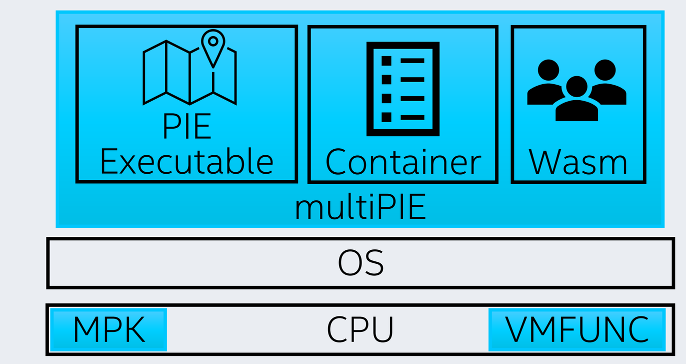
Challenge of Memory-Safe Language Security:
- Dependence on trusting the compiler and runtime toolchain
- Research directions:
  - Verified compilers and runtimes
  - Trusted Hardware support for memory-safe languages
  - Trusted supply chain with runtime validation

Challenge of Legacy Service Support:
- Legacy services are not written in memory-safe languages
- Without legacy support, slow adoption
- Research and Industry directions:
  - Wasm as compilation target for many languages
  - Memory-safe implementations of important interpreters
  - Light-weight hardware technique restricting access to memory regions

Optimization of a Memory-Safe CPU:
- Memory layout of memory-safe languages is simpler than usual applications
  - Statically bind virtual to physical mapping to reduce page miss handler
  - Memory address translation in 1 cycle instead of 4, no page miss handler

Challenge and Optimization of Secure Sharing between Services:
- Revocation of sharing not possible in single address space with today's CPU
- Capability-based Hardware (e.g., CHERI or Cryptographic Computing)
  - Offer memory permission at sub-page granularity
  - Sharing via forwarding of capabilities instead of pointers

multiPIE Approach: Fast, Efficient and Secure Software Runtime



Single-Process Software Model for all Services

- multiPIE capabilities:
  - Supports legacy executables, containers and Wasm modules
  - Loads existing service packages
  - Multiplexes system resources (e.g., files)
  - Is written in Rust to statically and automatically validate implementation against safety guarantees
  - Optimizes interactions between services and OS
- multiPIE is an intermediate layer:
  - Validate sharing abstractions between services
  - Demonstrate best case performance
  - Explore limitations of today's CPUs
  - Evaluate proposed CPU techniques
  - Offer new software model to industry