Practical Exploit Mitigation Against Code Re-Use Attacks and System Call Abuse

Christopher Jelesnianski

April 13, 2022

Committee:
Changwoo Min (Chair)
Yeongjin Jang, Danfeng Yao, Wenjie Xiong, & Haibo Zeng

VIRGINIA TECH™

Outline

- Motivation
- Background
- Contributions
- Future Work
- Summary

Outline

- **Motivation**
- Background
- Contributions
- Future Work
- Summary

Thesis Statement

Exploit mitigation techniques need to **shift focus** to not only provide an **effective solution** to the latest threat, but also to be a **practical defense** that can be readily **deployed**

Mainstream Adoption of Mitigation Techniques

- PaX Linux Address Space Layout Randomization (**ASLR**) (2001)
- Linux **Kernel Stack ASLR** (2002)

- No-Execute Bit (**NX-bit**)
  - Linux 2.6.8 (2004)
  - Windows XP SP2 (2004) -- Data Execution Prevention (**DEP**)

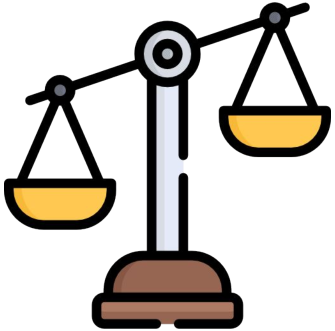
- System Call Whitelisting (**seccomp**)
  - Linux 2.6.12 - (2005)
  - Linux 2.6.23 - (2007)
  - seccomp mode 2 - Linux 3.5 (2012)
  - seccomp eBPF - Linux 3.8 (2013)

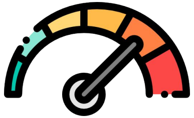
- GCC & Clang **Control-Flow Integrity & SafeStack** (USENIX Security 2014)

That's about it…

Challenges in Exploit Mitigation Design

Why is Practical Exploit Mitigation Design Difficult to Achieve?



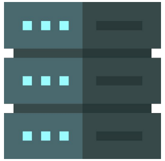


Non-Negligible Performance

- **Complex** or **frequent** verification is hard to make **fast**

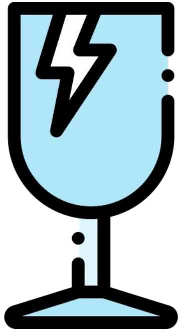
Attack Complexity

- Modern attacks require **more code components** to be **guarded**



Limited Scalability

- Designing mechanisms with **negligible system impact** is hard



Fragility

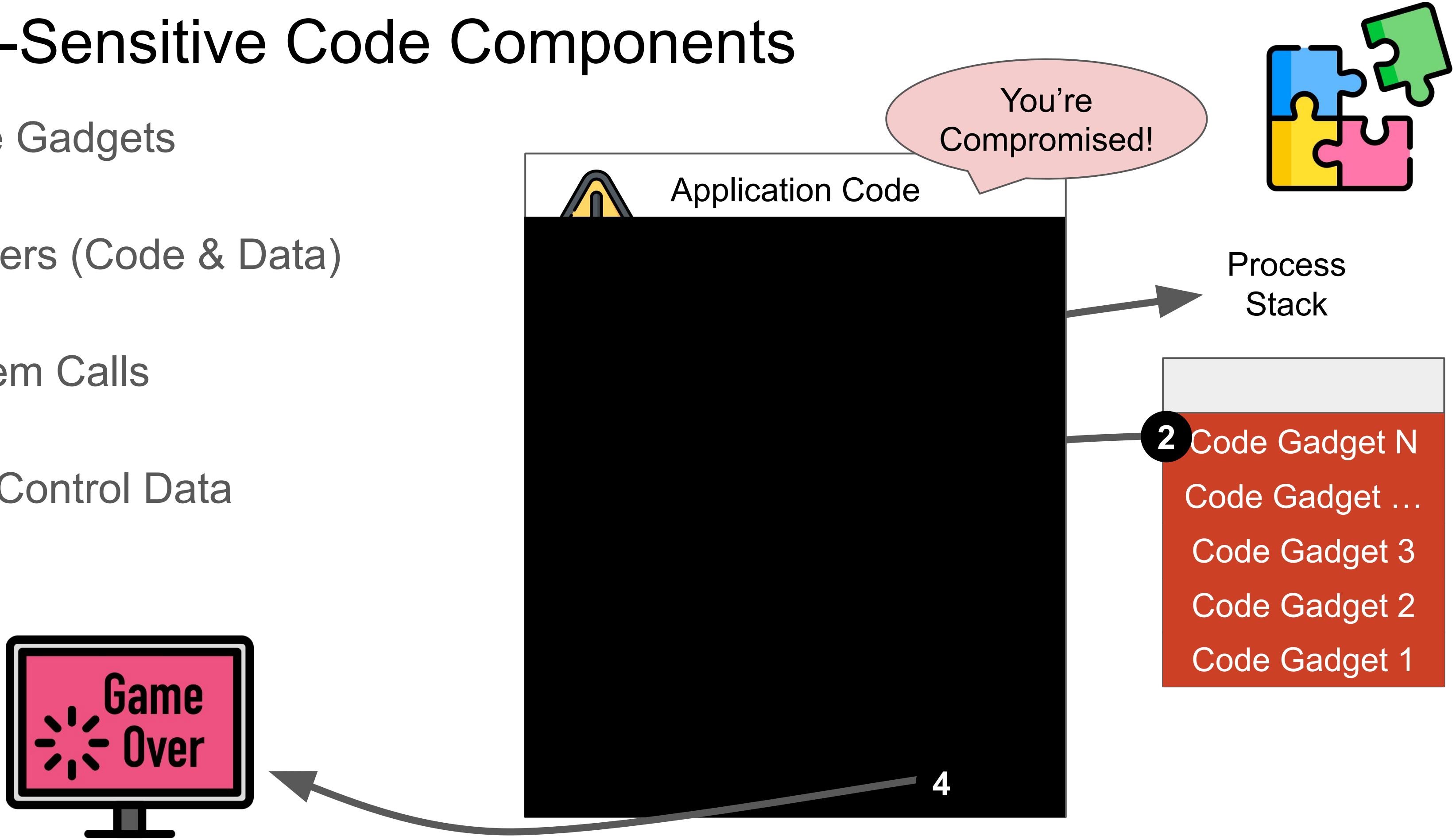
- **Perfect analysis** is challenging to achieve in practice

Outline

- Motivation
- **Background**
  - **Attack-Sensitive Code Components**
  - **How Code Re-Use Attacks Work**
  - **Modern Defense Archetypes**
- Contributions
- Future Work
- Summary

Attack-Sensitive Code Components

- Code Gadgets
- Pointers (Code & Data)
- System Calls
- Non-Control Data



How Code Re-Use Attacks Work

Assumptions / Threat Model

- Memory Vulnerability Exists
  - Buffer Overflow
  - Dangling Pointer

Generalized Code Re-Use Attack Procedure

1 → 2 → 3 → 4 → 5 = 

1. Identify Memory Vulnerability
2. Perform Memory Disclosure
3. Identify Code Components
4. Sensitive Data and/or Control-Flow Hijacking
5. Illegitimate System Call Usage

Code Re-Use Variants

- Return Oriented Programming (ROP) — *Shacham et al. (CCS 2007)*
- Just-In-Time ROP (JIT-ROP) — *Snow et al. (S&P 2013)*
- Blind ROP (BROP) — *Bittau et al. (S&P 2014)*
- Control Data & Non-Control Data Attacks — *van der Veen et al. (CCS 2017)*

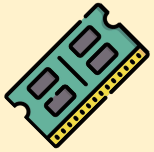
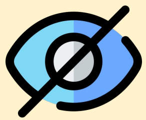
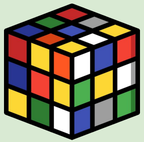
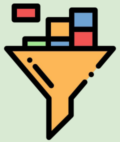
Modern Defense Archetypes

Generalized Code Re-Use Attack Procedure

0 → 1 → 2 → 3 → 4 → 5

0. Presence of Sensitive Code Components
1. Identify Memory Vulnerability
2. Perform Memory Disclosure
3. Identify Viable Code Components
4. Sensitive Data and/or Control-Flow Hijacking
5. Illegitimate System Call Usage

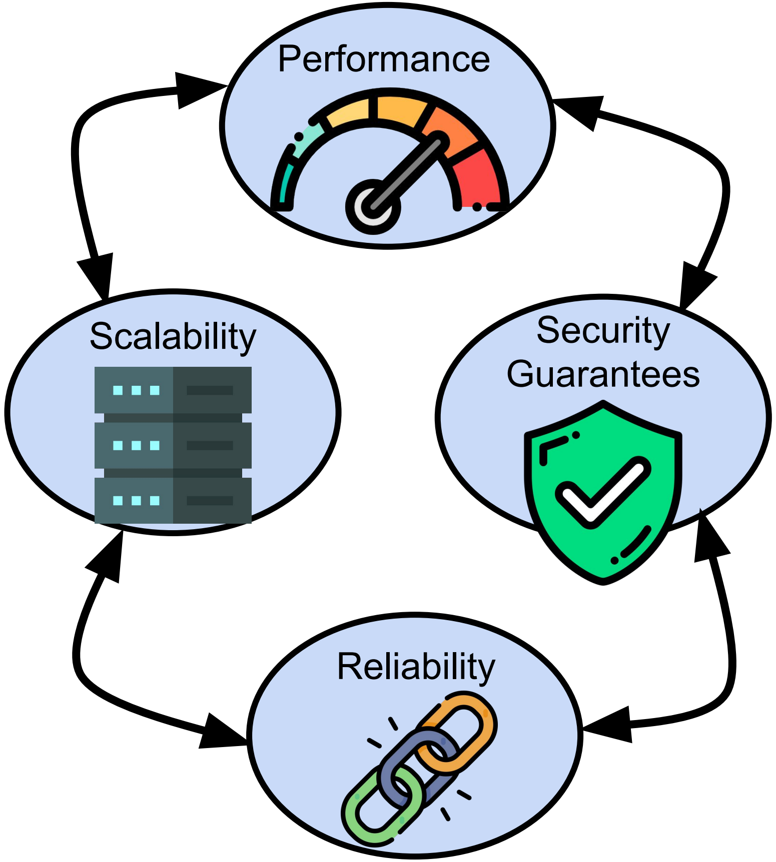
Generalized Defense Archetype Created

Debloating	Memory Safety	Information Hiding	Re-Randomization	Data Integrity & Control Flow Integrity	System Call Filtering
					

Blueprint For Success in Practical Exploit Mitigation Design

<u>Practical Design Properties</u>

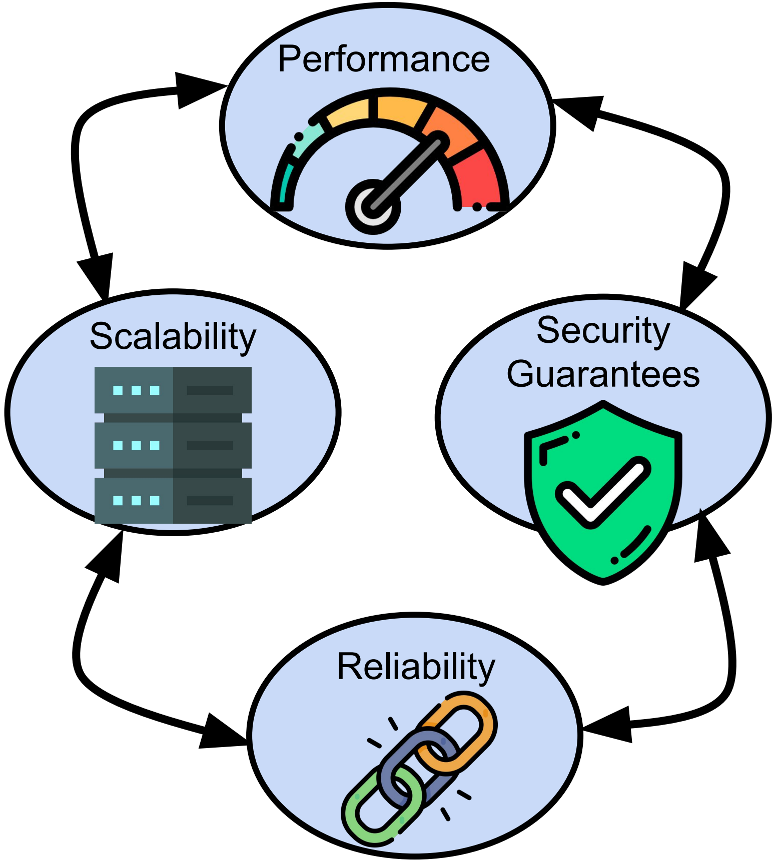
- Low Performance Impact
  - Defenses should achieve **low** performance
- Strong Security Guarantees
  - Defenses should provide **strong** security guarantees
- Scalable Framework
  - Defenses should **minimize** use of additional CPU or memory resources
- Reliable Defense
  - Defenses should **not** break the application runtime



Blueprint For Success in Practical Exploit Mitigation Design

<u>Necessary Considerations</u>

- Being Aware of Trade-offs
- Deriving the Essentials
    - What *<u>code pieces</u>* really needs to be protected?
    - What is the *<u>least</u>* amount of security coverage needed to still be strong and block attacks?

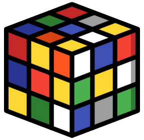


Outline

- Motivation
- Background
- **Contributions**
  - MARDU
  - BASTION
- Future Work
- Summary

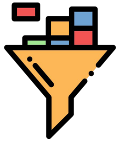
Thesis Contributions

This thesis proposal presents two Practical Exploit Mitigation Designs:



- **MARDU**: Efficient and Scalable Code Re-Randomization
  - Motivation
  - Design
  - Implementation
  - Evaluation

SYSTOR '20
DTRAP '22



- **BASTION**: Context Sensitive System Call Protection
  - Motivation
  - Preliminary Design

To Be Submitted
ASPLOS '23

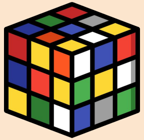
Outline

- Motivation
- Background
- **Contributions**
  - **MARDU**
  - BASTION
- Future Work
- Summary

3

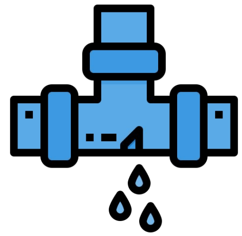
Identify Viable Code Components

Re-Randomization



Current randomization techniques are good ...

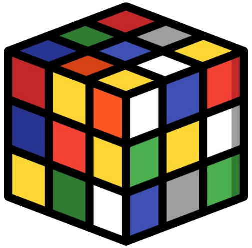
Code Randomization



- Address Space Layout Randomization (ASLR)
  - \+ Light-weight
  - \- Static code layout
  - \- One memory disclosure can compromise entire code layout

Oxymoron
Coarse-Grained ASLR
(Memory Page)

USENIX Security 2014



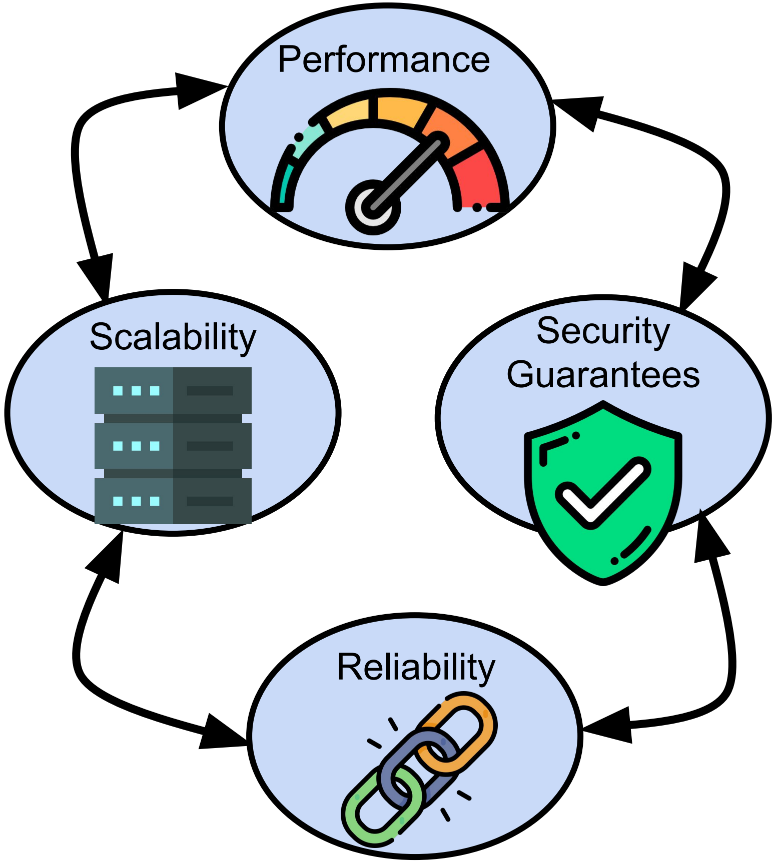
- Re-Randomization Techniques
  - \+ Continuous churn makes gadgets hard to find
  - \- High overhead
  - \- Rely on predictable thresholds such as
    - \- Time interval
    - \- Syscall invocation
    - \- Call history

Shuffler
Fine-Grained Runtime
Re-Randomization
(Function Granularity)

USENIX OSDI 2016

But They Are Not Practical. Why?

- Users desire **acceptable performance** (<10% avg & worst-case)
- Users desire **strong defenses**
- Users desire **scalability** as more computation is moved to the cloud
  - Have system-wide security coverage including shared libraries
- Users desire **reliable** defense that can be generically deployed
- Achieving <u>all</u> *together* is ***hard***



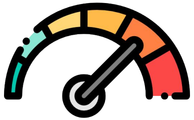
Challenges For Practical Re-randomization

- **Performance** Challenges
  - Avoiding Useless Overwork: Active randomization wastes CPU cycles in case of “what-if”
- **Security** Challenges
  - Code Disclosure: a single leaked pointer allows attacker to obtain code layout of a victim process
- **Scalability** Challenges
  - Code Tracking: to support runtime re-randomization tracking and updating of pc-relative code is a necessary and expensive evil
  - Stop-the-World: Updating shared code on-the-fly is challenging especially in concurrent access
- **Reliability** Challenges
  - Being Generic: Ability for incremental deployment is preferred to completely overhauling a system for isolated protection

MARDU

MARDU Goal Summary

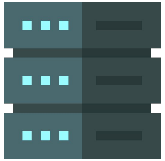
Goal 1:	Performance	Low performance impact, but still high entropy
Goal 2:	Security	Code is secure in runtime
Goal 3:	Scalability	Code-sharing is possible in MARDU
Goal 4:	Reliability	MARDU supports mixed (un)instrumented libraries



MARDU Key Points

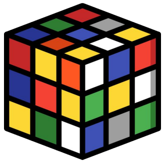
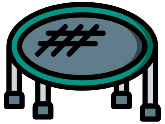
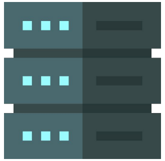
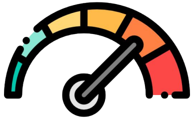
- Performance
  - MARDU performs reactive **on-demand** runtime re-randomization
  - No previous randomization scheme is capable of runtime re-randomization **AND** code sharing

- Security / Scalability / Reliability
  - MARDU uses code **trampolines** with PC-relative addressing
  - MARDU can share it's randomized code in a **system-wide** manner

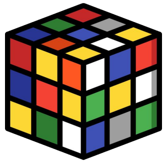


MARDU

- **MARDU Design**
  - MARDU: Design Overview
  - Key Concept: How Code Trampolines Work
  - Scalability: How MARDU Shares Code
  - Performance: Re-randomization Without Stopping the World
- Implementation
- Evaluation

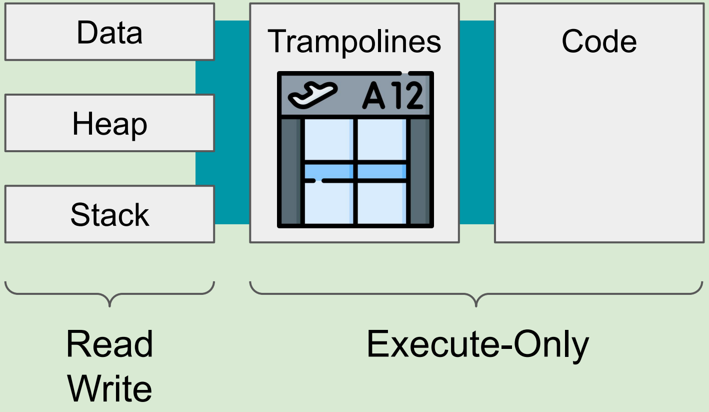
MARDU Design Overview



Compiler Component

- Code Analysis
- Create & Insert MARDU Trampolines
- Generate Fixup Information for Patching

MARDU Process

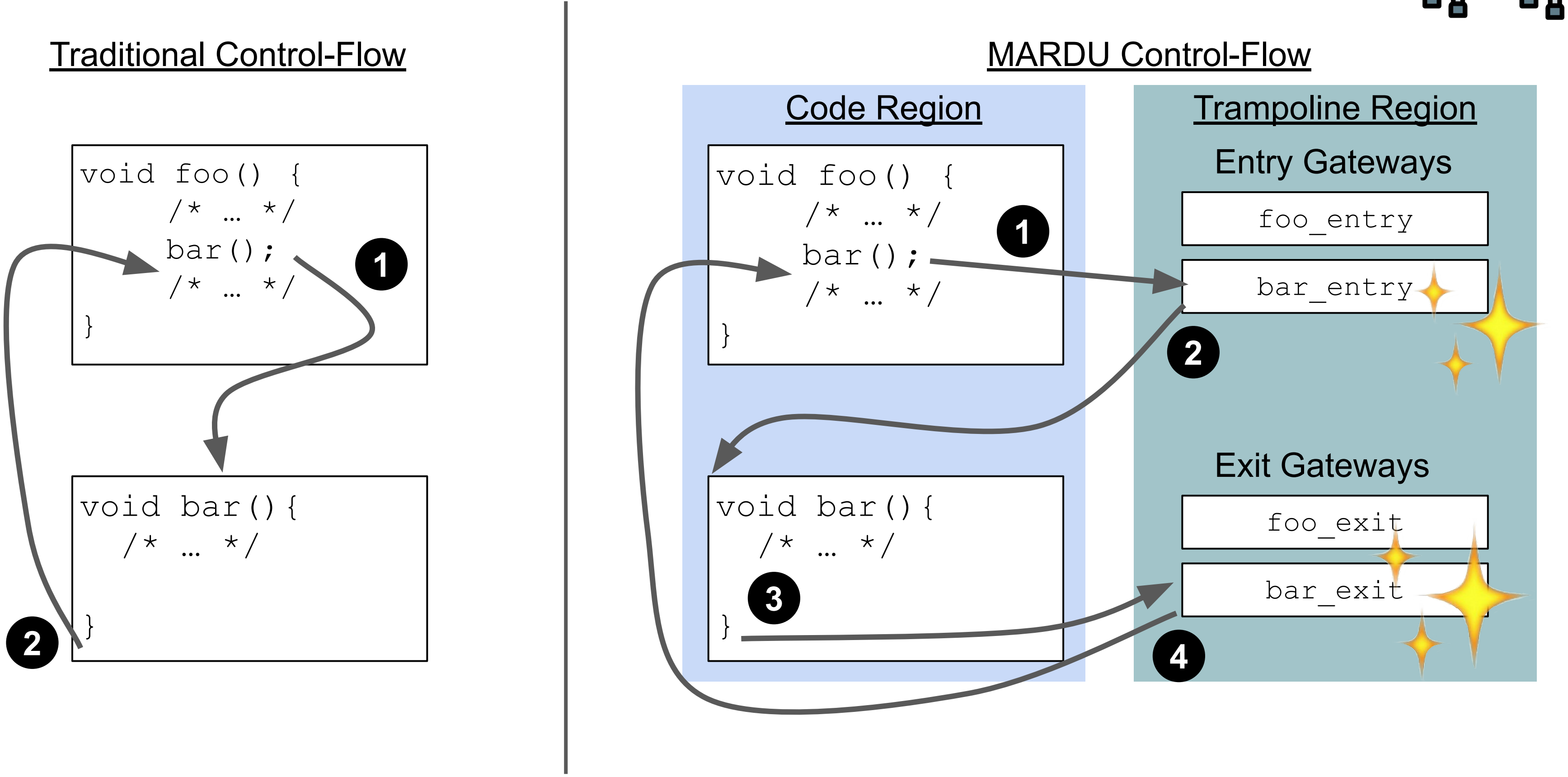


Trampolines act as:
entry gateway to each function call
exit gateway for each function return

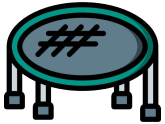
Kernel Component

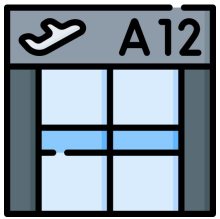
- Memory layout & Execute-only Overlay
- On-Demand Re-Randomization & Patching

How Code Trampolines Work



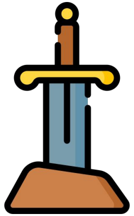
How Code Trampolines Work





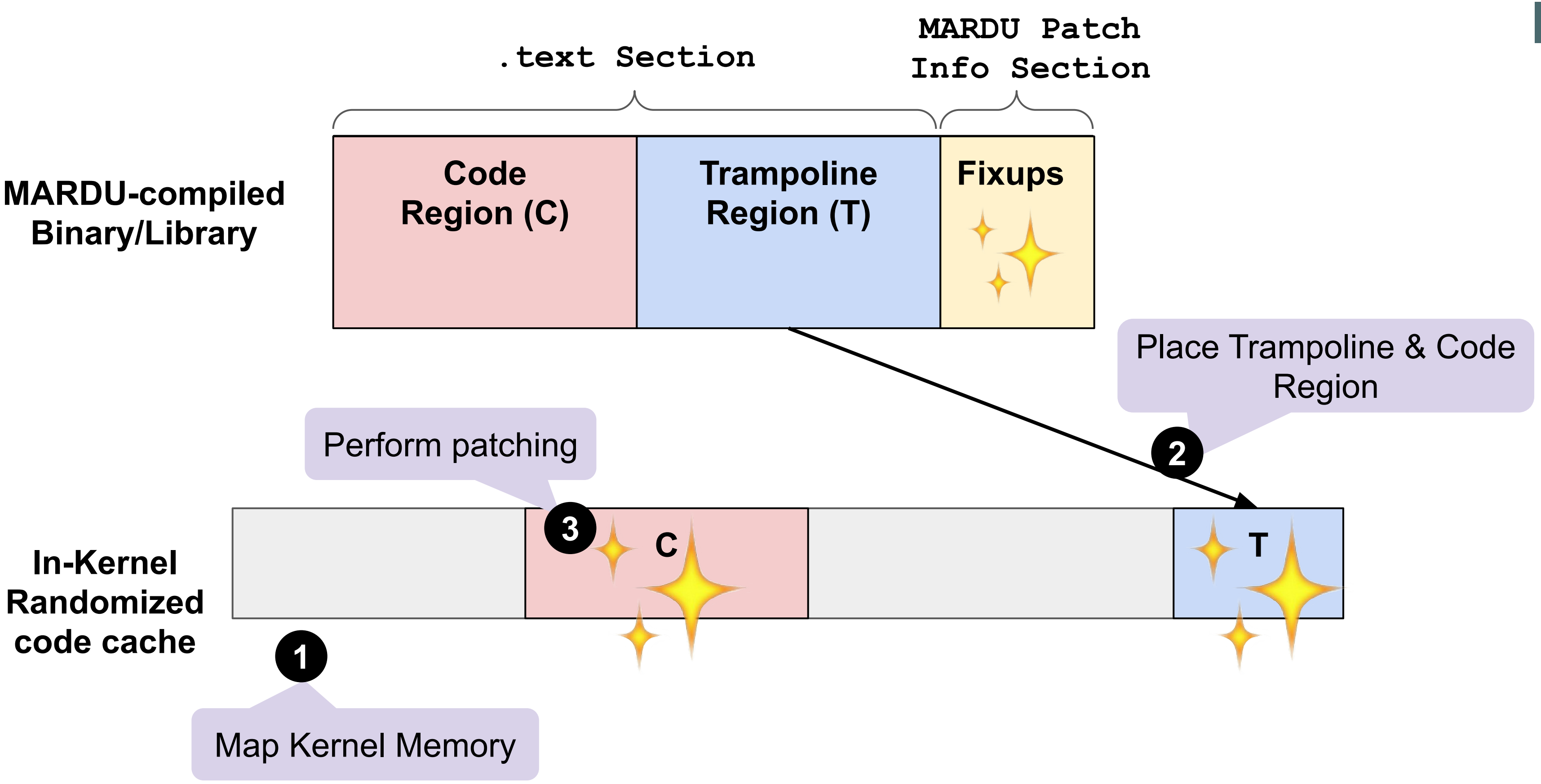
- MARDU runtime always transfers using the trampoline region

- Trampolines leverage immutable code
  - MARDU has a ground truth that will always maintain program semantics

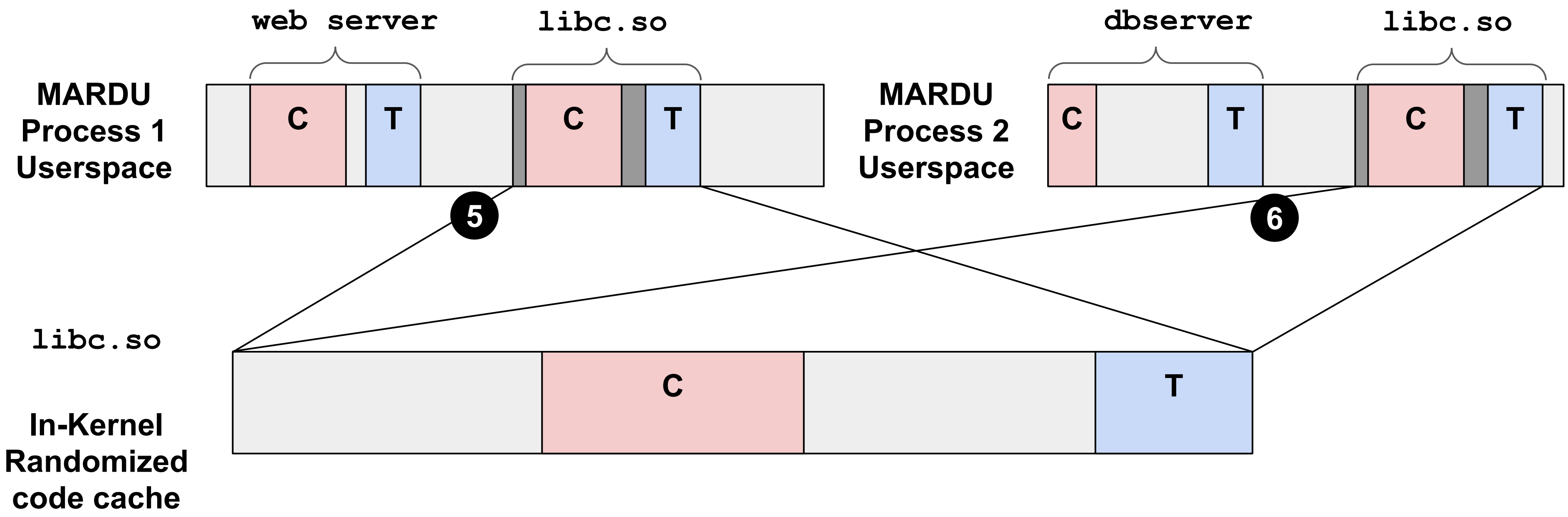


- Simplified Runtime Tracking & Re-Randomization

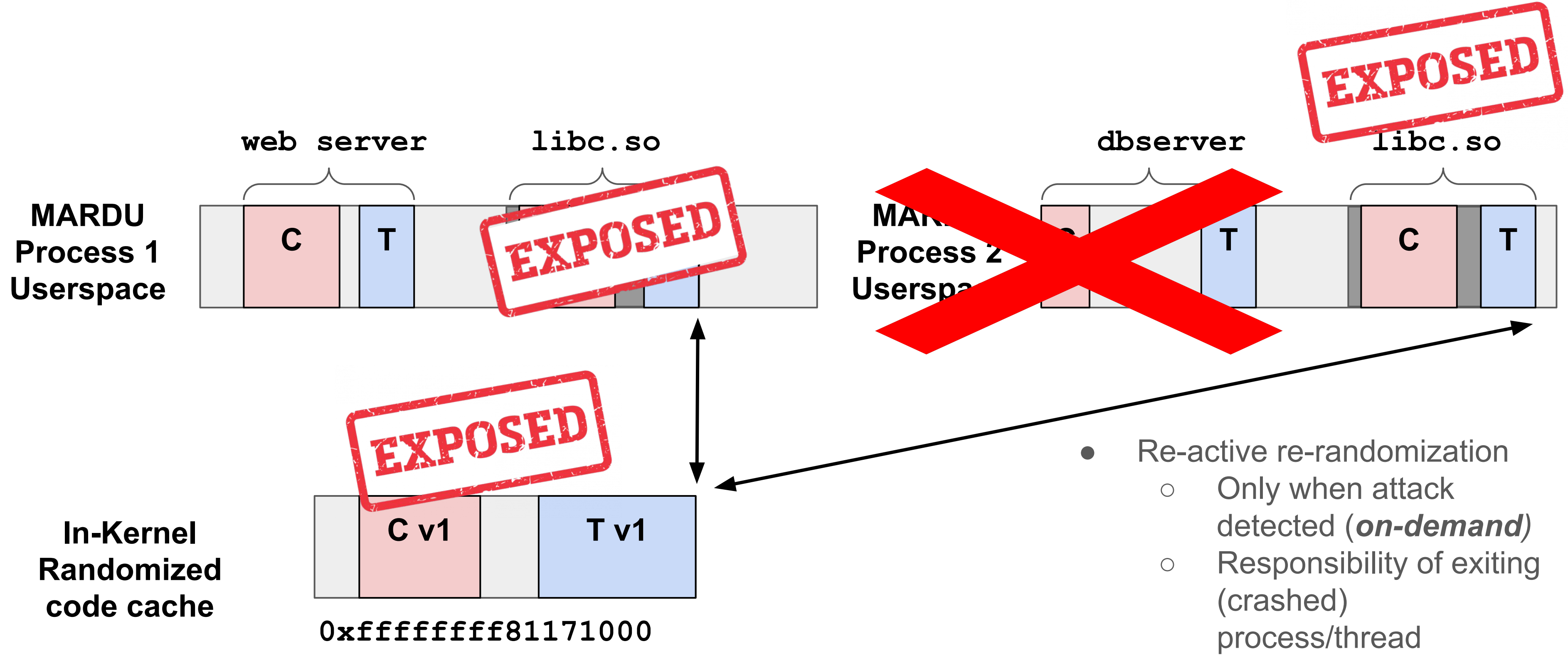
Example: How MARDU Shares Code



Example: How MARDU Shares Code



Re-Randomization Without Stopping the World

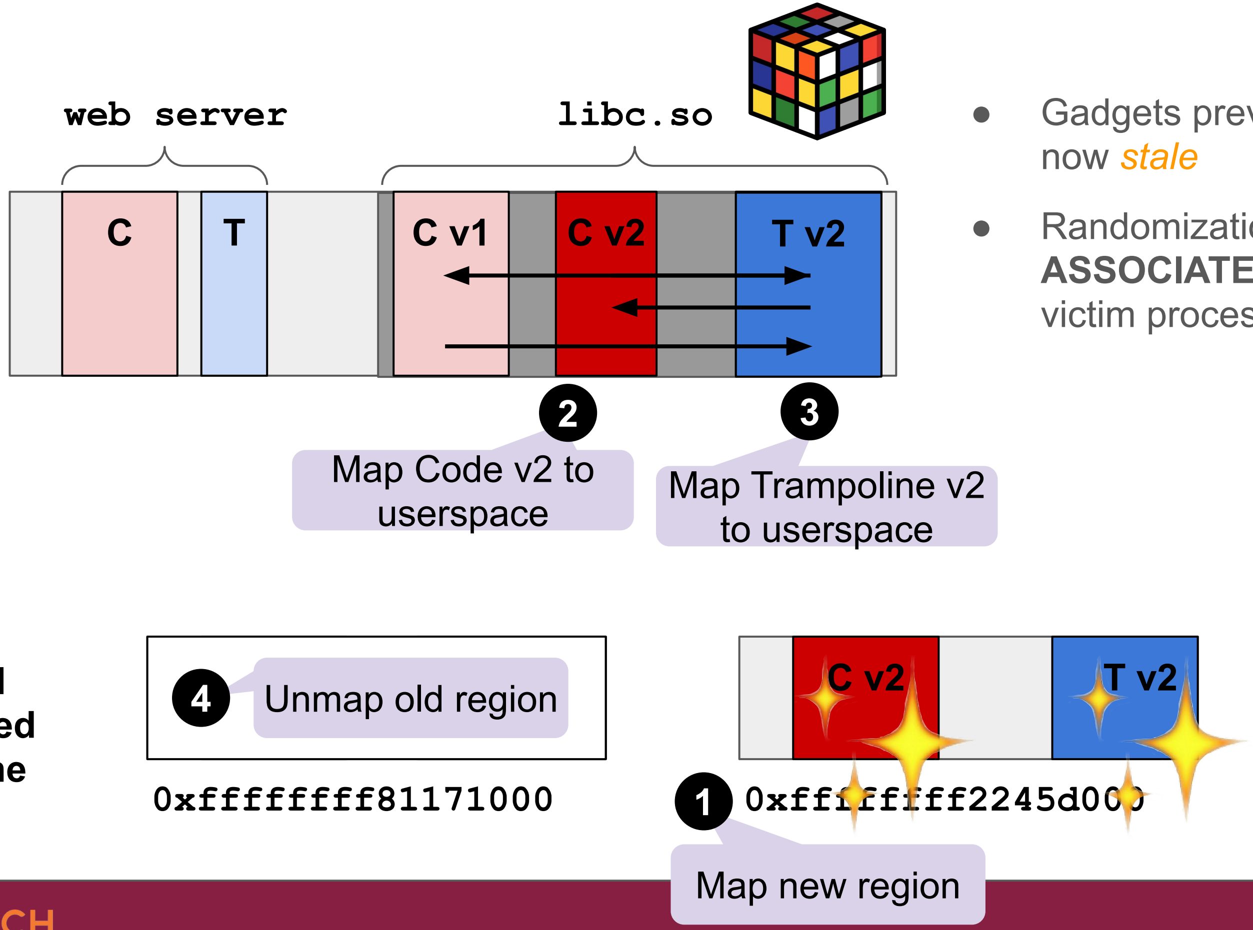


- Re-active re-randomization
  - Only when attack detected (***on-demand***)
  - Responsibility of exiting (crashed) process/thread

Re-Randomization Without Stopping the World

MARDU Process 1 Userspace

In-Kernel Randomized code cache



- Gadgets previously deduced are now *stale*
- Randomization is replicated for **ALL ASSOCIATED** shared code of a victim process

MARDU Implementation

- Working Framework

- Compiler
  - LLVM/Clang 6.0.0
  - 3.5K LOC

- Kernel
  - X86-64 linux 4.17.0
  - 4K LOC

- `musl LibC`
  - General C library

MARDU Evaluation

- **Evaluation**
  - Security: Popular ROP attacks -> MARDU Wins
  - Performance: Compute Bound -> Minimal Runtime Overhead
  - Scalability: Concurrent WebServer-> Negligible Runtime Overhead and Scalability

1) How much performance overhead does MARDU impose?

2) How scalable is MARDU in terms of memory savings?

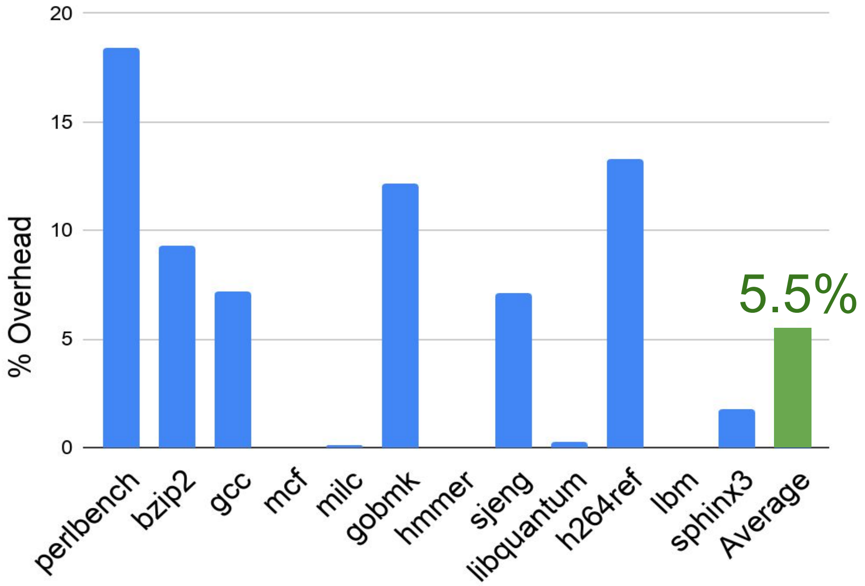
Experimental Setup and Applications

- Experimental Setup
  - All programs compiled with MARDU LLVM compiler and `-O2 -fPIC` optimization flags
  - Platform:
    - 24-core (48-Hardware thread) machine with two Intel Xeon Silver 4116 CPUs (2.10 GHz)
    - 128 GB DRAM

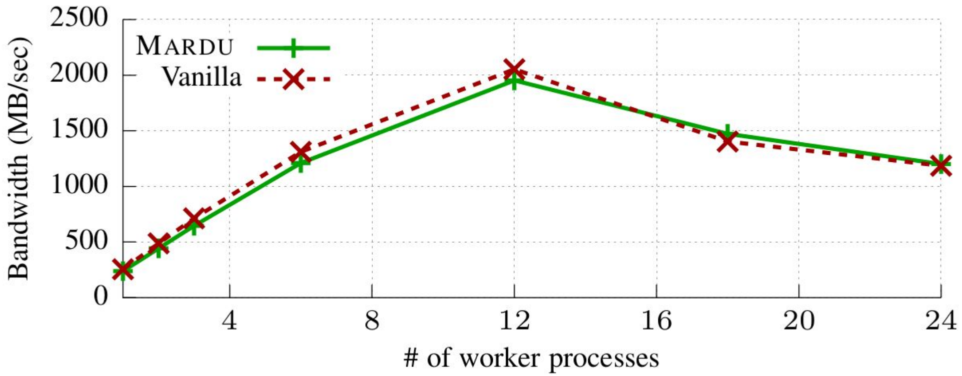
- Applications
  - SPEC CPU 2006 (All C applications)
  - NGINX web server

How MARDU Performs

CPU Intensive Benchmark
SPEC CPU 2006



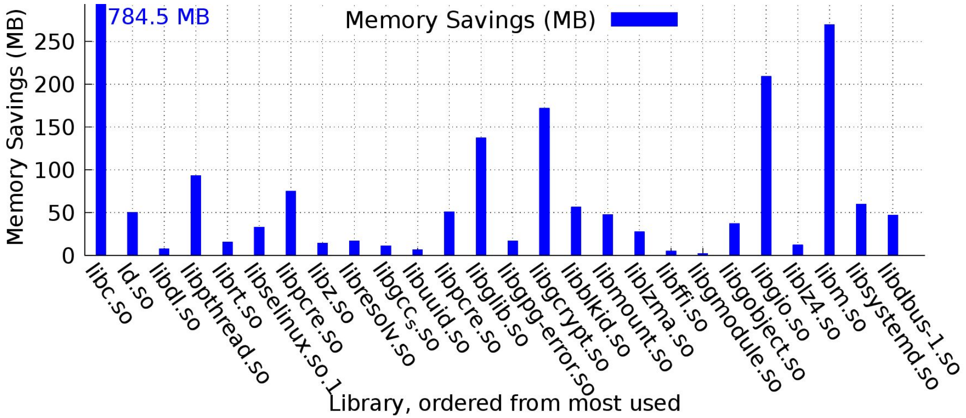
Concurrent Web Server
NGINX



NGINX Average Degradation: 4.4%

How MARDU Scales

- One-time upfront cost to instrument
- **Native** shared library total size: **787 MB**
- **Traditional non-sharing** would incur **8.8 GB** overhead
  - Library replicated & allocated new memory per process
- MARDU incurs **1.3 GB** memory usage (over **7.5 GB** memory **savings**)
- Biggest memory savings come from
  - **libc.so** saves **0.78 GB** memory
  - **libm.so**, saves **0.26 GB** memory



MARDU Summary & Limitations

Summary

- MARDU is Practical
- 5.5% average performance overhead
- Security blocks all ROP attack variants
- First randomization scheme capable of <u>runtime re-randomization</u> ***with*** <u>code sharing</u>
- Does not Stop-the-World for runtime re-randomization

Limitations

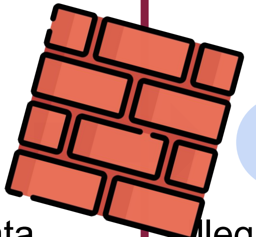
- Re-Randomization is a **probabilistic** defense
- Performance could be improved, worst case performance is 18.3%
- **Non-Control Data Attacks** are not covered by MARDU

Modern Security

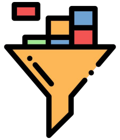
Generalized Code Re-Use Attack Procedure

1 → 2 → 3 → 4 → 5 = 

1. Identify Memory Vulnerability
2. Perform Memory Disclosure
3. Identify Viable Code Components
4. Sensitive Data and/or Control-Flow Hijacking
5. Illegitimate System Call Usage

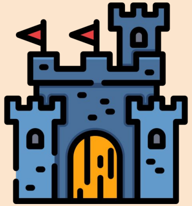


System Call Filtering



Outline

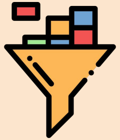
- Motivation
- Background
- **Contributions**
  - MARDU
  - **BASTION**
- Future Work
- Summary



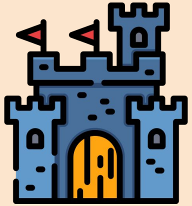
5

Illegitimate System Call Usage

System Call Filtering



Outline

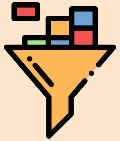


- Motivation
- Background
- **Contributions**
  - MARDU
  - **BASTION**
    - **Insights & Motivation**
    - **Design**
    - **Preliminary Evaluation**
- Future Work
- Summary

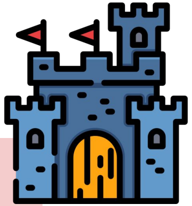
5

Illegitimate
System Call
Usage

System Call
Filtering

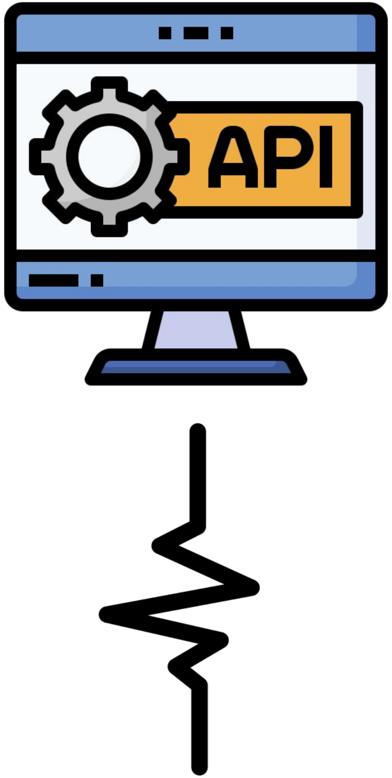


Insights of System Call Usage in Code Re-Use



For **Applications** (Legitimate Usage)

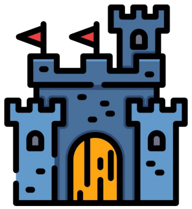
- Provide an API interface between applications and host OS kernel
- Provide **numerous services**
- **359** system calls currently implemented (Linux v5.17.1)
- Many system calls are **non-sensitive**
- System Calls are **scarcely** used in practice



For **Attacks** (Malicious Usage)

- Provide an API interface to further gain a foothold on victim host
- Some system calls are **security-critical** especially:
  - execve
  - mmap
  - mprotect

Example 1: Duality of System Call Usage



NGINX Web Server

Legitimate Uses

- **`execve()`** used to update server in place during runtime

Attacker Uses

- **`execve()`** can launch an attacker binary by reaching system call and corrupting `ctx->path, ctx->argv`

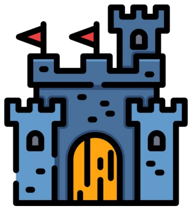
```
// nginx/src/os/unix/ngx_process.c
static void ngx_execute_proc(ngx_cycle_t *cycle,void *data){

        ngx_exec_ctx_t *ctx = data;

        if( execve( ctx->path, ctx->argv, ctx->envp ) == -1){

                ngx_log_error(NGX_LOG_ALERT, cycle->log,
                        ngx_errno, “execve() failed”);
        }
        exit(1);
}
```

Summary of System Call Usage



Code Re-Use Attacks

- Attacks greatly vary in approach, complexity, and end goals
- Attacks desire to **reach** and **use system calls** in needed way

Consequences:

- System calls could be considered a **critical lynchpin** in attack completion
- Protecting system calls can block many system-call based attacks
- **Strong contexts** are needed to adequately protect system calls

Bottom Line:

- System calls deserve more attention!

Current Practices in System Call Filtering

What is Current System Call Filtering Doing?

- Analysis to derive Allow/Block list of system calls
- Static Argument Constraints

`seccomp`: System Call Whitelisting[1]

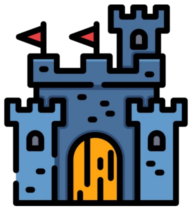
- Defacto **manual** system call policy mechanism implemented in Linux
- Needs **new separate policy** for every application, & every configuration

`sysfilter`: Automated Policy Generation[2]

- Introduction of automation from analysis framework to generate appropriate `seccomp` filter for a target application
- reduces burden on administrator, not stronger security

1 The kernel development community. Seccomp BPF (SECure COMPuting with filters), 2015
2 Nicholas DeMarinis et al. sysfilter: Automated system call filtering for commodity software. (*RAID 2020*)

BASTION: Our Solution



- *Surround* system calls with **three** tight, **specialized** contexts

BASTION System Call Contexts

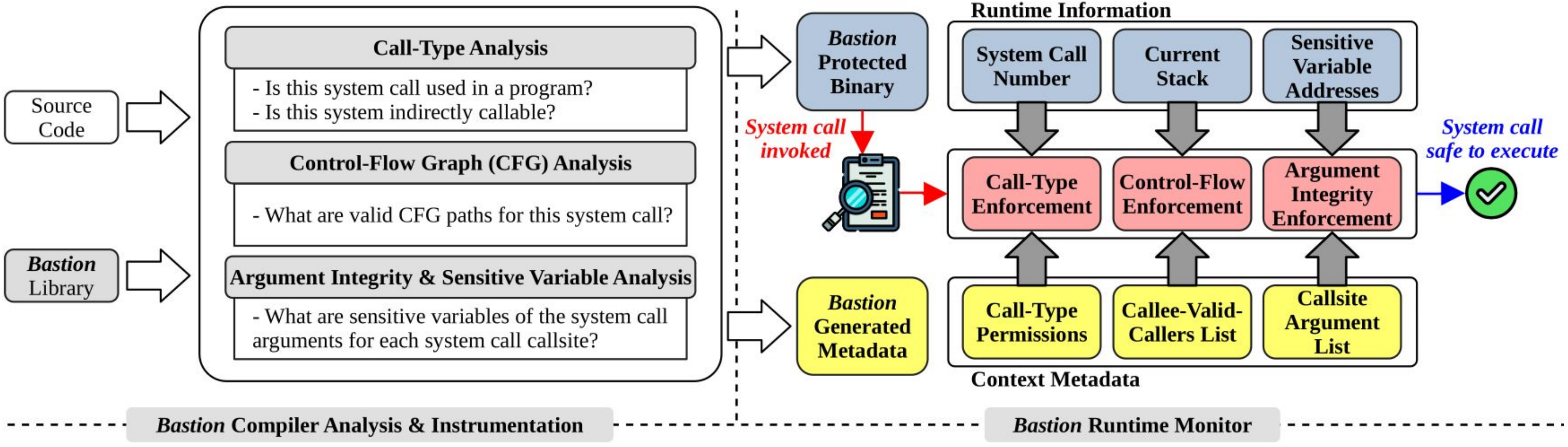
- **Call-Type** Context
  - Is this system call used in the program?
  - How is this system call invoked by the program? (directly, indirectly, or both)

- **Control-Flow** Context
  - What are the valid Control Flow Graph (CFG) paths that reach this system call callsite?

- **Argument Integrity** Context
  - What are the sensitive variables used as system call arguments and dependent variables for each system call callsite?

BASTION: Our Solution

BASTION Framework Overview



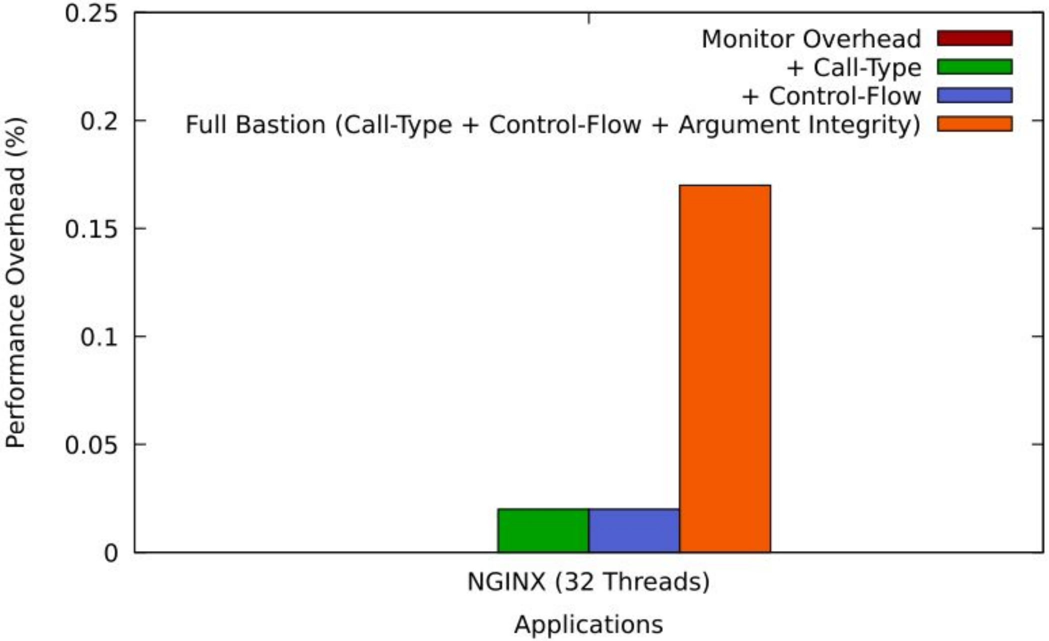
BASTION: Preliminary Evaluation

<u>Evaluation</u>

Application: NGINX Web Server

<u>Results</u>

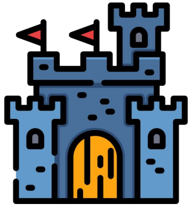
- Negligible Overhead
- BASTION Argument Integrity is ***significantly cheaper*** than DFI by being ***specialized***



Outline

- Motivation
- Background
- Contributions
- **Future Work**
- Summary

BASTION Future Work



Current Status:

- Bare-Bones Prototype of BASTION in place

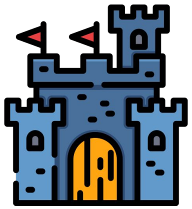
Ongoing Research

1. Complete BASTION System Call Defense Framework
2. Comprehensive **Security** Study of BASTION
3. Comprehensive **Performance** Study of BASTION

Questions To Answer:

- Can BASTION adequately protect against prominent Code Re-Use Attack Vectors?
- Is BASTION just as performant for a diverse set of real-world applications as shown in our preliminary evaluation?

BASTION Future Work



Proposed Security Study:

- Real-world CVEs
- Address Oblivious Code Re-Use (AOCR) *NDSS ’17*
- NEWTON: Dynamic Gadget Discovery Framework *CCS ’17*

Proposed Performance Study:

- NGINX Web Server
- SQLite SQL Database Engine

Other real-world application candidates

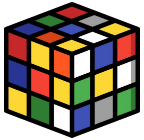
- vsftpd FTP server

Outline

- Motivation
- Background
- Contributions
- Future Work
- **Summary**

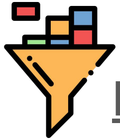
Summary of Research

- Defense techniques to fight *code re-use*
- Exploit Mitigation Mechanisms that are ***practical***



MARDU Key Contributions

- Enabled capability of **code sharing** for randomized code
- Efficient, **system-wide** (re-)randomization that is **on-demand**



BASTION Key Contributions

- Stop code re-use attack chain at system call usage step
- **Three new contexts** to protect system calls
- **Narrow** design only enforcing integrity on system call relevant components

Summary of Research

Publications:

BASTION

BASTION: Protect the System Call, Protect (most of) the World

- To be completed and submitted *ASPLOS '23*

MARDU

MARDU: Efficient and Scalable Code Re-randomization
Christopher Jelesnianski, Jinwoo Yom, Changwoo Min, and Yeongjin Jang (***SYSTOR'20***)

Securely Sharing Randomized Code that Flies
Christopher Jelesnianski, Jinwoo Yom, Changwoo Min, and Yeongjin Jang (***DTRAP'22***)

Summary of Research

Other Publications:

Security

Tightly Seal Your Sensitive Pointers with PACTight
Mohannad Ismail, Andrew Quach, **Christopher Jelesnianski**, Yeongjin Jang and Changwoo Min **(*To appear USENIX Security'22*)**

- Data protection utilizing ARM Pointer Authentication (PA) security primitive

VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks
Mohannad Ismail, Jinwoo Yom, **Christopher Jelesnianski**, Yeongjin Jang and Changwoo Min **(*CCS'21*)**

- Value Integrity for security-relevant data types

Compilers & System Software

Breaking the Boundaries in Heterogeneous-ISA Datacenters
Antonio Barbalace, Robert Lyerly, **Christopher Jelesnianski**, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran **(*ASPLOS'17*)**

Operating system process and thread migration in heterogeneous platforms
Robert Lyerly, Antonio Barbalace, **Christopher Jelesnianski**, Vincent Legout, Anthony Carno, and Binoy Ravindran **(*MaRS'16*)**

Popcorn: Bridging the programmability gap in heterogeneous-isa platforms
Antonio Barbalace, Marina Sadini, Saif Ansary, **Christopher Jelesnianski**, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran **(*Eurosys'15*)**

Practical Exploit Mitigation Against Code Re-Use Attacks and System Call Abuse

Christopher Jelesnianski

Thank you!

Questions?

VIRGINIA TECH™