

# Runtime Integrity Checking for Exploit Mitigation on Embedded Devices

Matthias Neugschwandtner

*IBM Research, Zurich*

*eug@zurich.ibm.com*



Collin Mulliner

*Northeastern University, Boston*

*collin@mulliner.org*



**9<sup>th</sup> International Conference on Trust & Trustworthy Computing**

Vienna, August 2016

# Embedded Devices?

---



# Internet of Things!

---



# Embedded Devices

---

- Produced in large quantities
  - not a computer, but actually a computer
- Mostly low cost/power/end RISC-based CPUs
  - exceptions, e.g. CPUs for smartphones
- Devices run open/free software such as Linux
  - light software stacks, for example uClibc

# Embedded Device Security

---

- Valuable targets
  - always on
  - contain interesting personal data
  - control important things
- Contain software vulnerabilities
  - e.g. memory corruption
  - exploited like desktops and servers

# Embedded Device Security

---

- Valuable targets
  - always on
  - contain interesting personal data
  - control important things
- Contain software vulnerabilities
  - e.g. memory corruption
  - exploited like desktops and servers
- **Mitigations not state of the art!**

# Exploit Mitigation - State of the Art

---

Exploit stages:

- Inject payload
- Hijack control flow
- Run payload

# Exploit Mitigation - State of the Art

---

- Inject payload
  - Data Execution Prevention
  - Address Space Layout Randomization
- Hijack control flow
  - Control Flow Integrity
- Run payload
  - Policies for system call usage
  - System call based IDS



# Exploit Mitigation - State of the Art

---

- Inject payload

- Data Execution Prevention
  - MMU hardware support required (SW emulation slow)
- Address Space Layout Randomization
  - limited address space on embedded devices



- Hijack control flow

- Control Flow Integrity
  - source code beneficial
  - high overhead
  - hardware support only for next-gen Intel processors



- Run payload

- Policies for system call usage
  - requires writing policies for every application
- System call based IDS
  - mimicry attacks, overhead



# Goal: SotA Mitigations for embedded RISC devices

---

- Lightweight exploit mitigation
  - also suitable for “budget” SoCs
- Use RISC hardware features
- Tailor for “binary only” / COTS
  - source code is not always available

# RISC Architecture Features

---

- Register only operations
  - load / store architecture
- Many registers and specialized registers
  - e.g. control flow
- Fixed instruction length
  - easier disassembly
- Instruction / address alignment
  - no jumping into the middle of an instruction



# Exploits revisited

---

- Exploits use OS functionality
  - read/write data, launch process, ...
- Exploit OS usage differs from original program
  - different syscall, different parameters, ...

# Exploits revisited

---

- Exploits use OS functionality
  - read/write data, launch process, ...
- Exploit OS usage differs from original program
  - different syscall, different parameters, ...
- **Ensure that runtime OS usage is coherent with OS usage in binary executable**

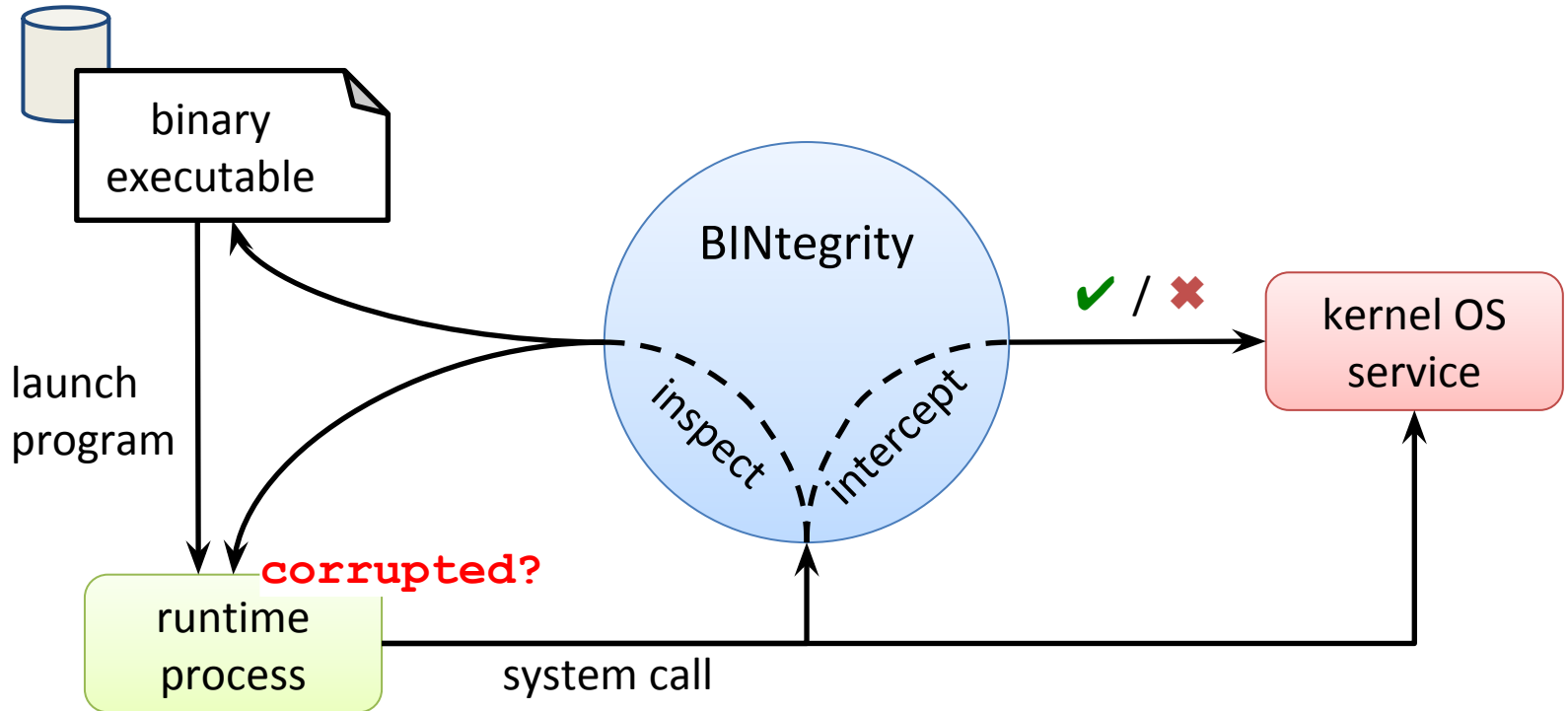
⇒ **BINtegrity**

# Threat Model

---

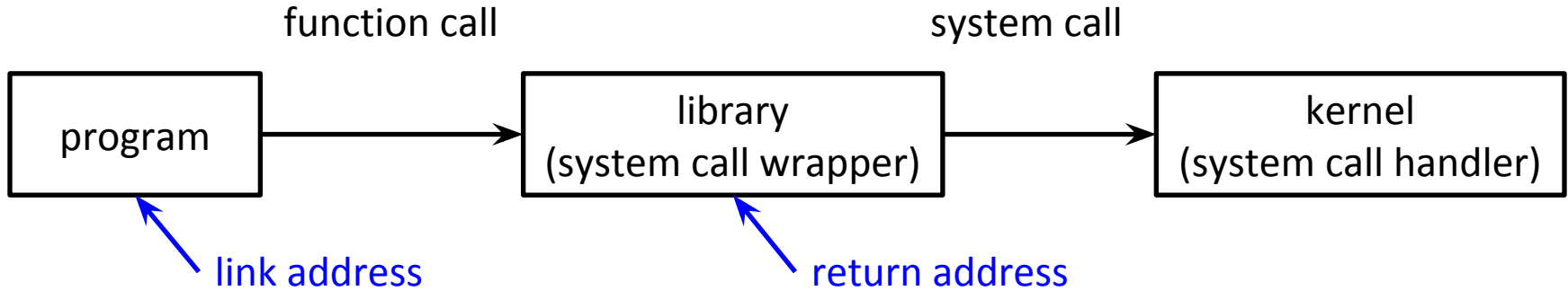
- Trusted kernel ✓
  - we protect user space code
- Trusted binaries on disk ✓
  - executable and libraries not modified by attacker
- Memory is untrusted ✗
  - we try to fight off memory corruption attacks!

# BINtegrity Overview



# Process Runtime State

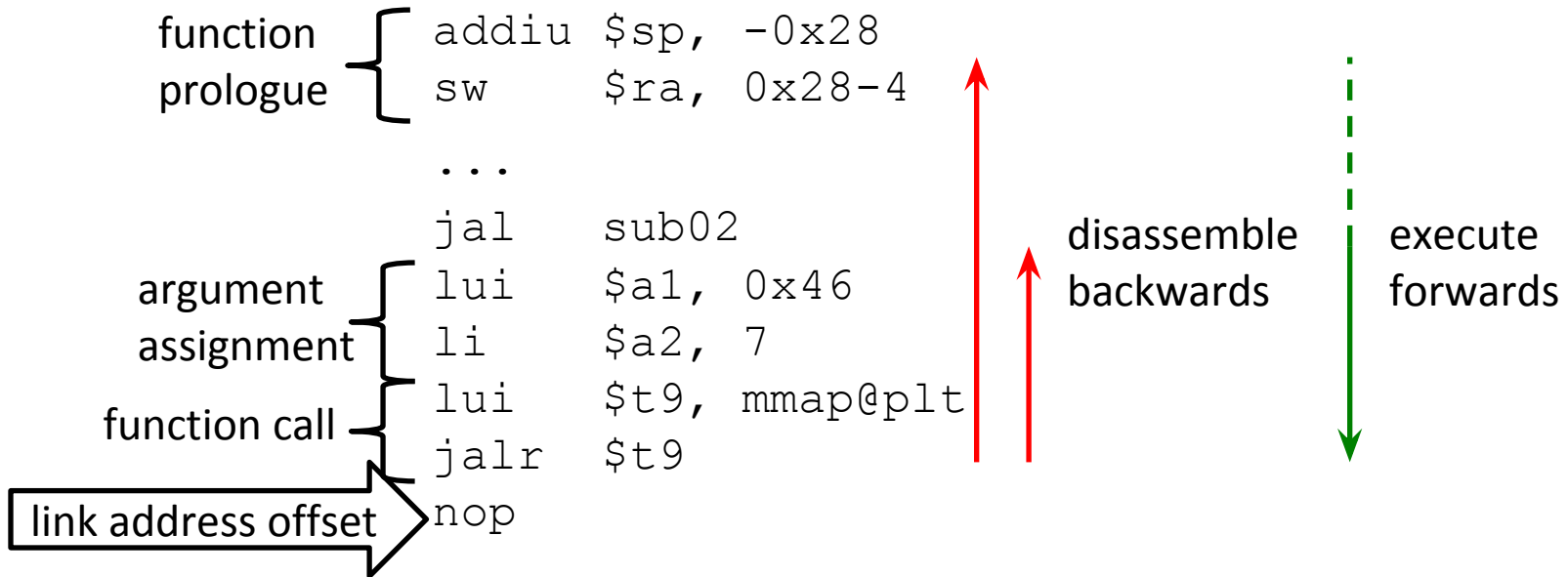
---



- System call return address  $ret_{sc}$
- System call information
  - System call number
  - System call arguments
- Link address  $ret_{lr}$ 
  - specific to RISC
  - register containing return address of last function invocation
- Indirect jump target (on MIPS)



# Code Invariant Extraction



- Lightweight execution state (only registers)
- Invariants = concrete values at end of execution
- Static analysis on the binary executable on disk

# Enforcing Integrity

---

## 1. Code Provenance

- where do function invocations originate from?
- only allow legit locations

## 2. Code Integrity

- is the call chain reflected by the binary?
- do the system call arguments match the invariants?

## 3. Symbol Integrity

- are called system call wrappers actually imported?

# Enforcing Code Provenance

---

- Trusted Application Code Base (TACB)
  - valid code regions of the process runtime image
    - mapped text segments of a running process
    - includes text segments of libraries
  - fixated after linking stage
- Call chain has to originate from the TACB
  - return addresses: both  $ret_{sc}$  and  $ret_{lr}$
  - everything outside TACB is invalid

# Enforcing Code Integrity

---

program code

```
...  
lui    $a3, 0x46  
li     $a0, 7  
lui    $t9, mmap@plt  
jalr   $t9  
nop
```

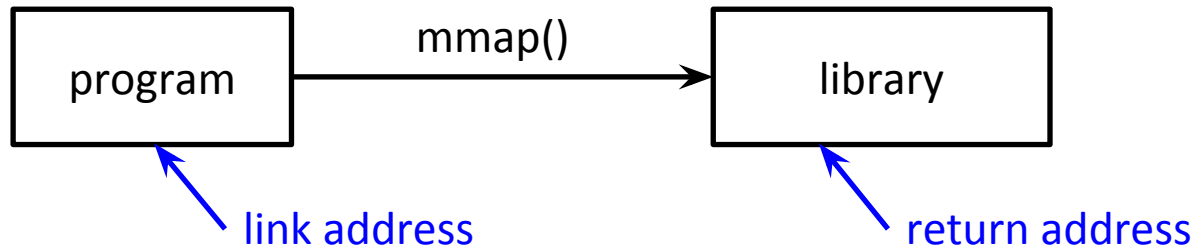
syscall wrapper

```
...  
lw     $t0, 0xcafe  
or     $a3, t0  
li     $v0, 0x101D  
syscall 0  
nop
```

- Is the predecessor of  $\text{ret}_{\text{sc}}$  really a syscall?
  - has the right syscall been invoked?
- Is the predecessor of  $\text{ret}_{\text{lr}}$  really a control flow transfer?
  - does the target of the branch match the callee?
- Do the actual syscall arguments match the invariants?
  - does the syscall wrapper modify arguments?

# Enforcing Symbol Integrity

---



- Dynamic linking uses function symbols
- Symbol mmap has to be
  - exported by the library
  - imported by the program
- Match
  - symbol of function identified by return address
  - imports of binary identified by link address

# Exploit Mitigation

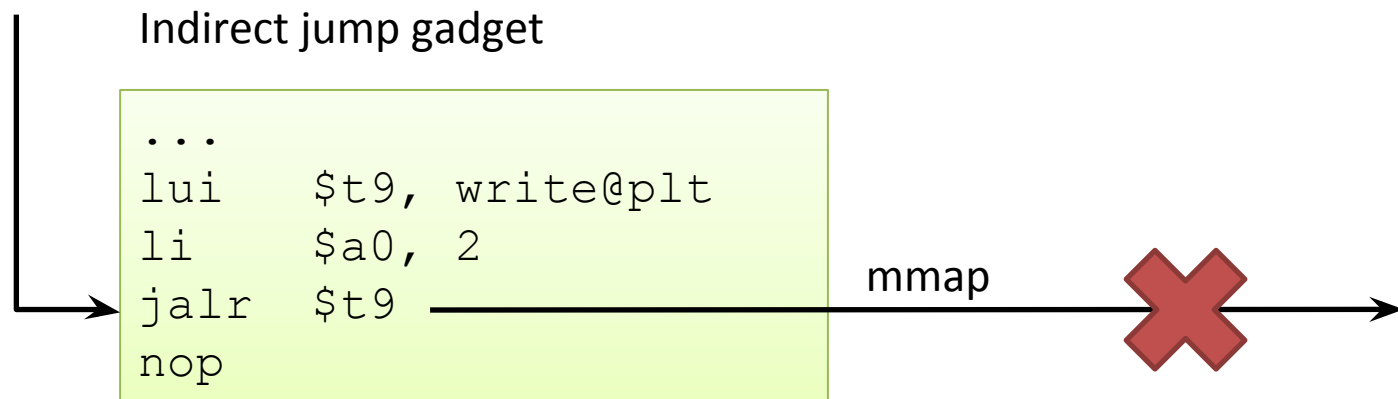
---

Attack class	Technique	Defense
Code injection	inject code in data segment	code provenance
	inject (and overwrite existing) code in text segment	code integrity (instruction mismatch)
Code reuse	use indirect jump gadget	code integrity (target of branch does not match)
		symbol integrity (function not imported)
	use gadget that calls library function	argument integrity (argument mismatch)

# Exploit Mitigation: Code Reuse

---

```
lui $t9, mmap_address
```

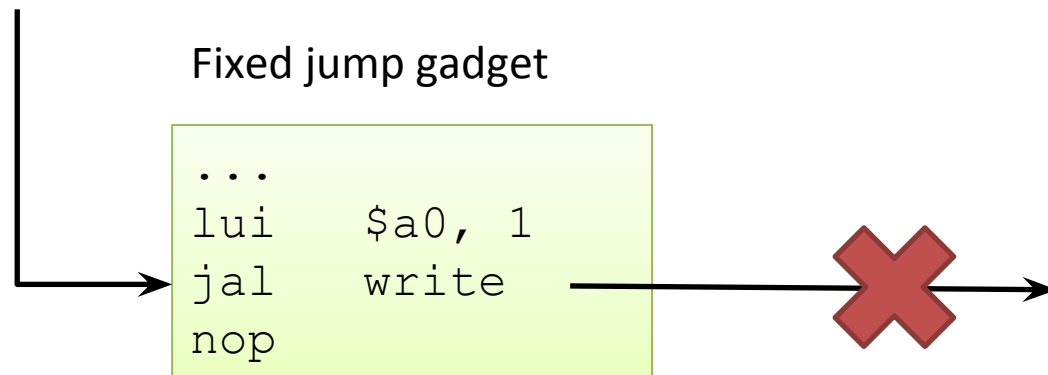


- Violates call chain integrity
  - register `$t9` does not match invariant

# Exploit Mitigation: Code Reuse

---

```
lui $a0, 12
```



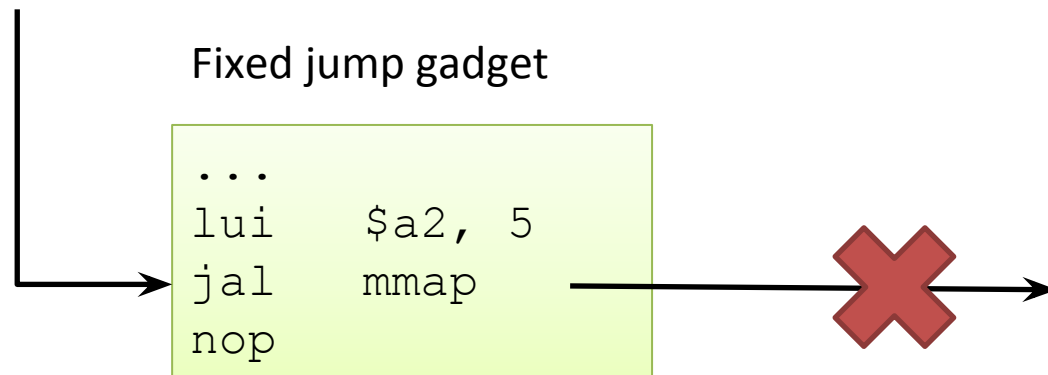
- Violates argument integrity
  - runtime state value for `$a0` contradicts invariant
  - `write` can only access stdout



# Exploit Mitigation: Code Reuse

---

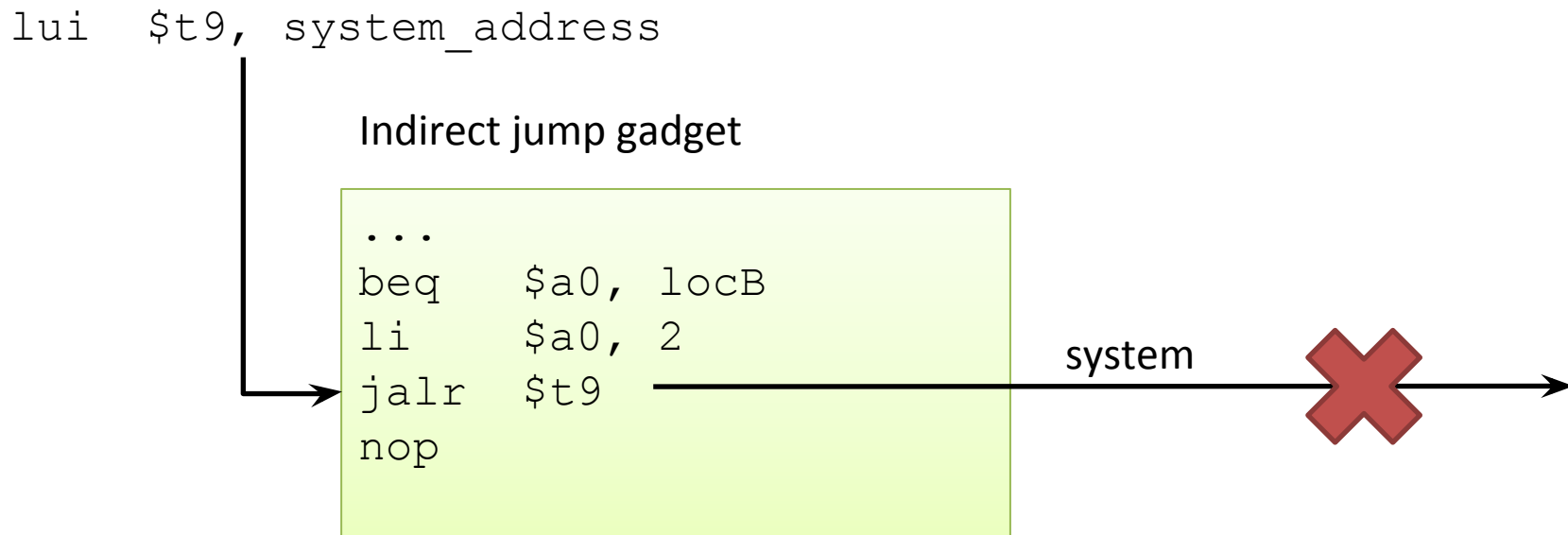
```
lui $a2, 7
```



- Violates argument integrity
  - runtime state value for \$a2 contradicts invariant: RWX (7) vs. RX (5)
  - mmap can only map read/write

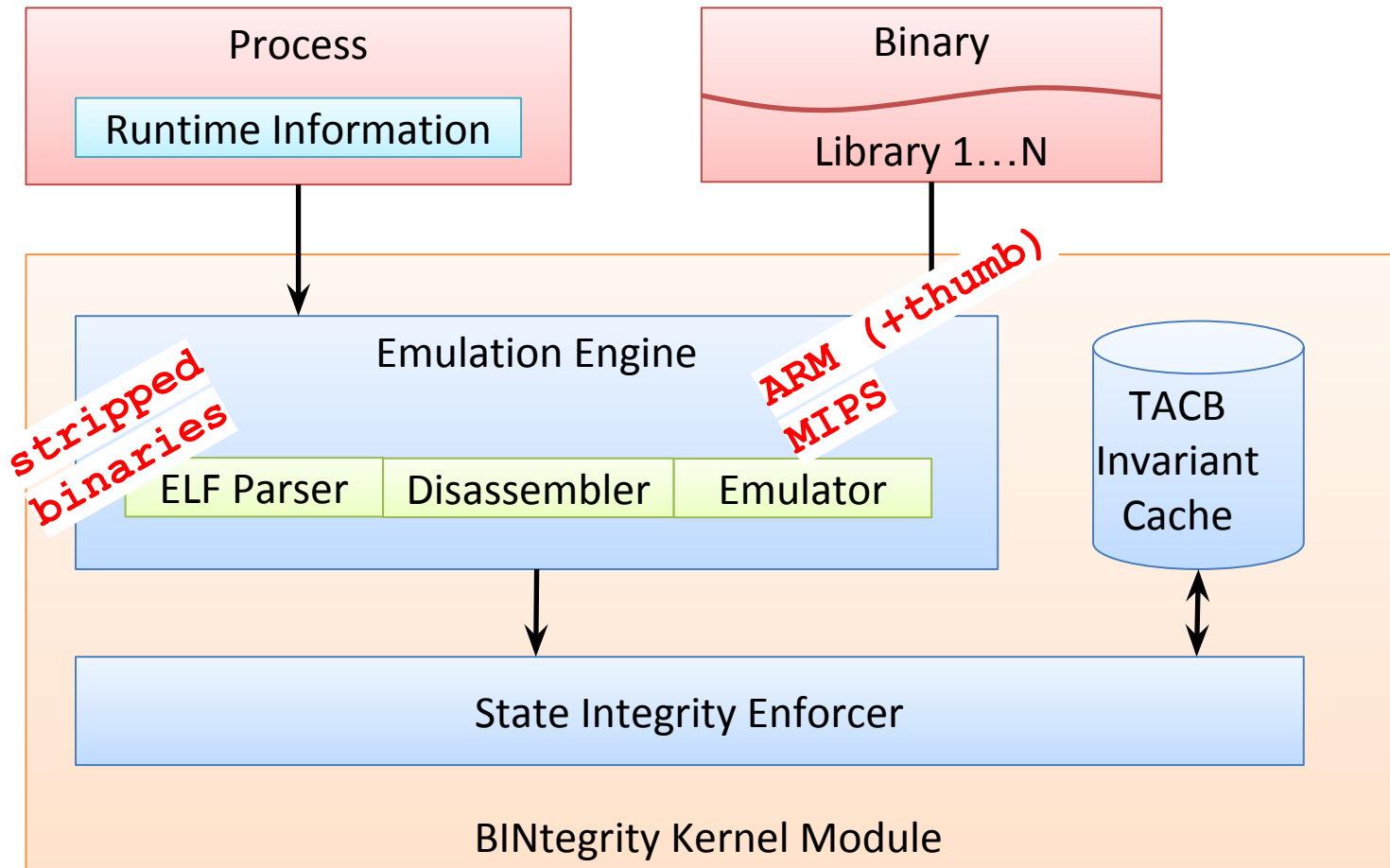
# Exploit Mitigation: Code Reuse

---



- Violates symbol integrity
  - `system` is not imported by the program

# The BINtegrity System



# Performance Evaluation

---

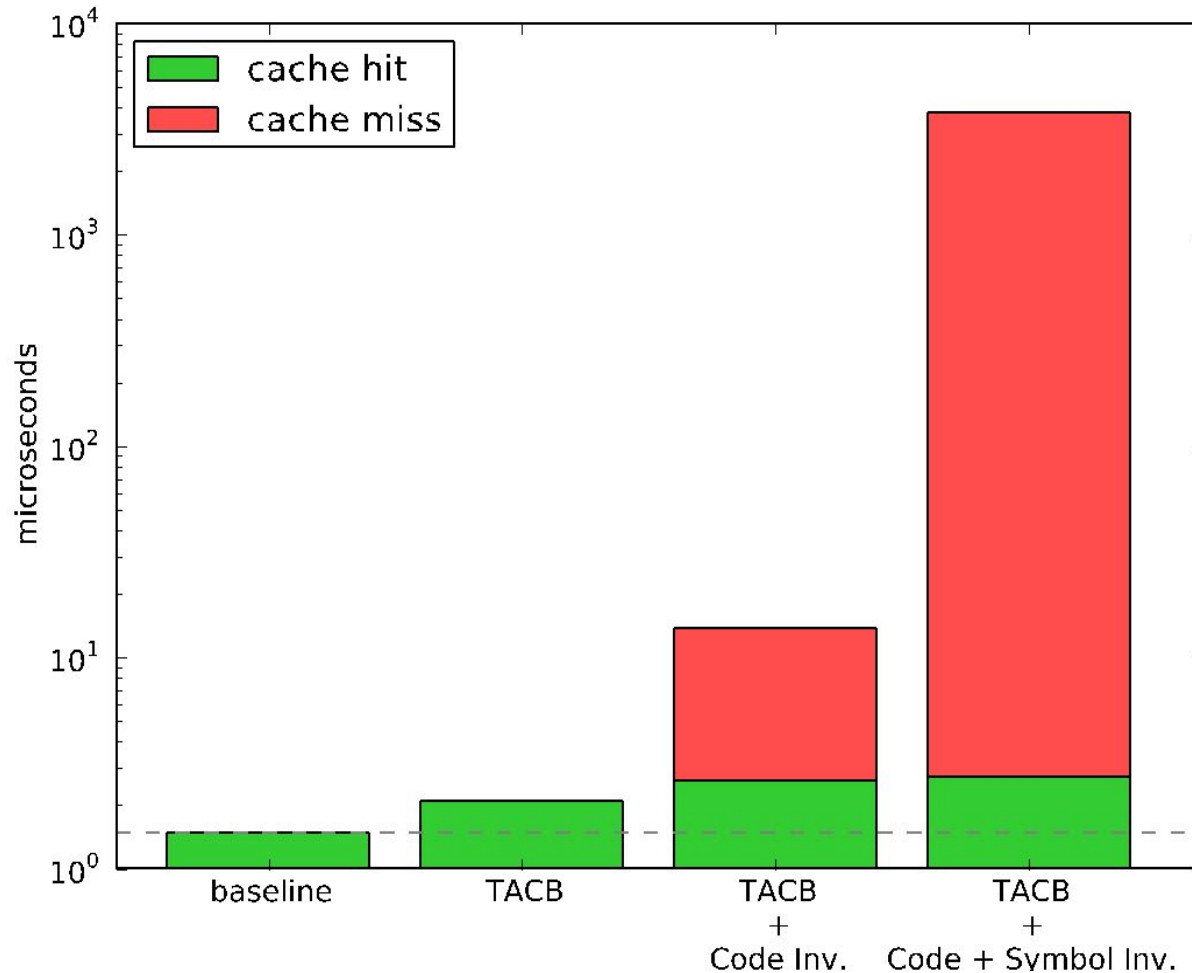
- Buffalo Router WZR-HP-G450H (MIPS)
  - Apache benchmark & nginx
  - runtime overhead: 2.03%
  
- Galaxy Nexus Phone (ARM)
  - AnTuTu benchmark
  - measures Android runtime & I/O subsystem
  - runtime overhead: 1.2%

# Internal Performance Evaluation

---

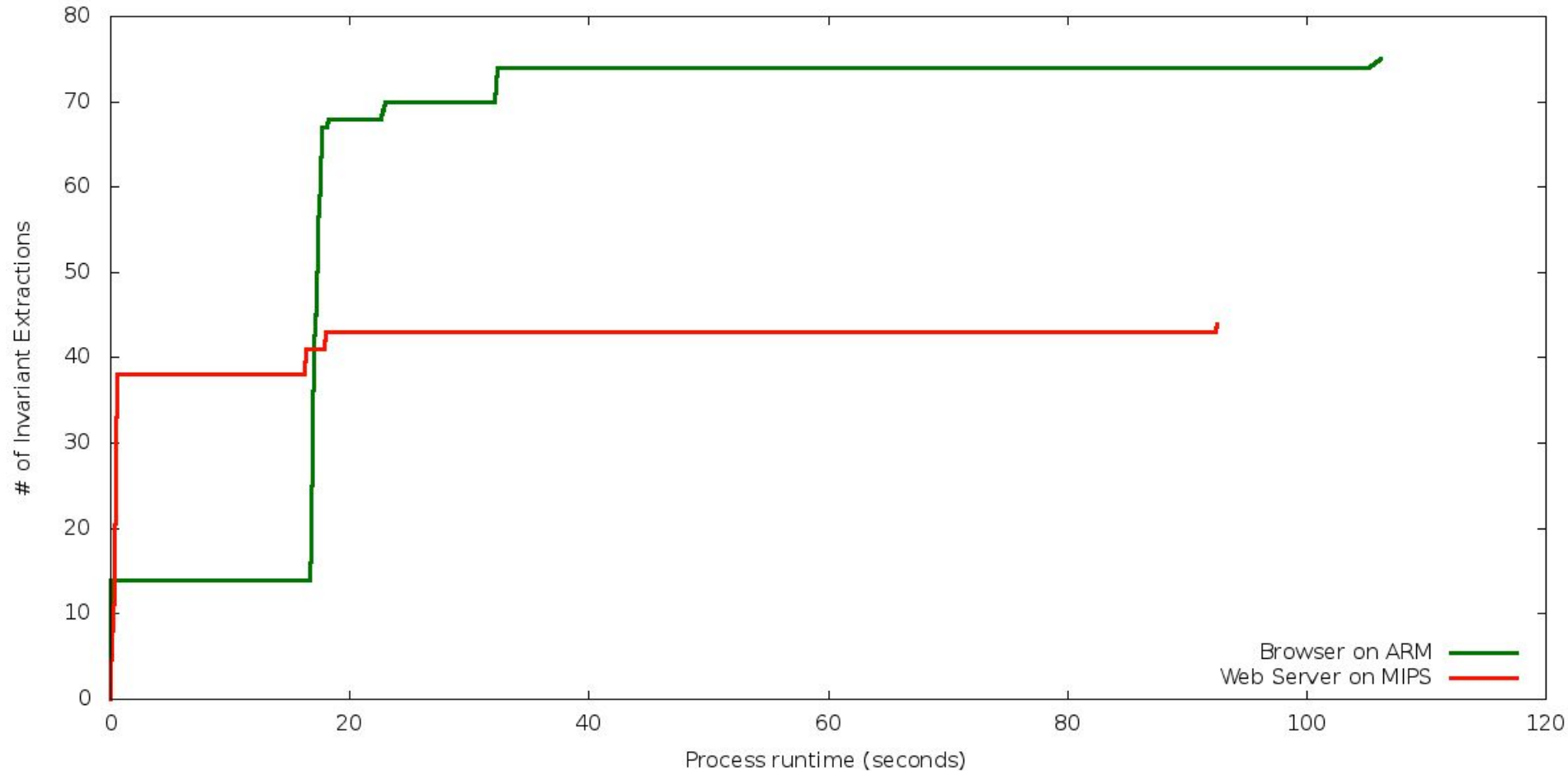
- Costly operations
  - reading and parsing files
  - instruction emulation
- Memory footprint
  - Kernel module code
  - Cache
    - cache invariants for  $< 257$  code points
    - 16 bytes per code point
    - requires total of 12KB per process

# Performance: Caching



# Performance: Invariant Extractions

---



# Conclusions

---

- Cover-all-bases mitigation approach
  - from payload injection, over hijacking control flow, to running the payload
- Practical
  - no rewriting, no instrumentation, no configuration
  - transparent to applications
- Efficient
  - only 2% overhead in application-level benchmarks
- Open source
  - download at <http://www.bintegrity.org/>



# End

---