Practical and Efficient Exploit Mitigation for Embedded Devices

Matthias Neugschwandtner
IBM Research, Zurich

Collin Mulliner
Northeastern University, Boston

Qualcomm Mobile Security Summit 2015

Embedded Devices

Embedded Devices

Embedded Devices

- Produced in large quantities
  - not a computer, but actually a computer
- Mostly low cost RISC-based CPUs
  - exceptions, e.g. CPUs for high-end smart phones
- Devices run open/free software such as Linux
  - Android is Linux, many Smart TVs run Linux

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!**

Mitigations: State of the Art

- Data Execution Prevention (DEP)
  - make memory pages non exec $\Rightarrow$ prevent code injection
  - requires hardware support (emulation is slow)
  - bypassed with code reuse: ret2lib, ROP, ...

- Address Space Layout Randomization (ASLR)
  - move code to “unpredictable” location in memory $\Rightarrow$ prevent code reuse (e.g. ROP)
  - bypassed with information leak, ROP works again
    - Andrea Bittau “BROP - Hacking Blind” (S&P 2013)

Mitigations: State of the Art cont.

- Control Flow Integrity (CFI)
  - detect if “code blocks” are executed “out of order”
    - mitigate code reuse, specifically: ret2lib and ROP
  - need access source code
  - requires compiler support
  - can lead to high overhead

- (Syscall) Policy enforcement
  - SELinux, AppArmor, syscall anomaly detection
  - per app configuration and/or learning

Hardware and Mitigations

- Hardware support for DEP
  - **x86:** No-eXecute (NX)
  - high-end **ARM** SoCs: eXecute Never (XN)

- PAX can emulate DEP
  - but only on IA32
  - MIPS declared a “hopeless case” (https://pax.grsecurity.net/docs/pax.txt)

- <u>Embedded CPU/SoC lack mitigation support</u>

(Mis)Using Hardware Features

- Many platform and architectural features, why not use them for security?
  - Advantages: precision, speed, harder to circumvent

- Last Branch Record (LBR) for ROP detection
  - Vasilis Pappas “kBouncer” 2012

- PMC for mispredicted returns for ROP detect.
  - Georg Wicherski “Taming ROP on Sandy Bridge” 2013

(Mis)Using Hardware Features

- Many platform and architectural features, why not use them for security?
  - Advantages: precision, speed, harder to circumvent

- Last Branch Record (LBR) [illegible] ROP detection
  - Vasilis Pappas “kBoun [illegible]

- PMC for [illegible]cted returns for ROP detect.
  - Georg Wic[illegible]ski “Taming ROP on Sandy Bridge” 2013

x86 only

Can we Leverage RISC Features?

- Use common hardware features for security!
  - More precision, better performance, hard to circumvent
- Many RISC flavors
  - ARM, MIPS, SuperH, PA-Risc, Sparc
- Use generic features $\Rightarrow$ broad application
  - Avoid SoC specific functionality

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

Goal: Bring SotA Mitigations to embedded RISC devices

- Build “replacements” for SotA mitigations
  - e.g. DEP and CFI
- Use RISC hardware features
  - speed and precision
- Tailor for “binary only” / COTS
  - source code is not always available

Binary Integrity

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

Binary Integrity

- 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**
  - **system call is actually used**
  - **system call arguments match**
  - **call chain matches**

Binary Integrity

BINtegrity

- **Ensure that <u>runtime</u> OS usage is coherent with OS usage in <u>binary executable</u>**
  - **system call is actually used**
  - **system call arguments match**
  - **call chain matches**

A Different Angle

- Policy based solutions
  - AppArmor, SELinux
  - what resources/OS services can be used
- Policy needs to be defined
  - create manually or via learning
  - too wide … attacker can bypass
  - too narrow … app doesn't work correctly
- Application update $\Rightarrow$ policy update!
  - otherwise application stops working

A Different Angle

- The application **binary** is the policy
  - binary provides all information about what it is doing*
- Enforce restrictions using the binary image
  - Track program’s “runtime state”
  - Compare with state extracted from binary image
  - <u>Non matching states ⇒ attack</u>
  - binary update == policy update *#win*

**information needs to be extracted and understood to be useful*

BINtegrity: Core Features

- DEP like functionality
  - only execute code that is present on disk
- Super lightweight CFI
  - extract and compare call chain with code on disk
- Syscall filter / policy
  - syscall can only be invoked if application uses it

BINtegrity: Core Features

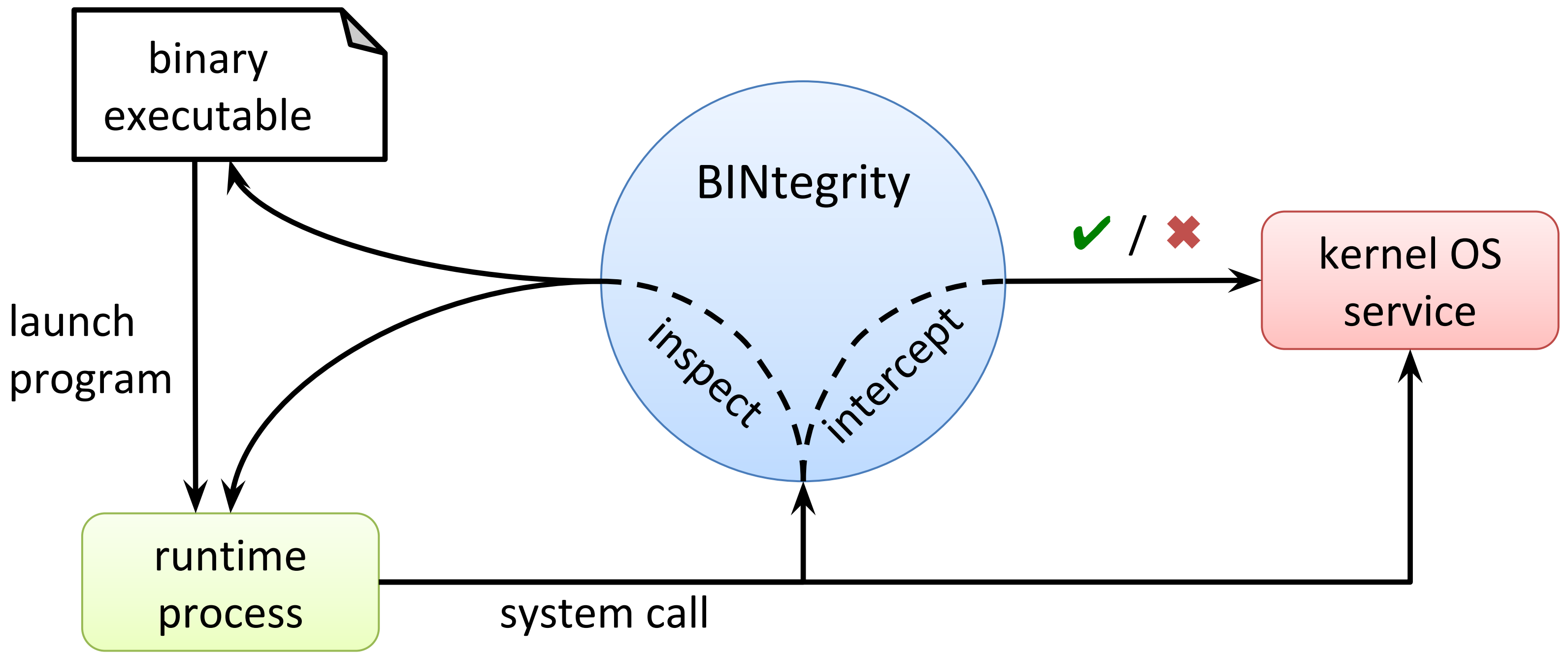
- DEP like functi[illegible]ity
  - only exec[illegible]hat is present
- [illegible]
  - [illegible]n code on disk
- S[illegible]icy
  - [illegible]y be invoked if application uses it

binary only
no rewrite
no instrumentation
no configuration

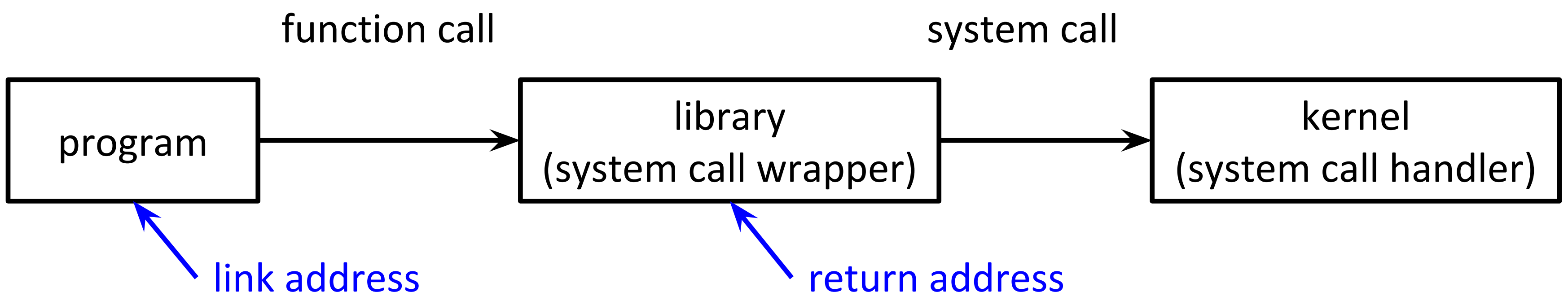
Threat Model

- 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

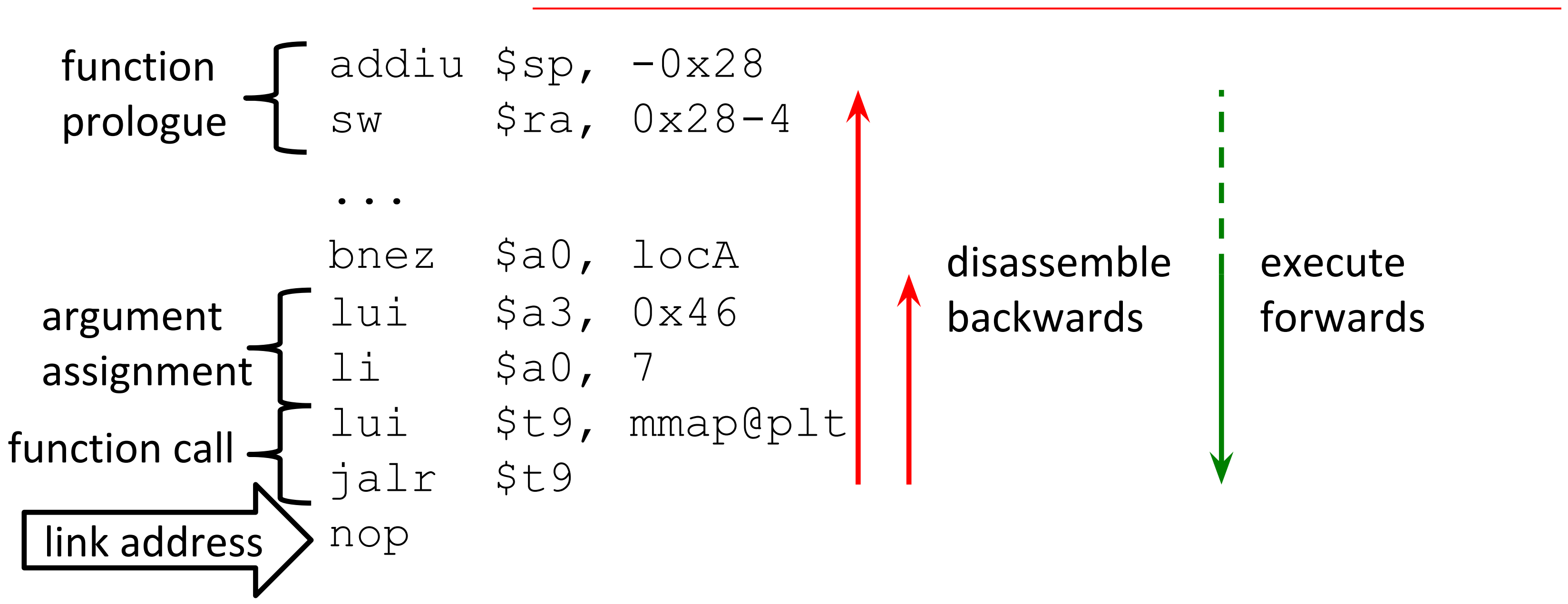


Process Runtime State



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

Code Invariant Extraction



```
addiu $sp, -0x28
sw    $ra, 0x28-4
...
bnez  $a0, locA
lui   $a3, 0x46
li    $a0, 7
lui   $t9, mmap@plt
jalr  $t9
nop
```

- Leightweight execution state (only registers)
- Invariants = concrete values at end of execution

Enforcing Integrity

- Code Provenance
  - where do function invocations originate from?
- Code Integrity
  - is the call chain reflected by the binary?
  - do the system call arguments match the invariants?
- Symbol Integrity
  - are called system call wrappers actually imported?

Enforcing Code Provenance

- Trusted Application Code Base (TACB)
  - mapped text segments of a running process
  - includes text segments of libraries
  - fixated after linking stage
- Return addresses have to point to TACB
  - both $\texttt{ret}_{\texttt{sc}}$ and $\texttt{ret}_{\texttt{lr}}$

Enforcing Code Integrity

program code

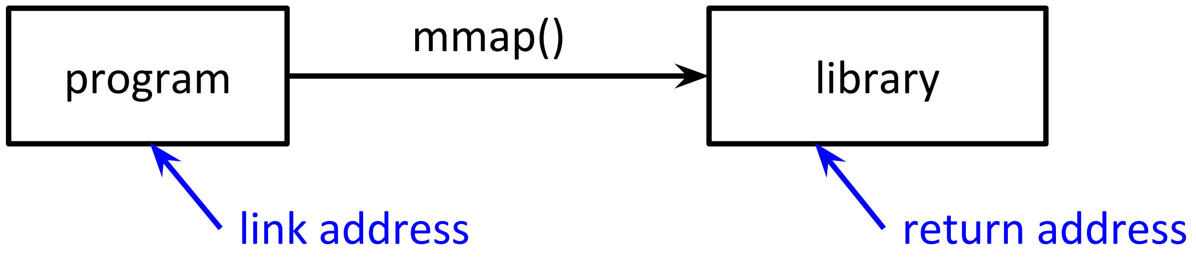
```
...
lui    $a3, 0x46
li     $a0, 7
lui    $t9, mmap@plt
jalr   $t9
nop
```

syscall wrapper

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

- Is the predecessor of $\mathtt{ret}_{\mathtt{sc}}$ really a syscall?
  - has the right syscall been invoked?
- Is the predecessor of $\mathtt{ret}_{\mathtt{lr}}$ 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

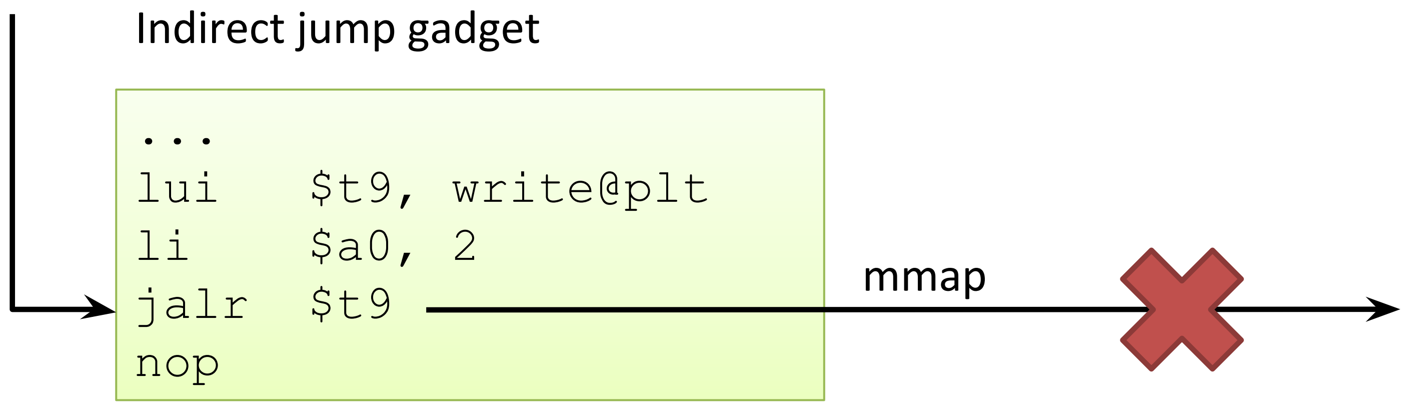


- Dynamic linking uses function symbols
- Symbol mmap has to be
  - exported by the library
  - imported by the program
- Match
  - symbol of function identified by return address
  - imports of binary identified by link address

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)

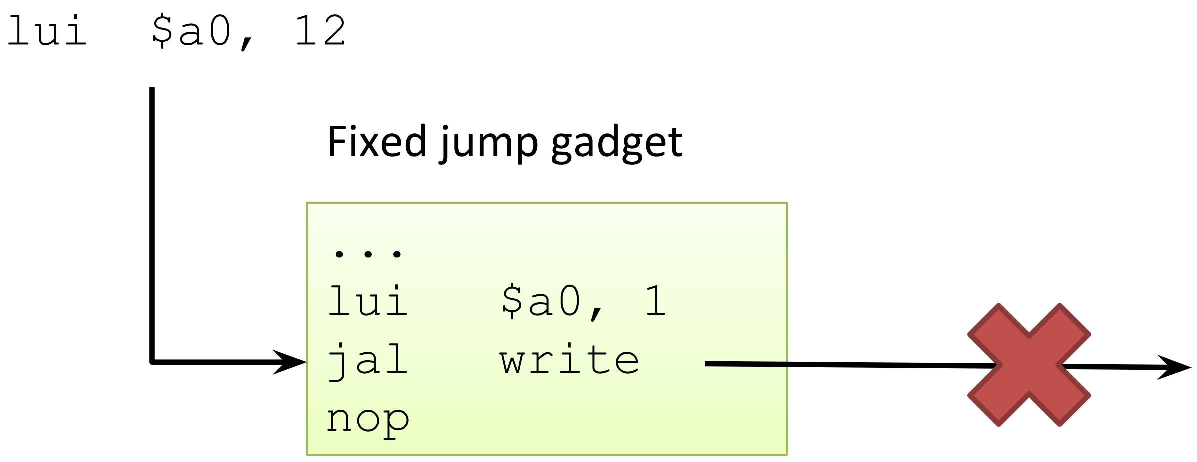
Exploit Mitigation: Code Reuse



```
...
lui    $t9, write@plt
li     $a0, 2
jalr   $t9
nop
```

- Violates control flow integrity
  - register `$t9` does not match invariant

Exploit Mitigation: Code Reuse



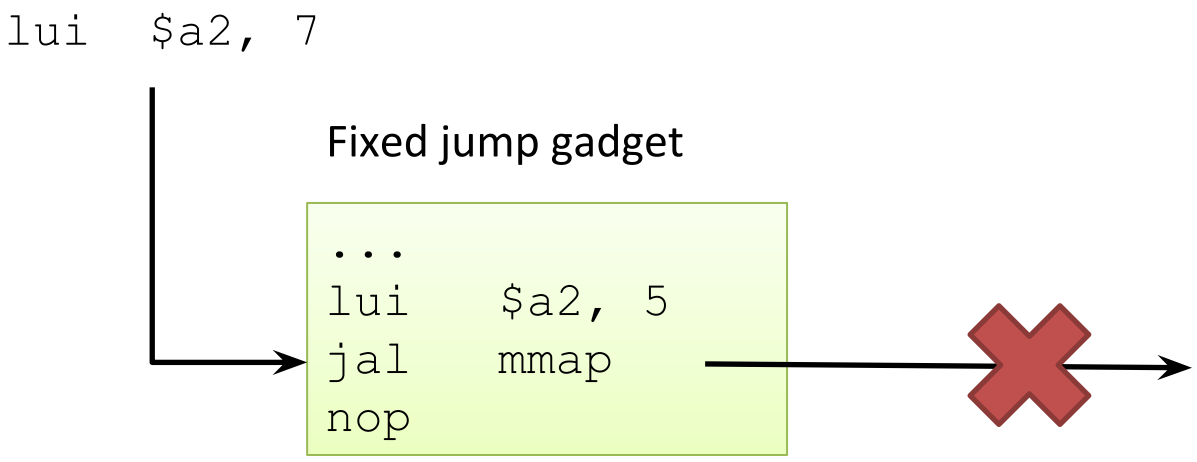
```
lui   $a0, 12
```

Fixed jump gadget

```
...
lui   $a0, 1
jal   write
nop
```

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

Exploit Mitigation: Code Reuse

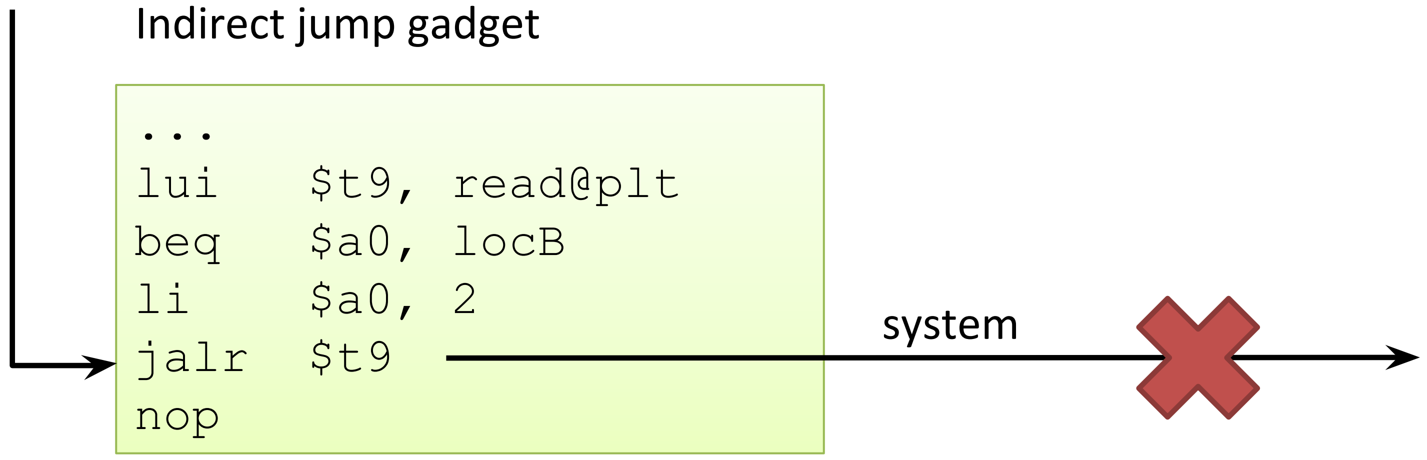


```
lui   $a2, 7
```

```
...
lui    $a2, 5
jal    mmap
nop
```

- 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



```
...
lui   $t9, read@plt
beq   $a0, locB
li    $a0, 2
jalr  $t9
nop
```

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

Exploit Mitigation: ROP stager

Combination of ROP and “traditional” shellcode

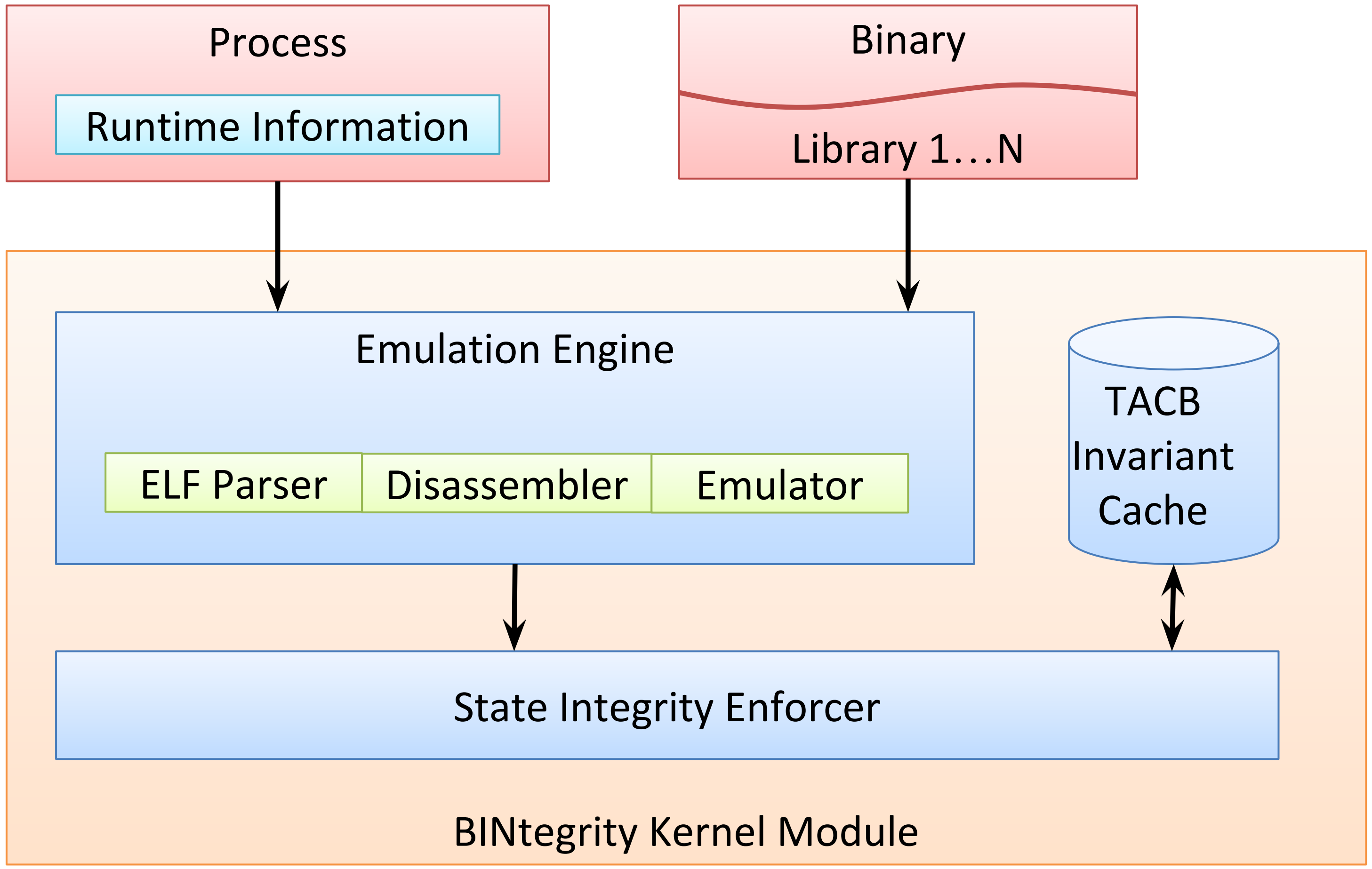
1. Use ROP for set up
   a. executable memory region (mmap and/or mprotect)
   b. on MIPS: flush cache

Code reuse mitigation

2. Execute “traditional” shellcode from separate memory region

Code injection mitigation

The BINtegrity System



Checking Level

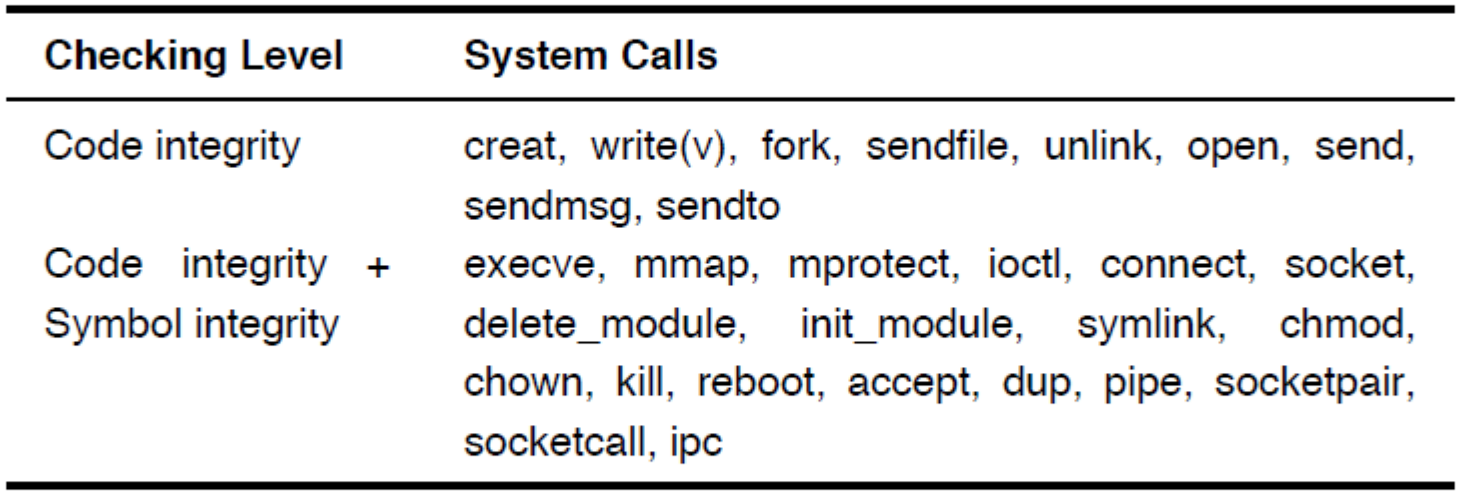
- Not all functions need all checks

⇒ reduce checking to increase performance

Level 1	Code Provenance
Level 2 (includes L1)	Code Integrity
Level 3 (includes L2)	Symbol Integrity

Syscalls vs Checking Levels

- 33 security critical syscalls
- 11 at checking level 2
- 22 at checking level 3



Checking Level	System Calls
Code integrity	creat, write(v), fork, sendfile, unlink, open, send, sendmsg, sendto
Code integrity + Symbol integrity	execve, mmap, mprotect, ioctl, connect, socket, delete_module, init_module, symlink, chmod, chown, kill, reboot, accept, dup, pipe, socketpair, socketcall, ipc

Dynamic Library Loading

- dlopen() vs BINtegrity
  - implemented via mmap()
  - mmap() is a Level 3 call
  - check if mmap() was invoked by dlopen()
  - check if dlopen() is found in imported symbols (Level 3)

- Add new library to TACB
  - no symbol integrity checks on calls to this library

Performance Evaluation

- Buffalo Router WZR-HP-G450H (MIPS)
  - Apache benchmark & nginx
  - <u>runtime overhead: 2.03%</u>

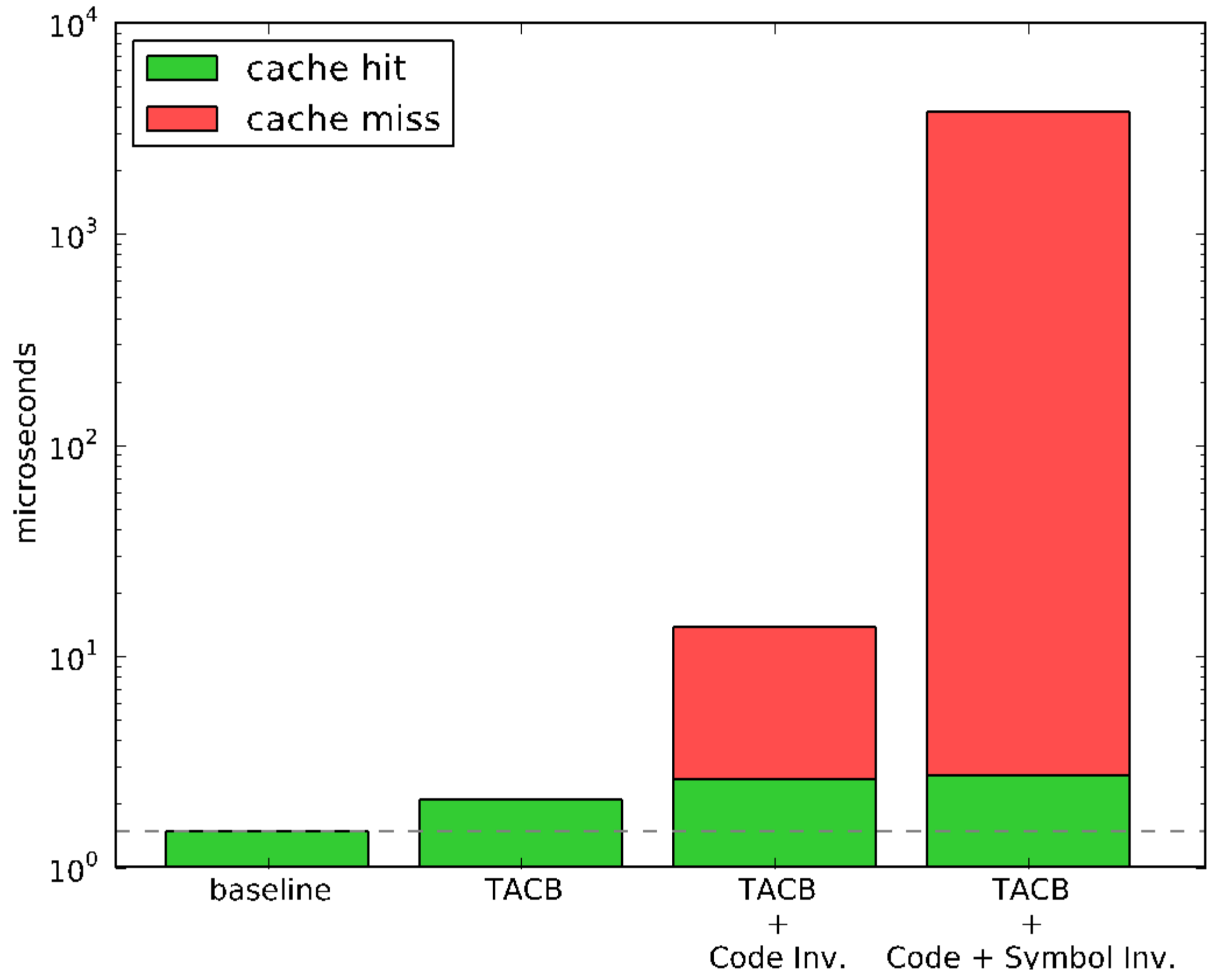
- Galaxy Nexus Phone (ARM)
  - AnTuTu benchmark
  - measures Android runtime & I/O subsystem
  - <u>runtime overhead: 1.2%</u>

Internal Performance Evaluation

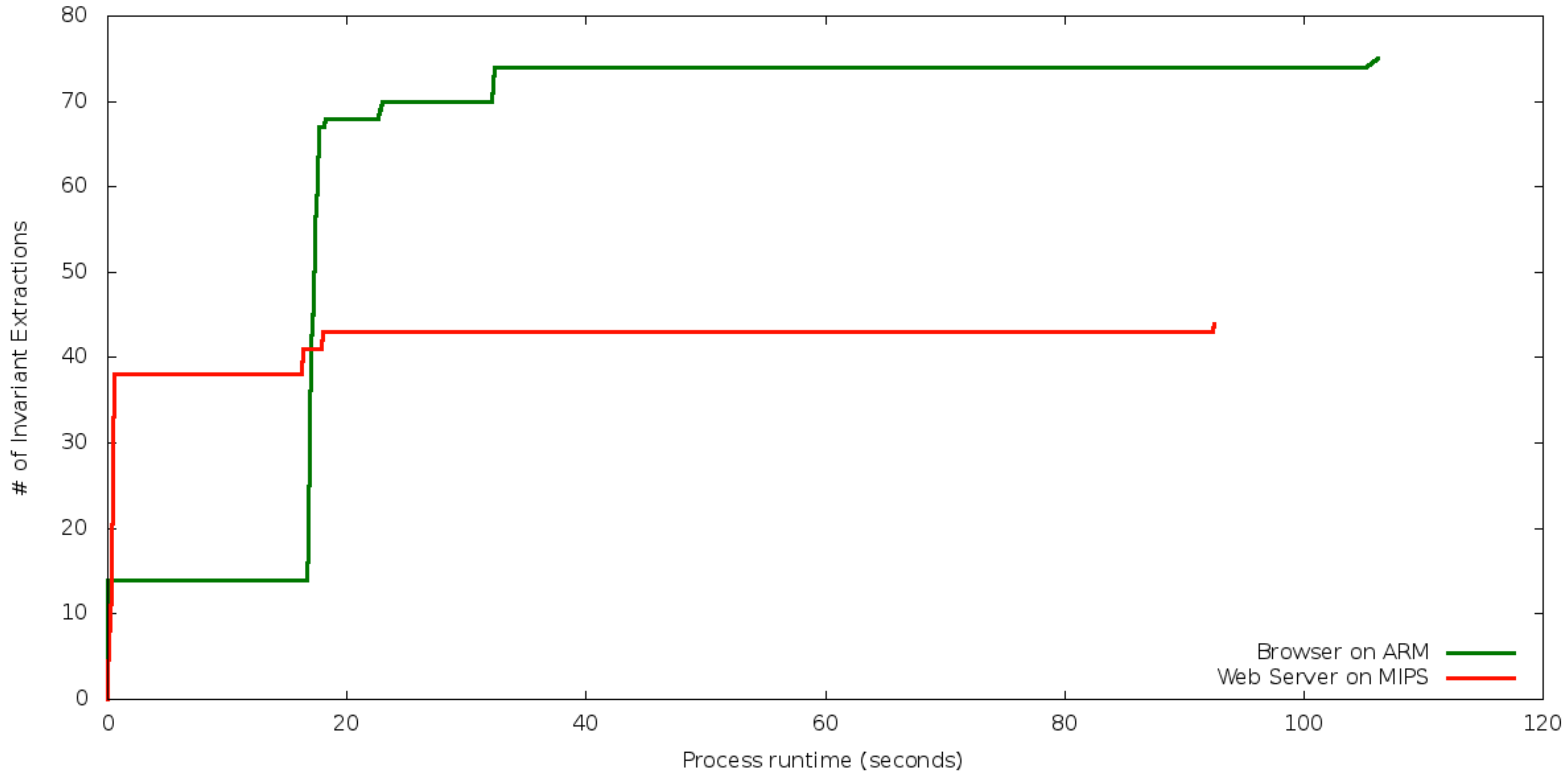
- Costly operations
  - reading and parsing files
  - instruction emulation

- Memory footprint
  - cache invariants for < 257 code points
  - 16 bytes per code point
  - requires total of 12KB per process

Performance: Caching



Performance: Invariant Extractions



Limitations

- Library implementation
  - generic system call wrapper
  - wrappers that alter arguments or use indirections
  - can be solved by re-compiling libc

System call wrappers	Bionic	uClibc
Total	194	243
Using indirections	71	31
Modifying arguments	1	69

Limitations

- Dynamic code loading
  - reduces effectiveness of symbol integrity
- Link address validity
  - could be forged
  - difficult to do in practice
    - forged address needs to pass integrity checks
    - attacker needs to regain control

Conclusions

- Use architectural features to improve security
  - specifically for platforms <u>without</u> hw security features
- BINtegrity provides
  - DEP like functionality
  - Super lightweight CFI for binary only applications
  - Syscall filter / policy extract from binary image
- Practical and efficient: only 1% - 2% overhead
  - transparent to applications (supports binary only / COTS)

Conclusions

- Use architectural features to i rove security
  - specifically for pl urity features

Thank you!

- BINtegrity provides
  - DEP like functionality
  - Super lightweight CFI f ications
  - Sysca nage

Questions?

- Practical and efficient: only 1% - 2% overhead
  - transparent to applications (supports binary only / COTS)