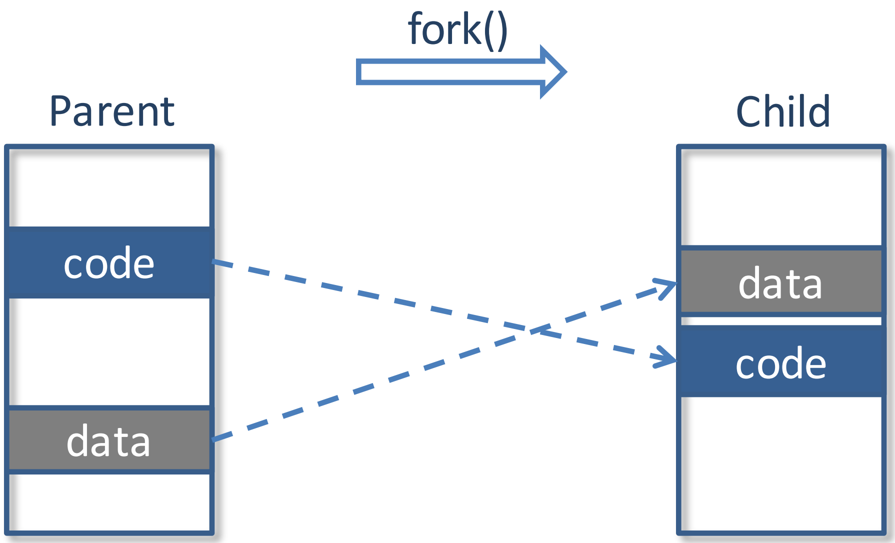
How to Make ASLR Win the Clone Wars: Runtime Re-Randomization

Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee

Georgia Tech, CISPA, Saarland University, MPI-SWS, DFKI

What did we do?

- We re-randomize the memory layout of the cloned (*i.e.*, forked) processes at **runtime**

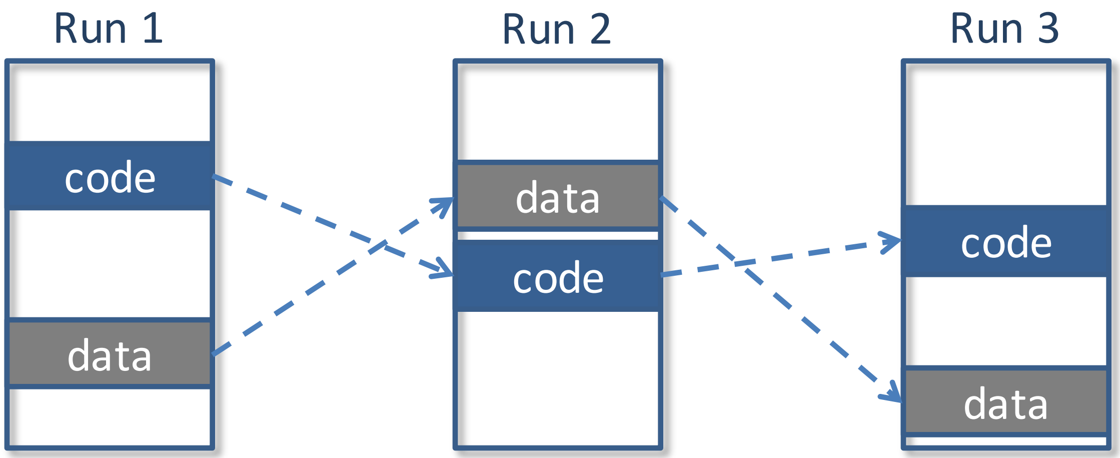


In this talk, I will explain…

- **Why** we need to re-randomize cloned processes?
  – To prevent clone-probing attacks
- **How** to re-randomize them?
  – A semantic-preserving and runtime-based approach
- **What** are the results?
  – Defeated clone-probing, e.g., Blind ROP attack
  – No performance overhead to cloned processes

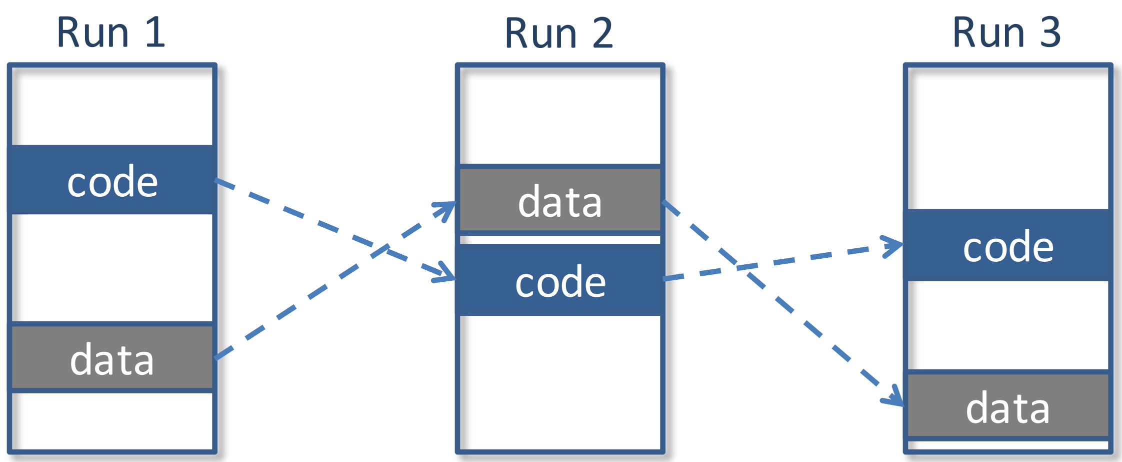
Background - ASLR

- Address Space Layout Randomization (ASLR)
  - Mitigating code reuses attacks, privilege escalation, and information leaks



Background - ASLR

- Address Space Layout Randomization (ASLR)
  - Mitigating code reuses attacks, privilege escalation, and information leaks

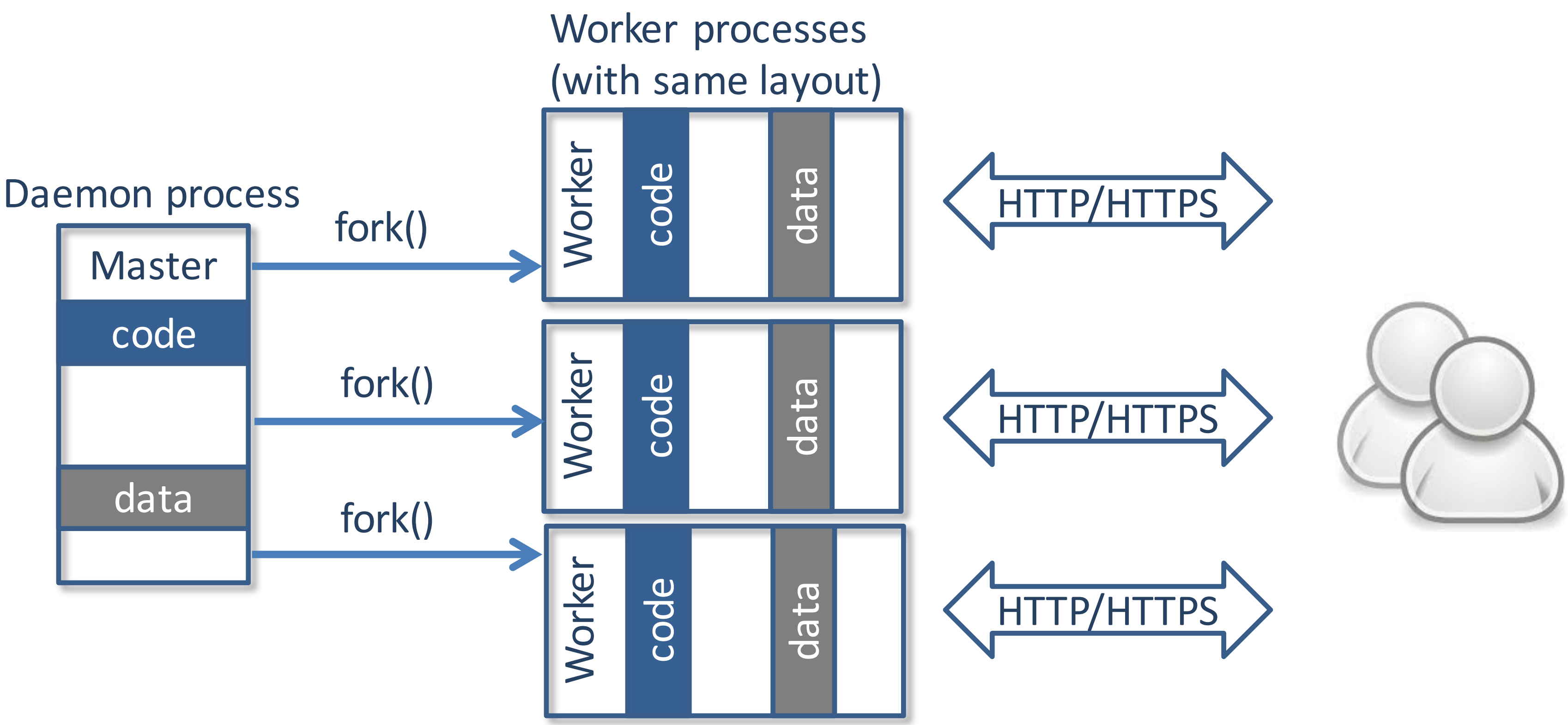


  - One time, per-process, load-time

Background – Daemon Servers

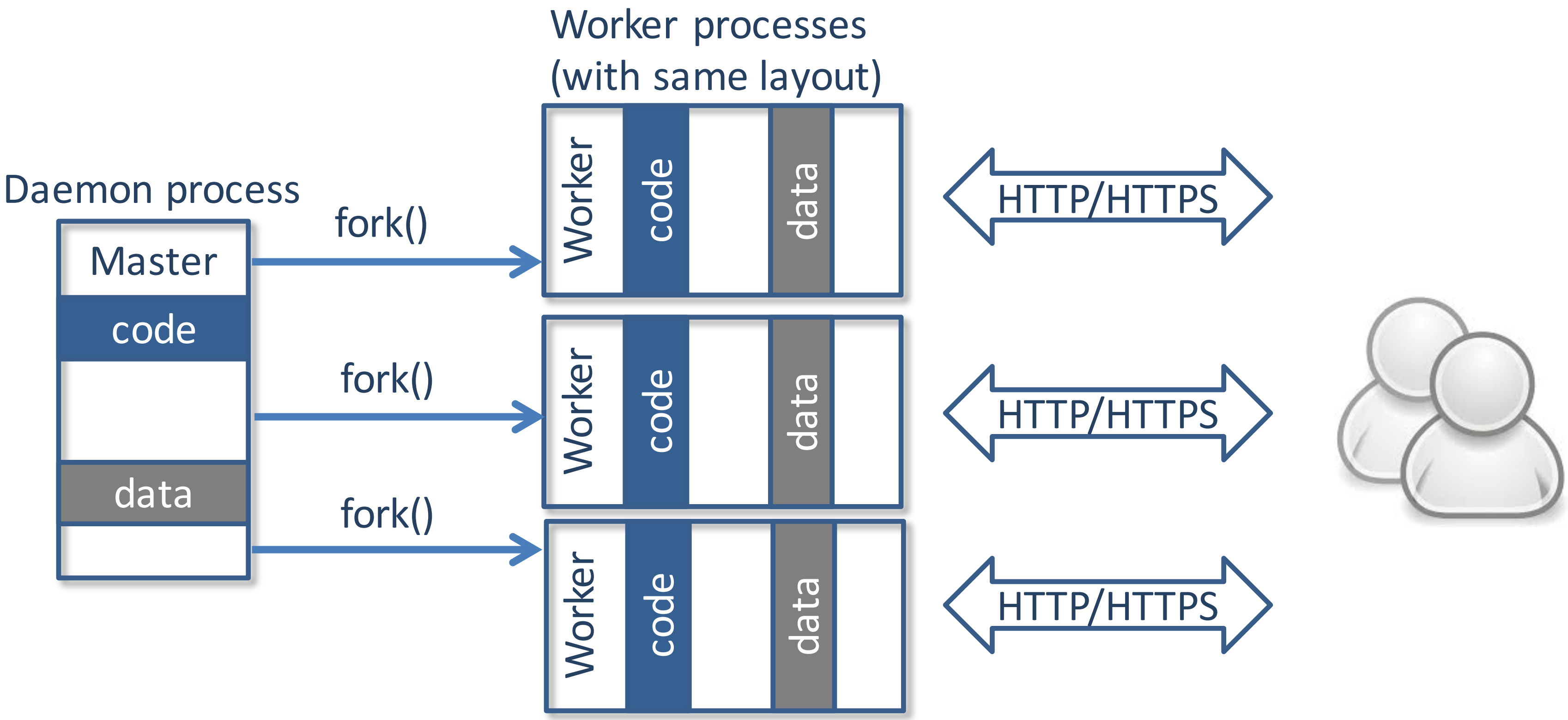
- Web services are powered by daemon servers, e.g., Nginx web server

Designs of Daemon Server



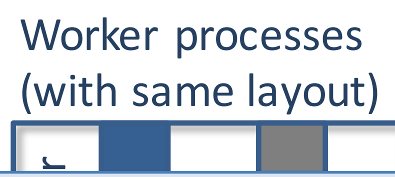
1) The daemon process pre-forks multiple worker processes that handle users requests

Designs of Daemon Server



1) The daemon process pre-forks multiple worker processes that handle users requests
2) The daemon will re-fork a new worker process if it crashes, to be robust

Designs of Daemon Server



All forked worker processes share the same memory layout as the daemon process

2) The daemon will re-fork a new worker process if it crashes, to be robust

When ASLR meets daemon servers…

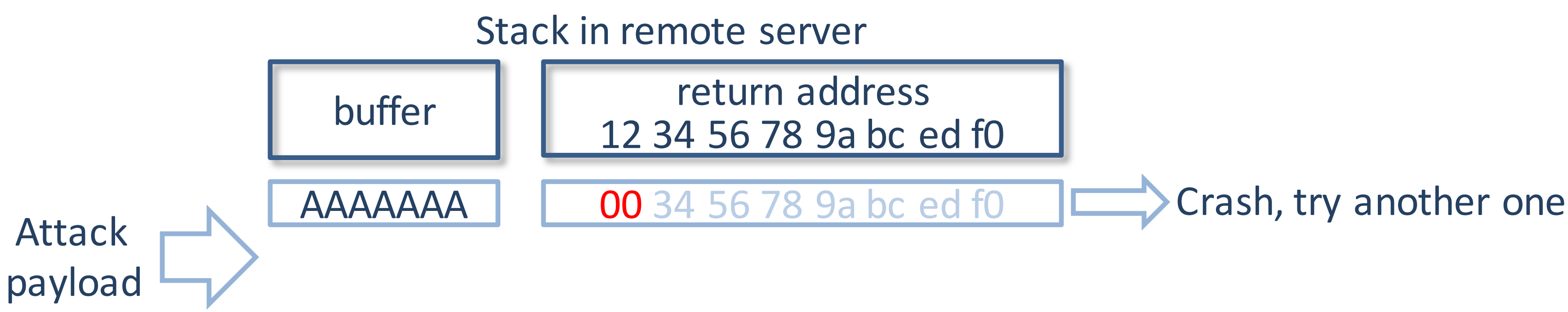
Clone-Probing Attack

- Attack goal: guess the randomized address (e.g., return address), say a web server with a stack buffer overflow vulnerability

Stack in remote server

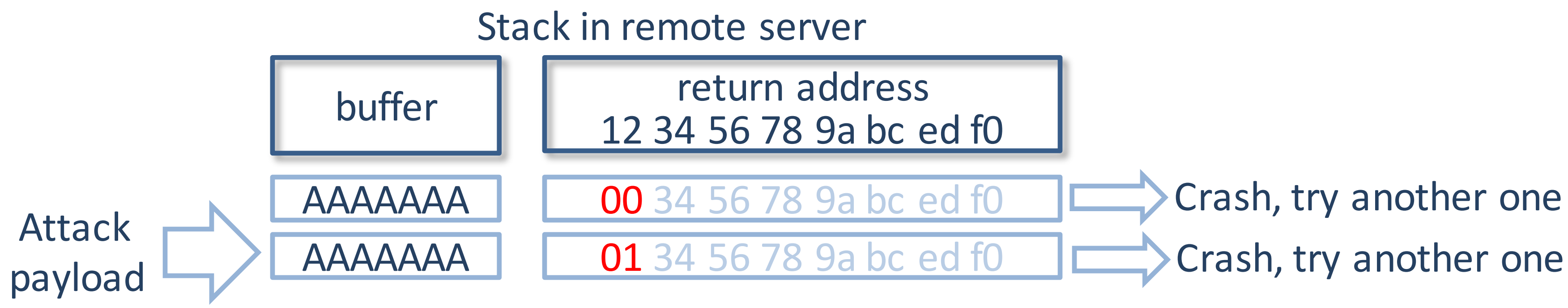
Clone-Probing Attack

- Attack goal: guess the randomized address (e.g., return address), say a web server with a stack buffer overflow vulnerability



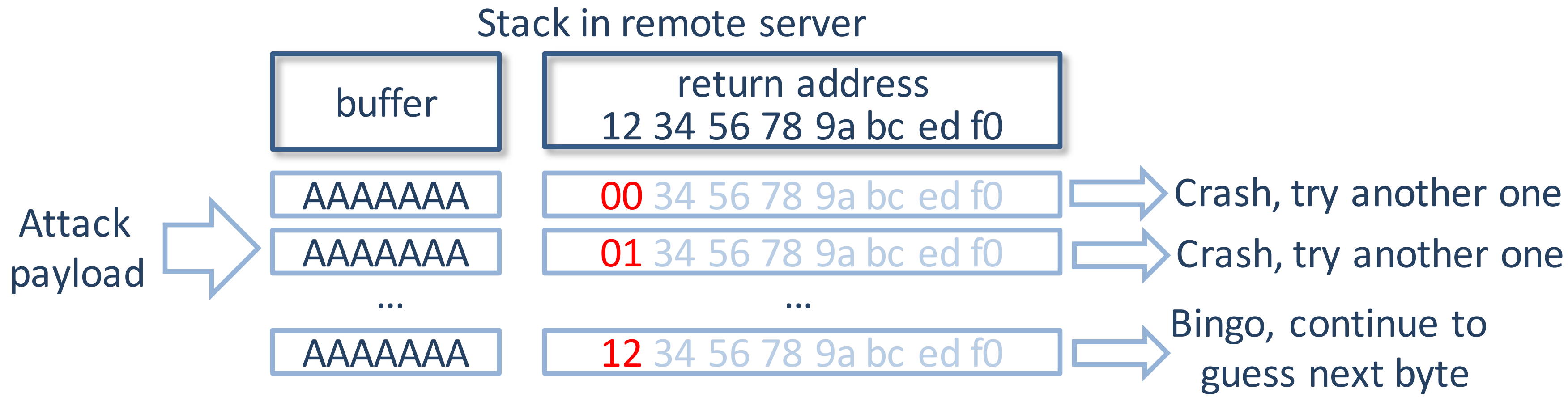
Clone-Probing Attack

- Attack goal: guess the randomized address (e.g., return address), say a web server with a stack buffer overflow vulnerability



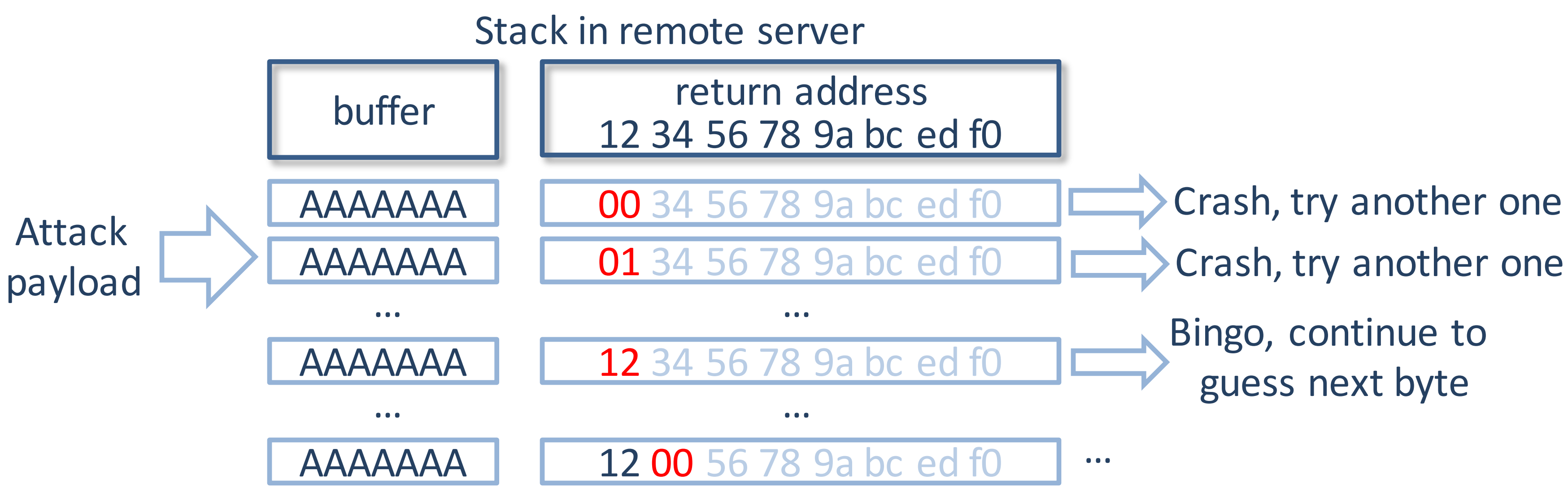
Clone-Probing Attack

- Attack goal: guess the randomized address (e.g., return address), say a web server with a stack buffer overflow vulnerability



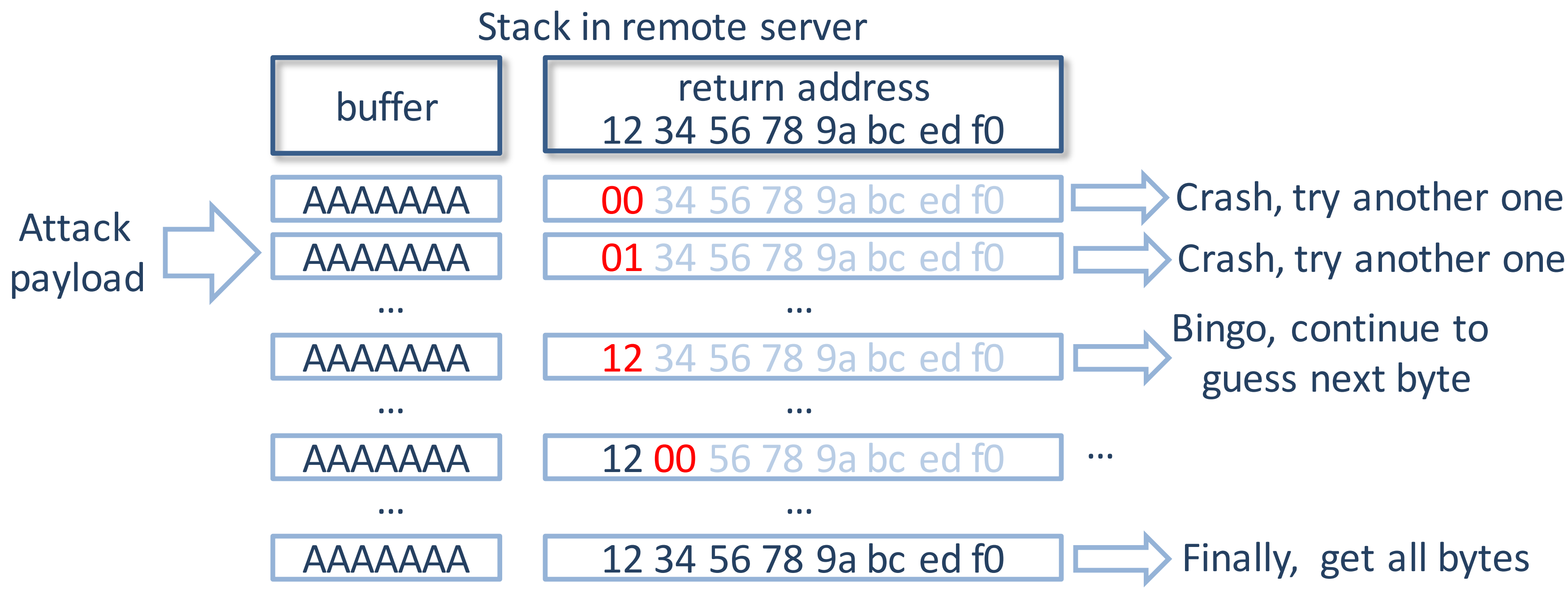
Clone-Probing Attack

- Attack goal: guess the randomized address (e.g., return address), say a web server with a stack buffer overflow vulnerability



Clone-Probing Attack

- Attack goal: guess the randomized address (e.g., return address), say a web server with a stack buffer overflow vulnerability



Clone-Probing Attack

- Attack goal: guess the randomized address

Brute-forcing complexity is reduced from 2^{64} to $8*2^{8}$

(From thousands of years to 2 minutes ☺)

AAAAAAA	12 34 56 78 9a bc ed f0	⇨ Finally, get all bytes

This Attack is Critical!

A simple buffer overflow→ bypass ASLR (two minutes)→ control daemon server ☹

Preventing clone-probing with RuntimeASLR

Solution: re-randomizing the memory layout of cloned processes

Challenge

- Remapping memory → dangling pointers

- How to track all pointers on the fly and update them?
  - Accuracy
  - Efficiency

Pointer Tracking Problem

- Treat it as a taint tracking problem

Source Pointers

- Kernel routinely loads program
  - Easy to find source pointers
- Only in stack and registers

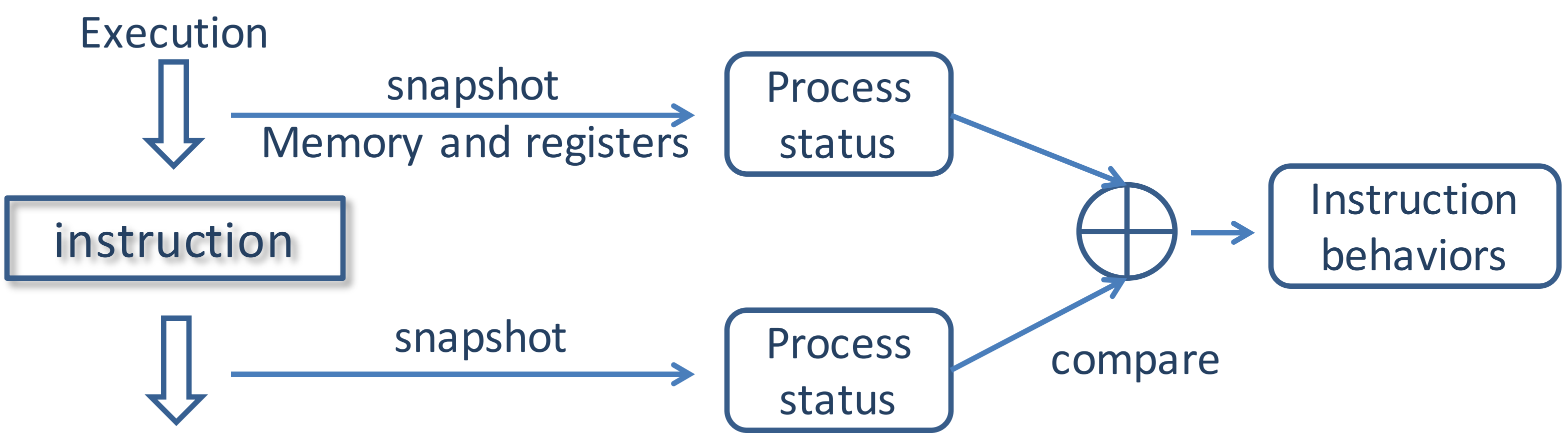
Pointer Tracking Policy

Pointer Tracking Policy

- Read 1,513- pages Intel ISA manual and manually define them??

Automatic Tracking Policy Generation

- Automatically identifying instructions behaviors



- This way, we know if it generates or destroys some “values”

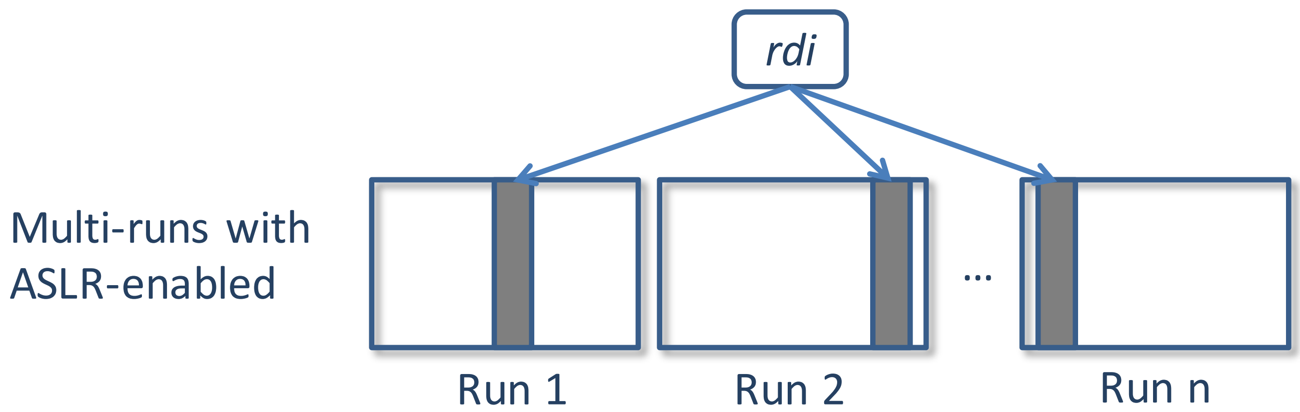
How to Determine a Pointer?

- Without type info, how do we know if a value is a pointer?

- Example: *mov rdi, rsp*
  - Before: *rsp*=0xcafebabe, and know it is a pointer
  - After: *rdi*=0xcafebabe, memory is unchanged
  - How to know if *rdi* is a pointer?

Multi-Run Pointer Verification

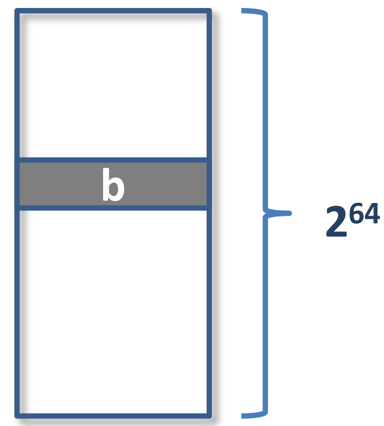
- Observation: *rdi* is likely a pointer if it points to mapped memory on 64-bits platform, why?
- Run program *n* times with ASLR, if *rdi* always points to mapped memory, *rdi* is more and more likely a pointer
  - Mapping *n* runs with instruction execution sequence



Accuracy of Multi-Run Verification

- Assume size of mapped memory is ***b*** bytes, run ***n*** times on 64-bits platform, false positive rate for one value is:

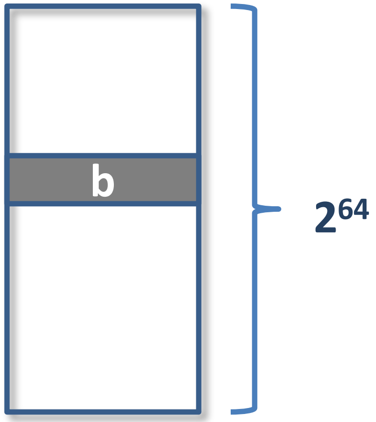
$$P_{fpr} = \left(\frac{b}{2^{64}}\right)^{n} = b \cdot 2^{-64 \cdot n}$$



Accuracy of Multi-Run Verification

- Assume size of mapped memory is ***b*** bytes, run ***n*** times on 64-bits platform, false positive rate for one value is:

$$P_{fpr} = \left(\frac{b}{2^{64}}\right)^{n} = b \cdot 2^{-64 \cdot n}$$



- Say *b* is 22MB (Nginx) and run 2 times. This will result in FPR=2^{-103}

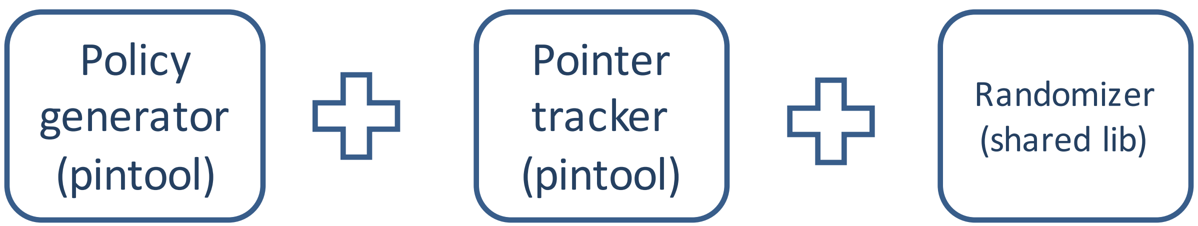
Export Policy

- Given *mov reg1, reg2*
  - if *reg2* is a 64-bits register and tainted (i.e., a pointer) → taint *reg1* after execution

Track All Pointers

Implementation

- Intel’s PIN—a dynamic instrumentation tool
- Three modules



- Source code
  – Coming soon

Evaluation

- Correctness
  - Applied to Nginx web server
  - Memory snapshot analysis to find all pointers
  - RuntimeASLR correctly finds all pointers

Evaluation

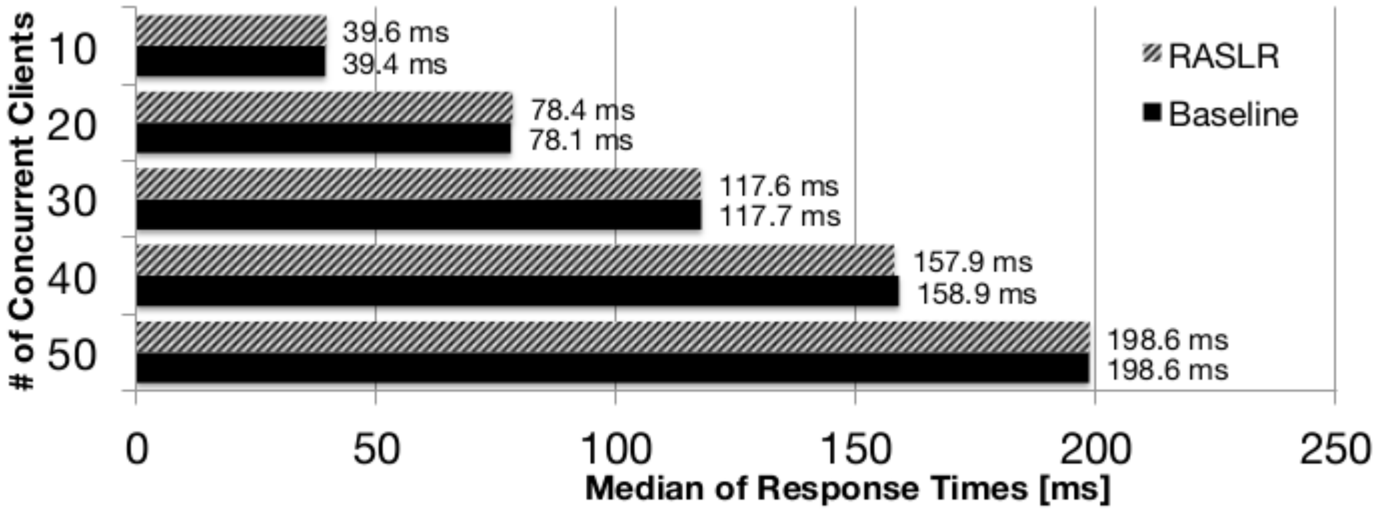
- Security
  - Blind ROP is a clone-probing attack
  - Addresses of all modules are re-randomized
  - RuntimeASLR successfully defeats it

Without RuntimeASLR

With RuntimeASLR

Evaluation

- Performance
  - Pointer tracking is extremely expensive: >10,000 times on SPEC CPU2006
    - One time overhead at startup; 35 seconds for Nginx
  - However, no overhead on cloned worker processes



Discussions and Limitations

- Ambiguous policy
- Completeness of tracking policy
- Applicability for general programs
- Supporting pointer obfuscation

Demo

- Defeat Blind ROP attack with RuntimeASLR

Recap

- Clone-probing attacks → bypass ASLR → control daemon server or steal sensitive data
- We proposed RuntimeASLR to defeat clone-probing attacks
  – Automatic pointer tracking policy generation
  – Support COTS binaries, no system modifications
  – No overhead to cloned worker processes (after fork())

Recap

- Clone-probing attacks → bypass ASLR → control daemon server or steal sensitive data
- We proposed RuntimeASLR to defeat clone-probing attacks
  - Automatic pointer tracking policy generation
  - Support COTS binaries, no system modifications
  - No overhead to cloned worker processes (after fork())

THANKS

Backup slides

Pointer Tracking Approaches

- Compiler-based instrumentation
  - Pros: type info, efficient in tracking
  - Cons: type-confusion, hard to decouple instrumentation, require source
- Dynamic instrumentation
  - Pros: easy to decouple instrumentation, support COTS
  - Cons: lack of type info, tracking is expensive

Pointer Tracking Approaches

- Compiler-based instrumentation
  - Pros: type info, efficient in tracking
  - Cons: type-confusion, hard to decouple instrumentation, require source
- Dynamic instrumentation
  - Pros: easy to decouple instrumentation, support COTS
  - Cons: lack of type info, tracking is expensive

Accuracy of Multi-Run Verification

- Assume ***b*** instructions in ***b*** bytes memory. Probability for at least one non-pointer value misidentified as a pointer is:

$$1 - (1 - P_{fpr})^{b} \Longrightarrow b^{2} \cdot 2^{-64 \cdot n}$$

Accuracy of Multi-Run Verification

- Assume ***b*** instructions in ***b*** bytes memory. Probability for at least one non-pointer value misidentified as a pointer is:

$$1 - (1 - P_{fpr})^{b} \Longrightarrow b^{2} \cdot 2^{-64 \cdot n}$$

- Say *b* is 100MB and run *2* times. This will result in FPR=2^{-76}