Resilient Mission Computer (RMC)

Dr. Hamed Okhravi

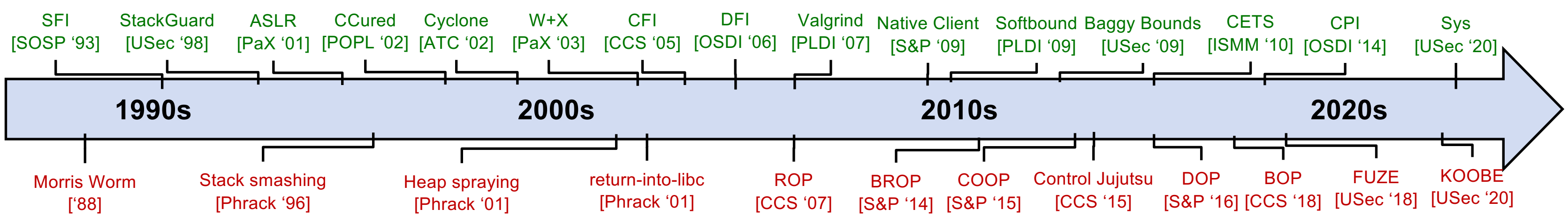
21 November 2022

LINCOLN LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

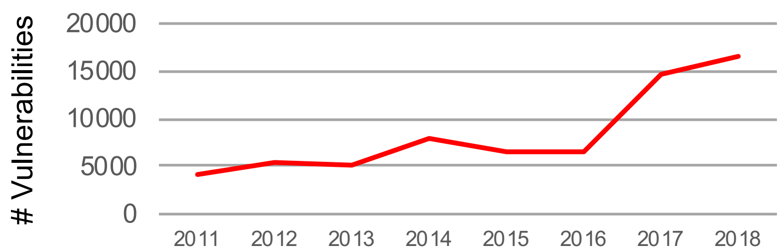
The Arms Race in Computer Security

- **The community has introduced decades worth of defenses:**



- **…but also, decades of attack advancements**

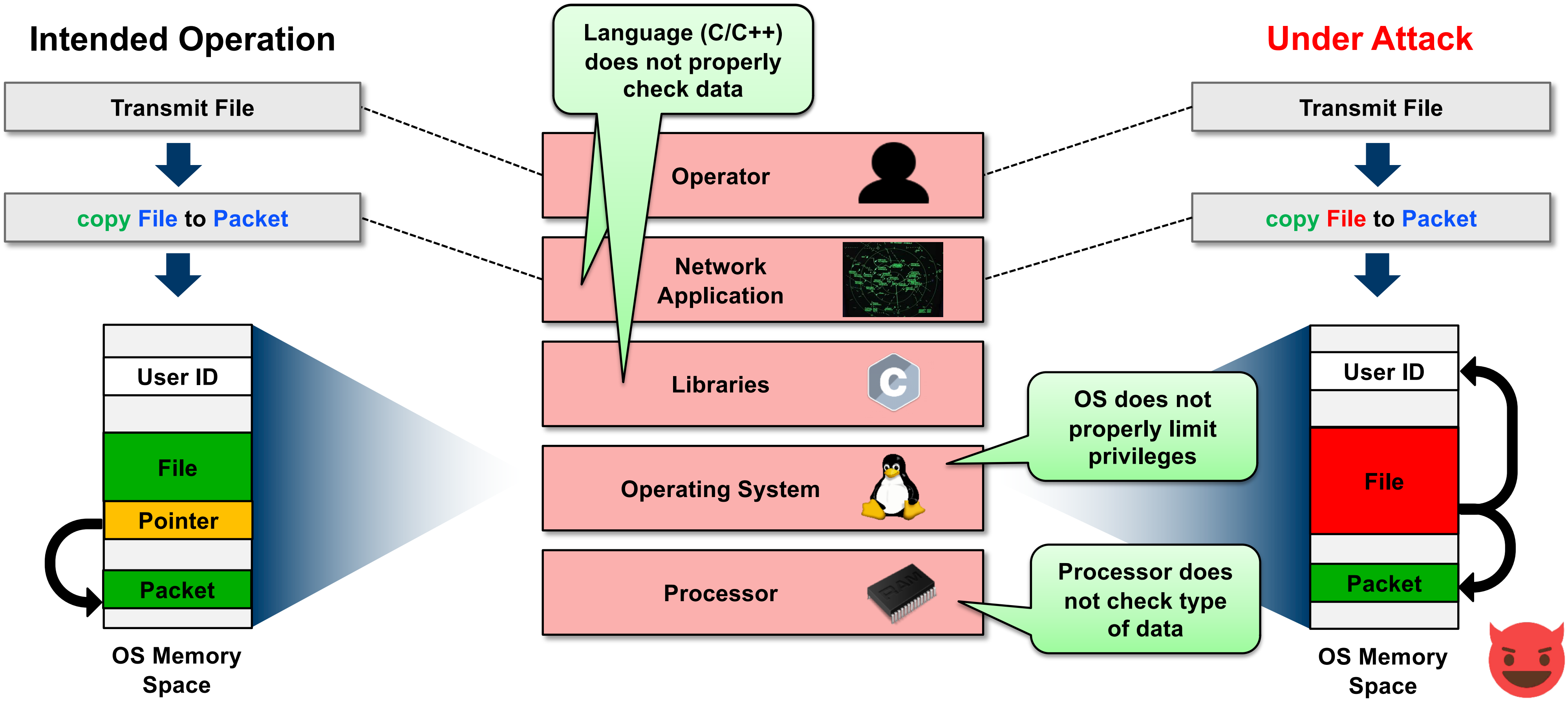
<u>Software Vulnerabilities Increasing in Number and Severity</u>



This tension has lead to little change in observed CVEs

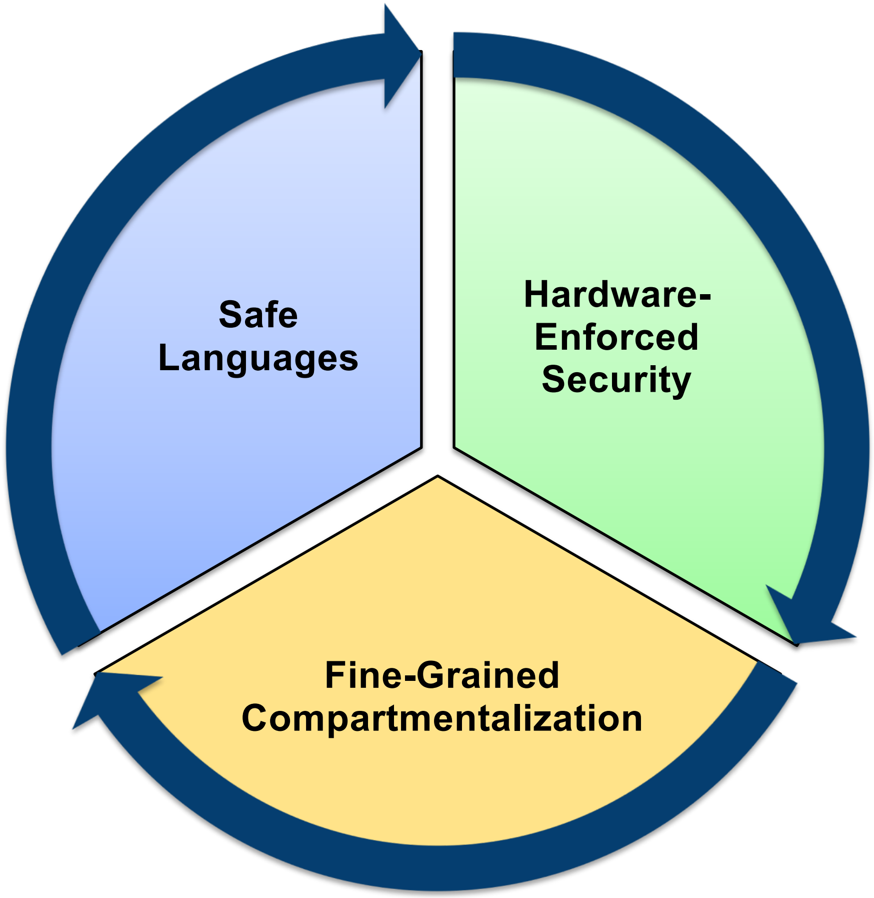
Despite the defenses, classic vulnerabilities still affect modern computer systems

Anatomy of a Cyber Attack



Inspired by CVE-2016-4997 – known privilege escalation vulnerability in Linux networking

Design Principles



Resilient Mission Computer (RMC)

resilient mission computer

Moonshot Vision: Create a secure-by-design system in which the mission can succeed regardless of attempted attacks

Resilient Mission Computer (RMC) Architecture

- **Secure Applications** — App written in secure language (Rust)
  - High-level app semantics preserved
- **Safe Libraries** — Secure lang. stops many classes of vul.
  - Secure interfaces between lib and OS
- **Compartmentalized Operating System** (HAKC[1]) — Compartmentalized OS
  - Semantics extended to HW interface
- **Security-Aware Processor** — Tagged processor checks semantics
  - Formal sanitization of raw input


Current PoC[2] Boards


Quadcopter Test Platform


Program Team

Timeline

- 2016: Seedling Idea Developed
- 2017: Initial Architecture Developed
- 2018: Baseline PoC (Rust + Tock OS) Successfully Tested
- Secure App-OS Interface Developed
- 2019: Secure OS-HW Interface Developed
- Quadcopter Drone Tested
- 2020–2021: Compartmentalized OS
- PoC Fully Integrated and Tested on ARM
- 2022: A feature-rich, secure operating system (HAKC)

RMC featured on the cover of prestigious IEEE Security & Privacy May/June 2021

RMC by numbers: 5 Inventions, 2 Open Source Software, 15 Papers (9 top-tier), 9 Masters theses, 2 Demos, 15+ Talks, 4 Awards
Transitioning to DARPA, DOT&E, NAVAIR, NAWCAD, industry partners, and other external sponsors

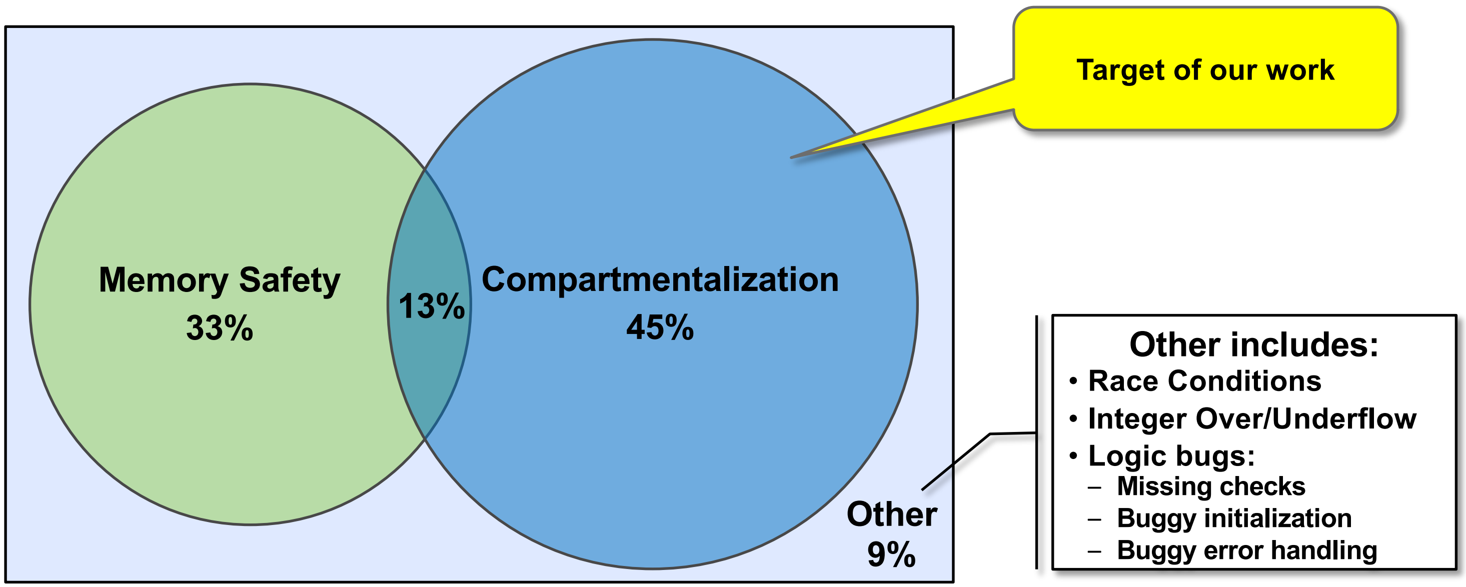
1: HAKC: Hardware-Assisted Kernel Compartmentalization, a Zero Trust version of the Linux OS
2: PoC: Proof-of-Concept

Outline

- Compartmentalizing the OS
  - Why compartmentalization
  - Implementation on commodity processors
  - Evaluation
- Securely using safe languages
  - How safe languages work
  - Cross-language attacks
  - Defending against cross-language attacks
- Conclusion

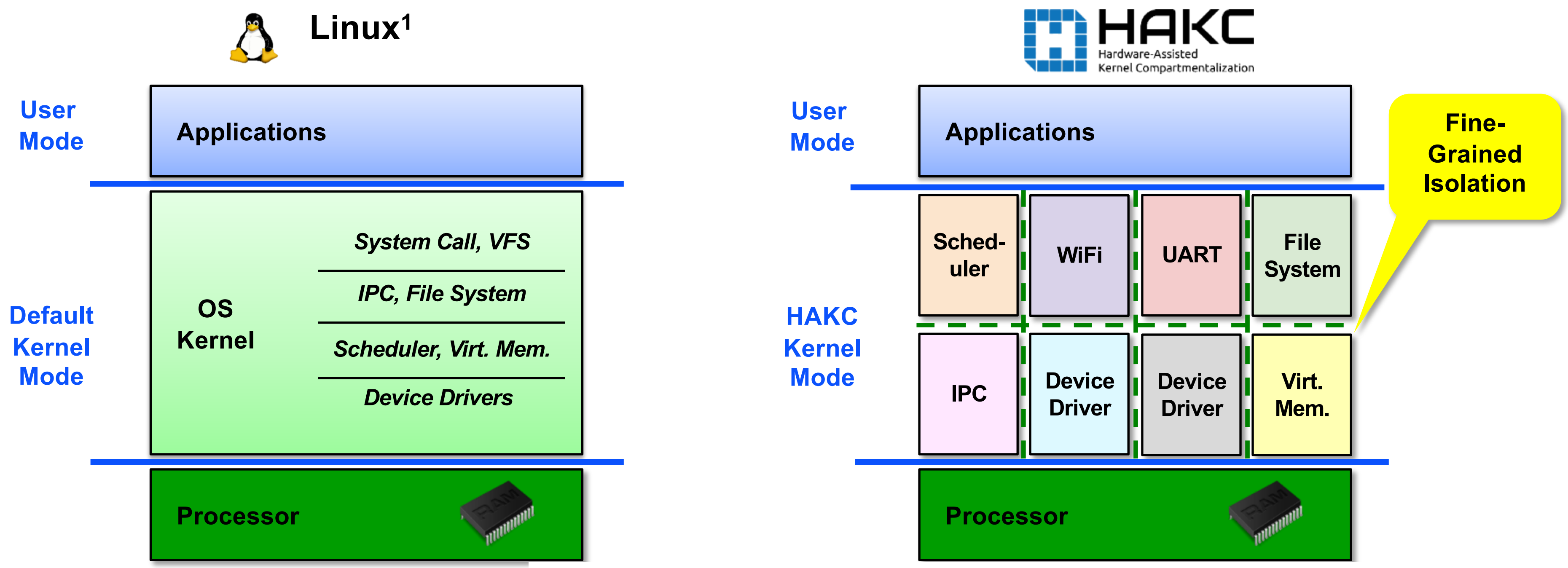
How to Secure an OS?

We analyzed the past 5 years of vulnerabilities in Linux: 508 with *critical* or *high* severity



We enforce *compartmentalization* to prevent the most common class of bugs

Compartmentalized Operating System

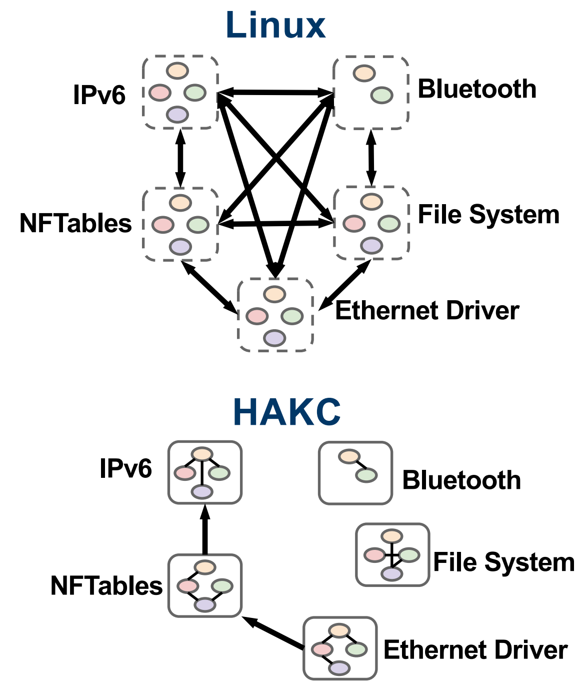


	Linux	HAKC
Security	*Weak*	*Strong*
Existing App Support	*Strong*	*Strong*

1. Also: Windows, MacOS, VXWorks, etc.

Hardware-Assisted Kernel Compartmentalization (HAKC)[1]

- **Invented compartmentalization enforcement mechanism using limited tag bits**
  - **Uses ARM PAC and MTE**
  - **Heavy weight compartment boundaries, lighter weight cliques**
- **Compartmented the IPv6 and NFTables kernel modules**
  - **Security evaluation using emulation (QEMU)**
  - **Performance evaluation using surrogate instructions on Raspberry Pi**
    - **Current overhead 2 – 24%**
- **HAKC is fully compatible with existing applications/servers that run on Linux**

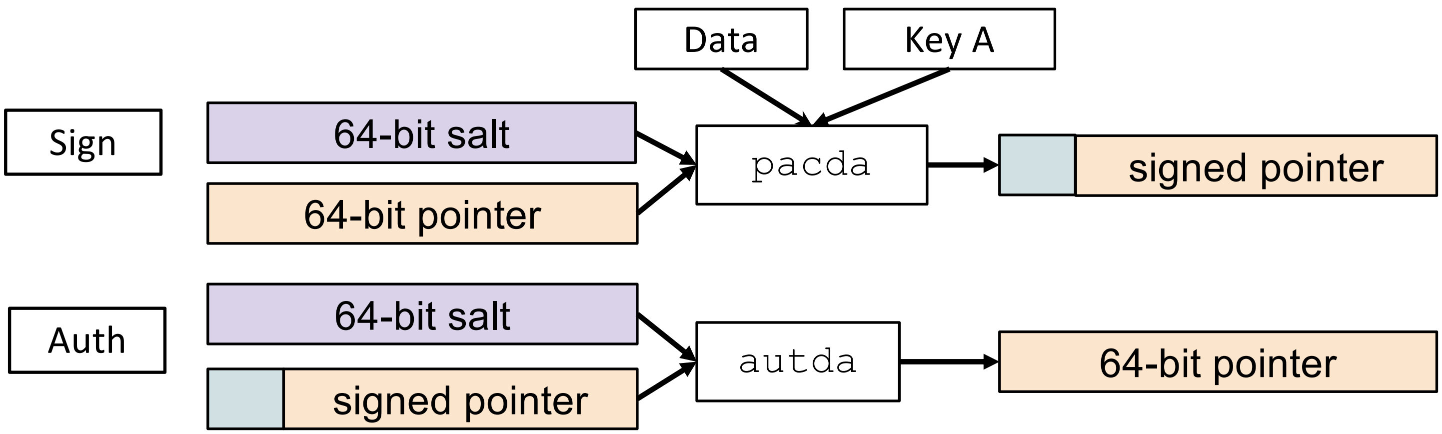


HAKC uses ARM security extensions to secure Linux via compartmentalization

1. Derrick McKee, Yianni Giannaris, Carolina Ortega, Howard Shrobe, Mathias Payer, Hamed Okhravi, Nathan Burow. "Preventing Kernel Hacks with HAKCs". NDSS 2022. **Best Paper**

ARM Security Primitives -- PAC

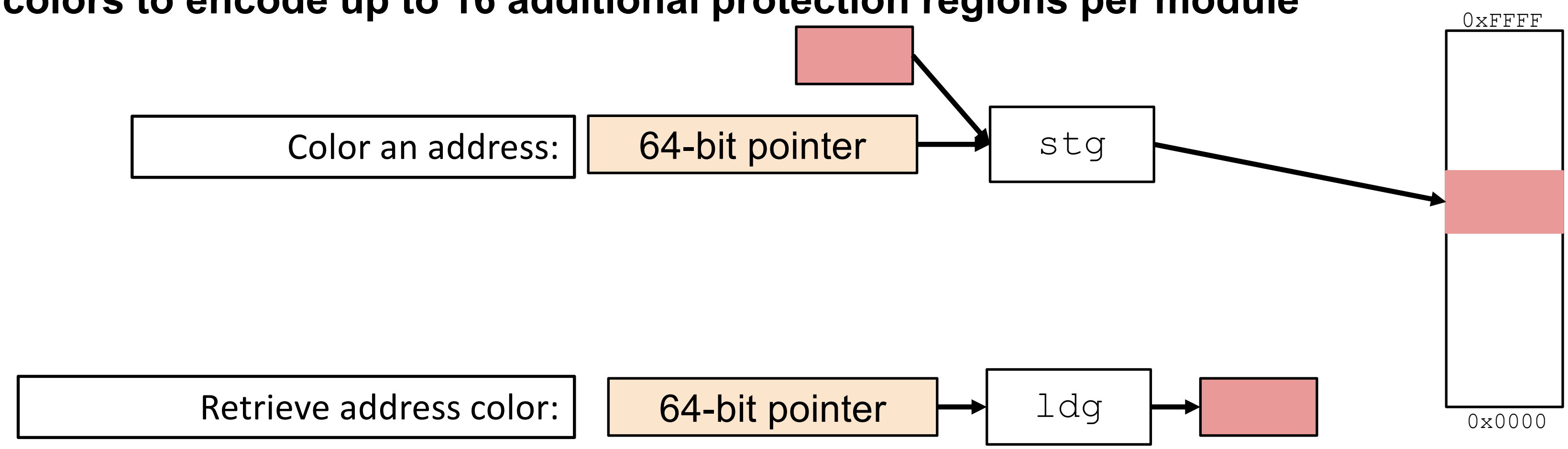
- **Pointer Authentication Code (PAC)**
- **Can sign a pointer with a 64 bit salt value**
- **Salt is used to encode kernel module, e.g., IPv6**



PAC can compartmentalize an arbitrary number of kernel modules

ARM Security Primitives -- MTE

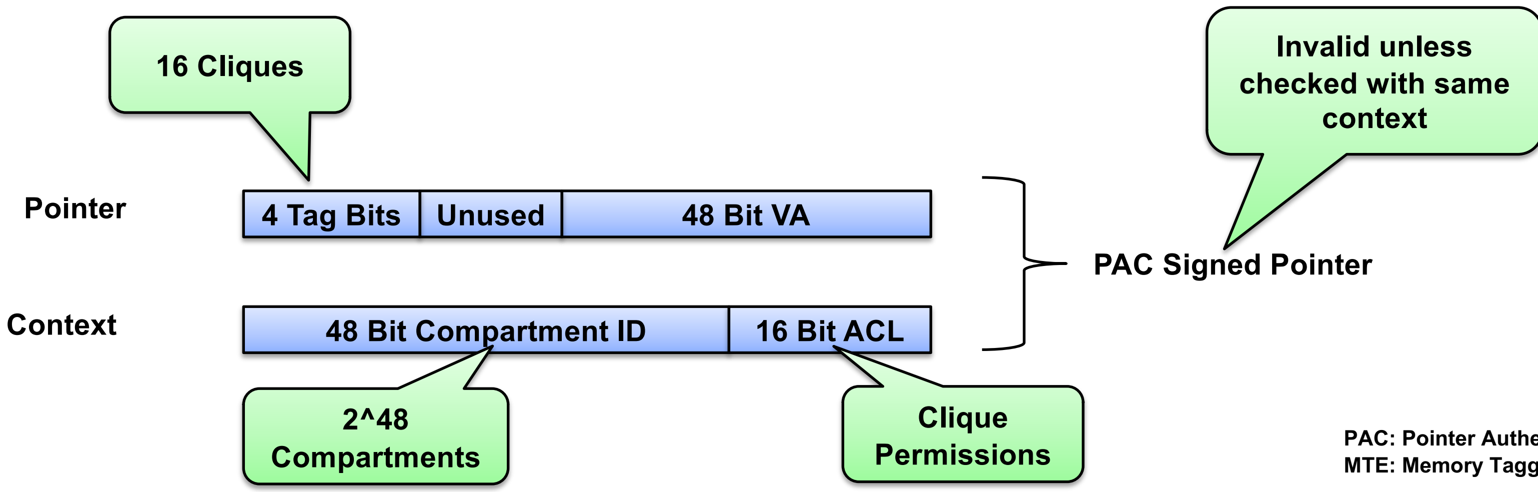
- **Memory Tagging Extension (MTE)**
- **Can add a 4-bit "color" to memory and pointers**
- **Use colors to encode up to 16 additional protection regions per module**



MTE allows finer-grained compartmentalization within kernel modules

Unlimited Compartments

- Have 4 tag bits from MTE – standard number from our literature survey
- Have the ability to sign a pointer with 64 bits of context from PAC
  - 48 bits used for compartment ID
  - 16 bits used for other clique metadata
- Achieve 2^48 compartments, each with 2^4 cliques within them



PAC: Pointer Authentication Code
MTE: Memory Tagging Extension

Applying Compartments

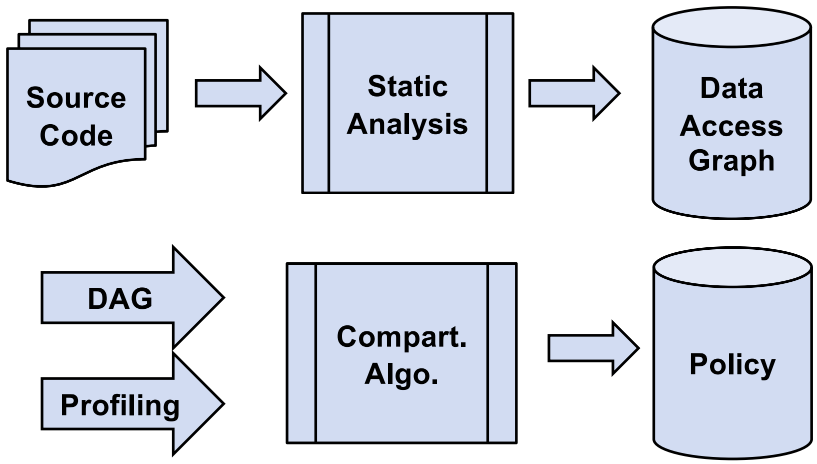
```
#include <linux/hakc.h>

//Declare Compartment
HAKC_MODULE_CLAQUE(…);
//Declare Allowed Transitions
HAKC_EXIT(…);

int foo(int *x, int y){
    //Compiler added check
    *(HAKC_CHECK_DATA_ACCESS(x)) = y;
}
```

- **GUI that allows developers to specify what compartments at the granularity of functions (or entire files)**
- **Compiler automatically adds checks to pointer dereferences**
- **Checks validate:**
  - **Pointer and data are owned by the same compartment**
  - **Pointer and data are in the same clique (or there is a valid connection)**

Minimal developer intervention required *once* to set compartmentalization policy

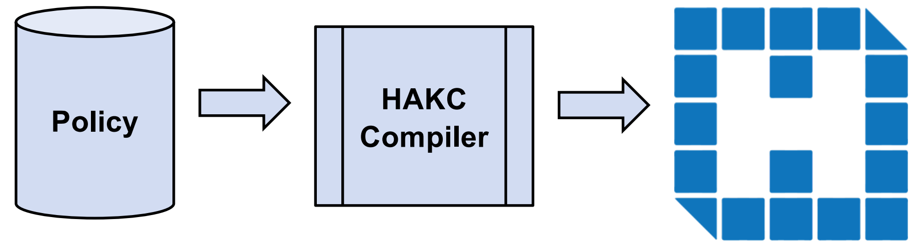
Automating Compartmentalization

Discovering Policies



- Possible compartmentalization algorithms:
  - Minimum Spanning Tree
  - Weighted Knapsack

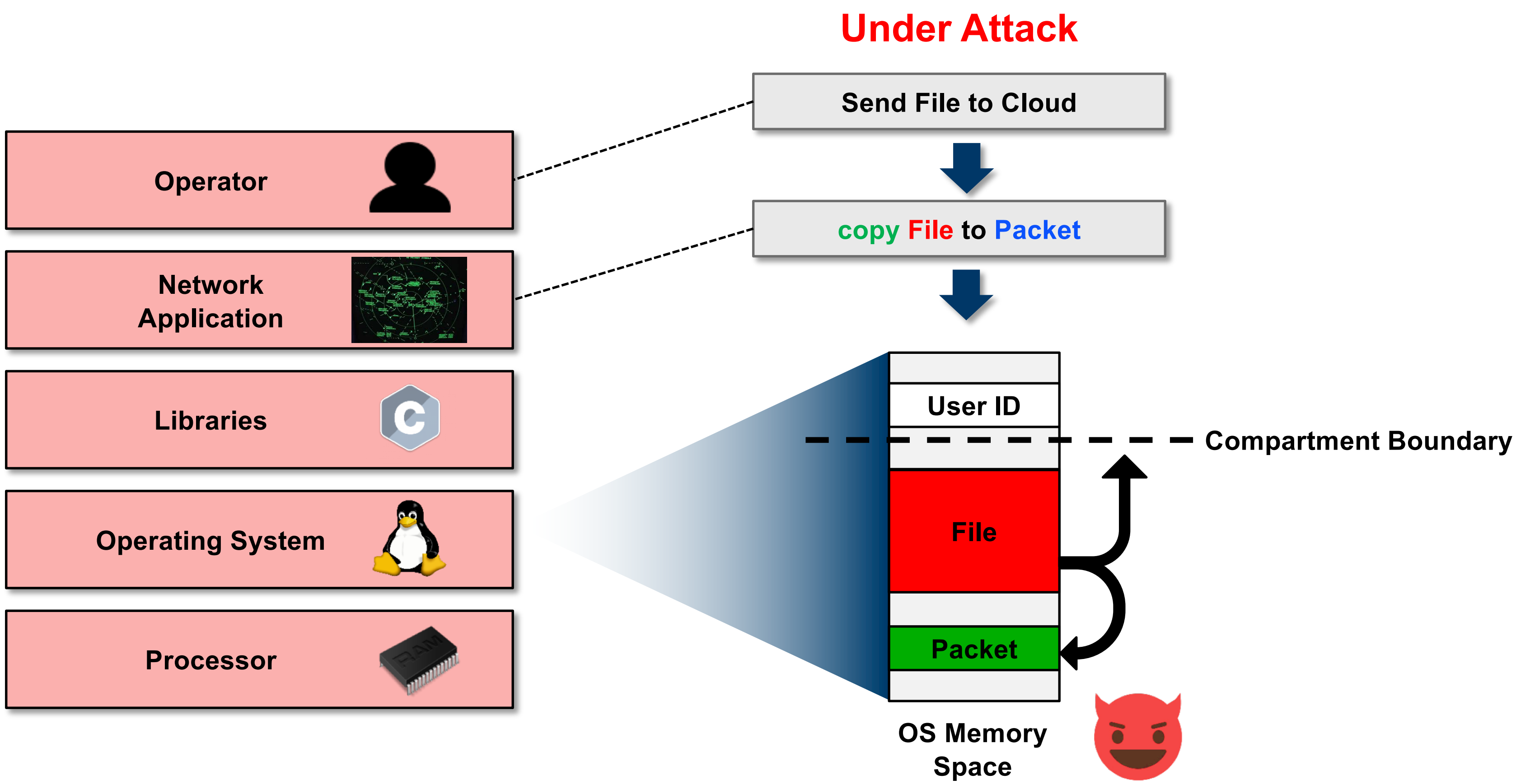
Applying Policies



- No code annotations to specify policy
- Automating data transfer between compartments
- Enable rapid experimentation with compartmentalization algorithms

Creating a framework to systematically evaluate performance vs security trade-offs

Attack Mitigated by Compartmentalization



Inspired by CVE-2016-4997

Evaluation: Web Server Case Study

- Current results, optimization ongoing
- Relies on conservative (additional overhead) substitute instruction sequences
- Run our kernel on Raspberry Pi, IPv6 LAN connection to a laptop
- Run Apache on Pi, measure time to serve 3 different file sizes

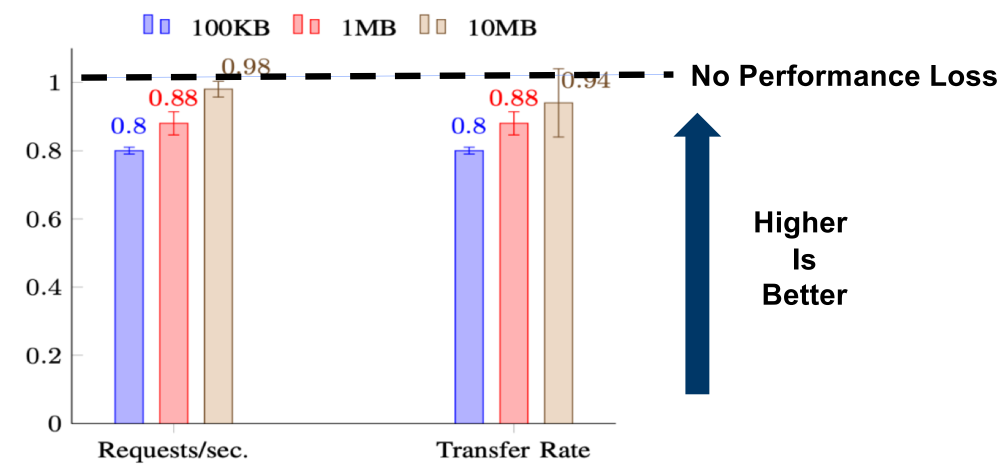


Fig. 7: ipv6.ko overhead normalized to unmodified kernel when transferring various sized payloads.

Performance tolerable under maximum load for server applications

Evaluation: Web Browsing Case Study

- Visited the Alexa Top 50 Websites to see impact of HAKC on load times
- Table shows the measured time differences between HAKC and the baseline kernel, averaged over 5 samples taken at different times of day

Websites with lowest standard deviation

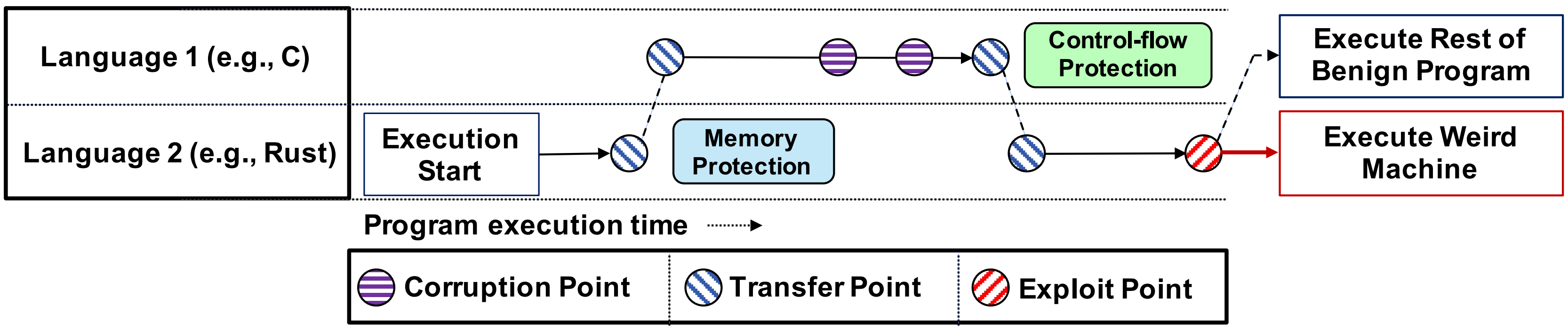
Websites with highest standard deviation

Website	Delta (s)	Stdev (s)
linkedin.com	-0.47	0.065
hdfcbank.com	-0.12	0.085
google.cn	-0.068	0.086
bing.com	-0.087	0.13
investing.com	38	62
okezone.com	-11	20
cnn.com	-9.8	15
yahoo.com	-4.9	15

Negative numbers
→
Slower load time with HAKC

Performance impact within standard deviation for most websites

How to Securely Use Safe Languages



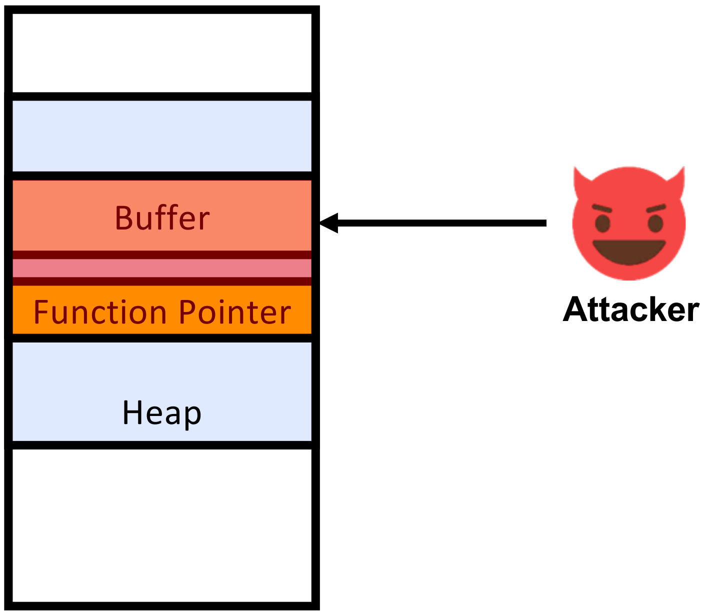
- Adding Rust to hardened legacy applications may ***decrease*** security!
- Attackers can leverage novel ***cross-language attacks***
- Incrementally deploying Rust safely requires accurate threat models

Need novel security policies for mixed-language applications

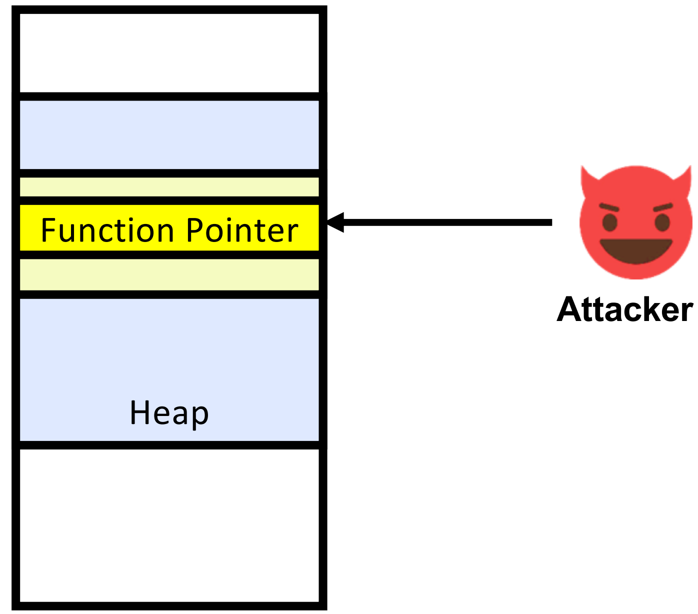
Outline

- Compartmentalizing the OS
  - Why compartmentalization
  - Implementation on commodity processors
  - Evaluation
- Securely using safe languages
  - How safe languages work
  - Cross-language attacks
  - Defending against cross-language attacks
- Conclusion

Recall: Memory Corruption Attacks



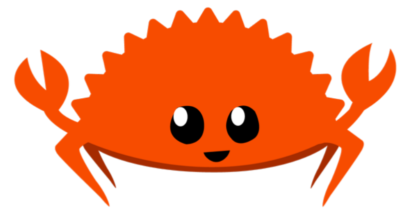
Spatial Memory Violation



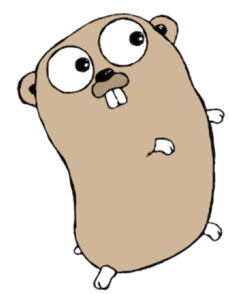
Temporal Memory Violation

Rise of Safe, System Programming Languages

- Can we prevent memory problems at the onset?
  - Without insane performance costs
- Rust
  - Compile-time checks
    - Strong type system → Prevents arbitrary casting
    - Bounds checks on static data
    - Ownership and Lifetimes
  - Run-time checks
    - Bounds checks on dynamic data
- Go
  - Compile-time checks
  -
    - Garbage collection: Leads to slightly larger run-time (but still performant!)



Acceptable run-time even for the systems domain

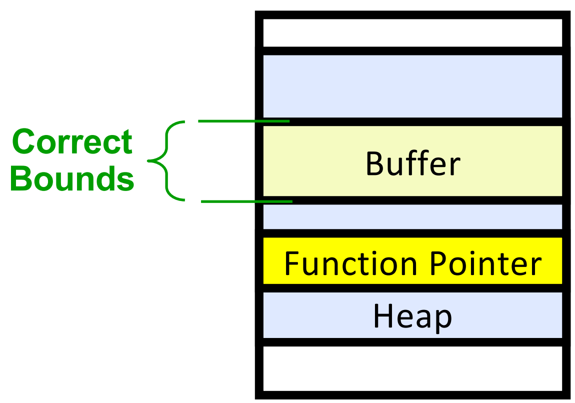


New programming languages → catalyst for real change

Rust: Memory-Safe Programming Language

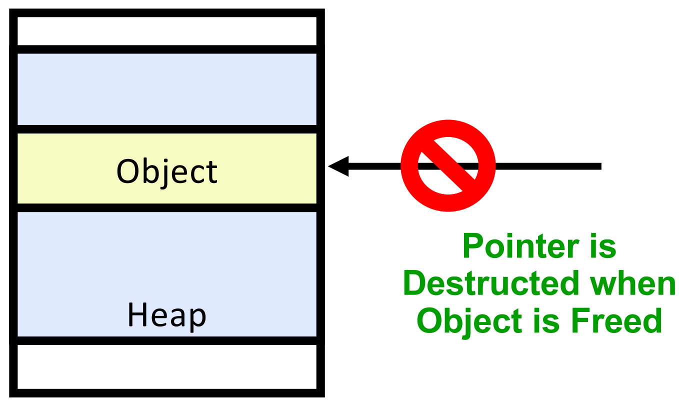
- **A systems programming language that is memory-safe**
- **Small language runtime: is translated to instructions directly; no need for language VMs**

- **Spatial safety (no buffer overflows):**
  - **Statically-sized objects: compile-time checks**
  - **Dynamically-sized objects: runtime bounds checks**



Spatial Memory Safety

- **Temporal safety (no use-after-frees):**
  - **Ownership: only one owner of object at a time**
  - **Burrowing: ownership can be temporarily transferred**



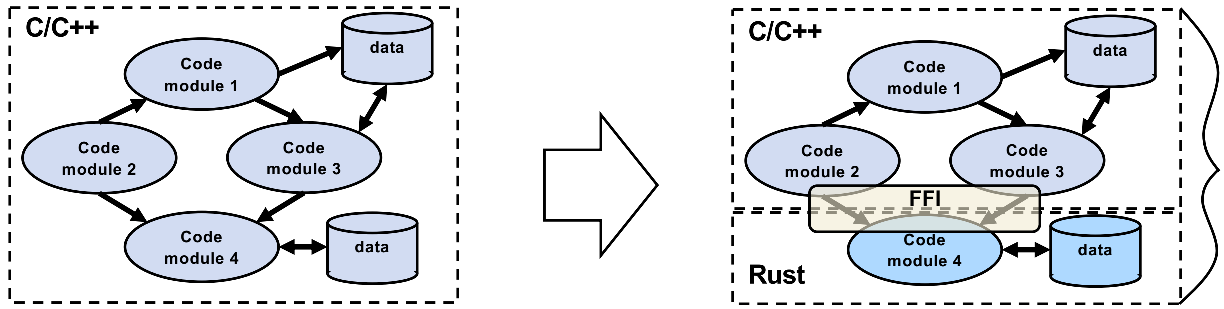
Temporal Memory Safety

LINCOLN LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Focus on Safe Rust

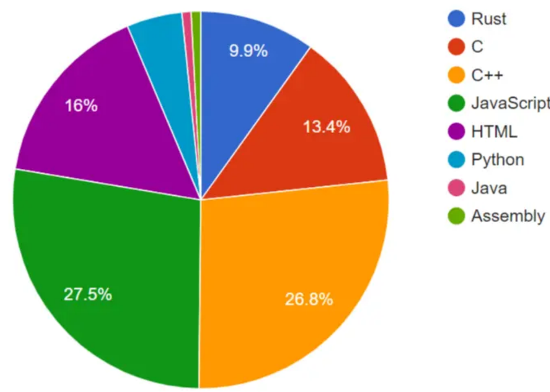
- Rust's checks can be disabled by using the `unsafe{}` keyword
- Done when Rust's checks are too restrictive
- Example: manipulating raw bits for interfacing with hardware devices in device drivers
- Unsafe Rust is trivially vulnerable to memory corruption like C/C++
- We focus on Safe Rust

Practical Deployment of Safe Languages



If not done carefully, incremental deployment of safe languages can reduce security

- What about legacy C/C++ code?
  - Rust/Go offer strong Foreign Function Interfaces (FFI)
    - FFI facilitates incremental adoption into legacy code bases
    - Results in a Multi-Language Application (MLA)
- Multi-language applications are common:
  - Rust: Firefox, Tor, Windows, Fuchsia, etc.
  -

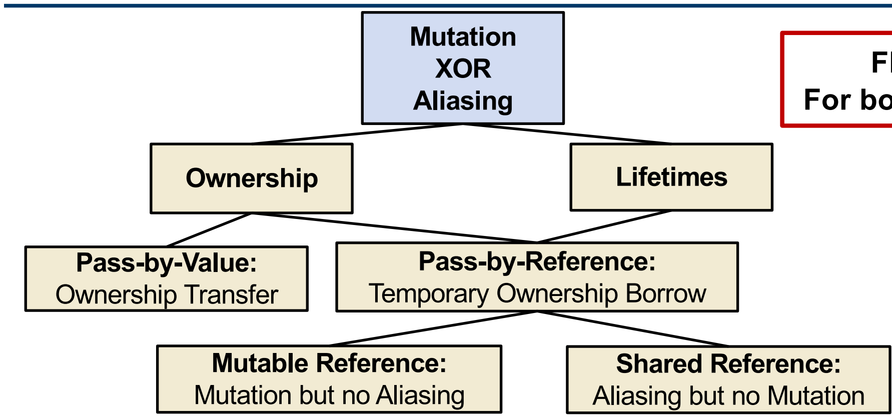


Firefox Language Breakdown

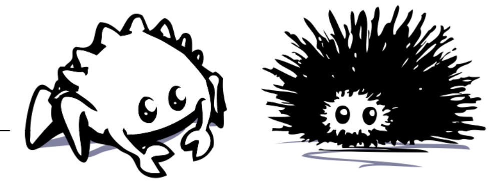
Safe languages are often gradually deployed into legacy code

https://www.programmersought.net/en/article/324364766.html

Rust Safety



FFI is fundamentally unsafe behavior:
For both intended and unintended interactions!

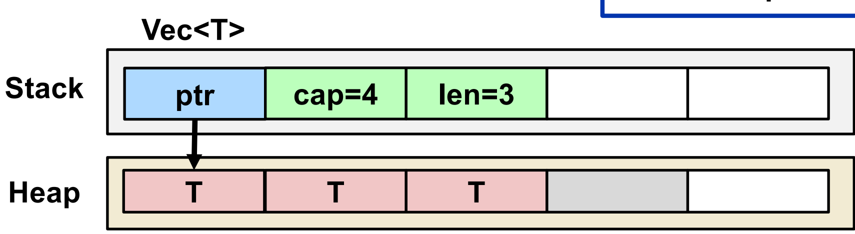


```
fn rust_fn() {
    // Create some data
    let mut v: Vec<i32> = vec![1, 2, 4];

    // Ownership borrow (mutable reference)
    v.push(3);

    // Manual memory modification requires unsafe
    unsafe { *v.as_ptr().add(1) = 8; }

    // Ownership borrow (shared reference)
    println!("{}", v[1]);

    // Ownership transfer
    give_me_a_vec(v); // automatically free'd on return

    // No longer owner, would result in an error:
    // v.push(4);
}
```

Escape Rust safety with "unsafe"

Multiple types of Ownership Transfer

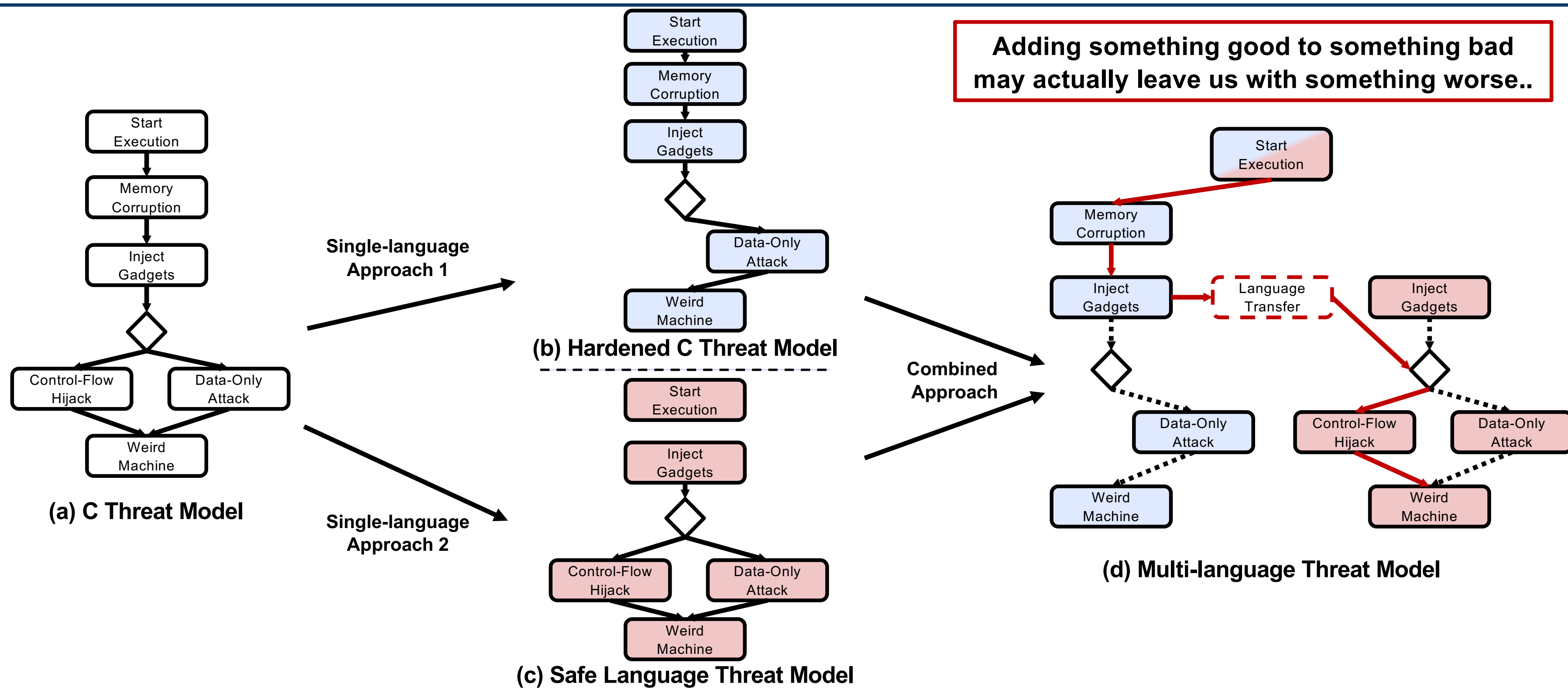
Spatial and temporal safety



Rust provides both spatial and temporal safety

Single vs. Multi-Language Application Threat Models

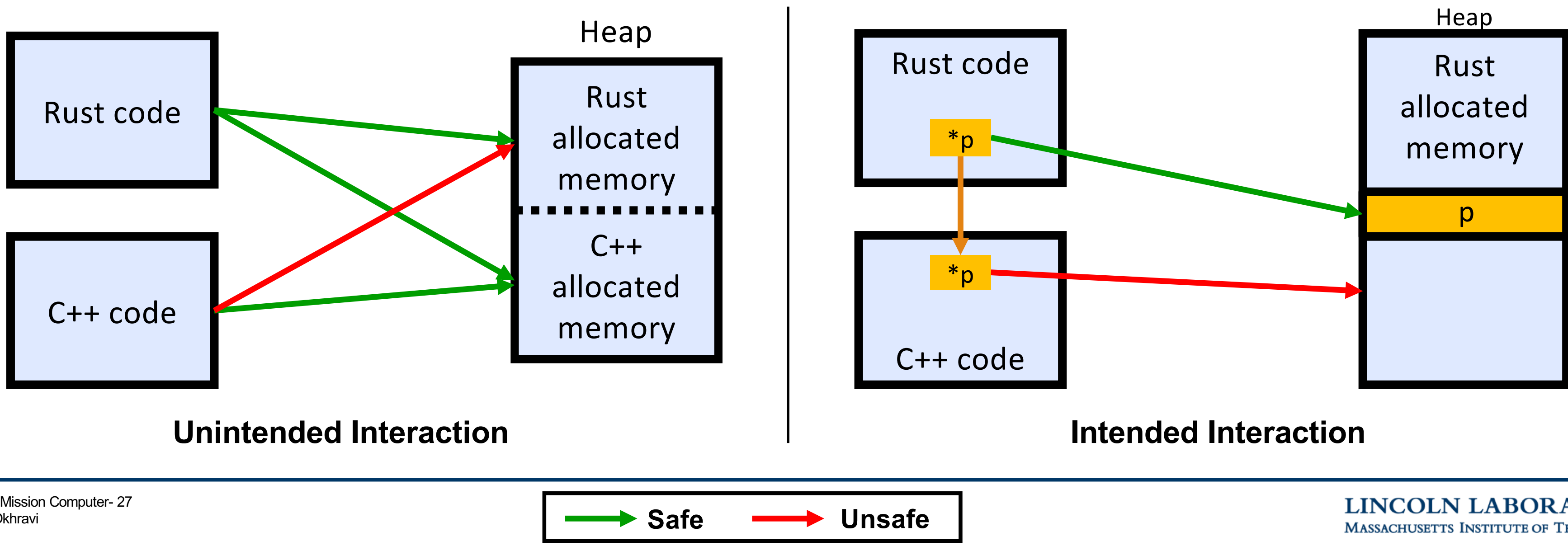
Adding something good to something bad may actually leave us with something worse..



MLA threat model is actually similar to the original C threat model

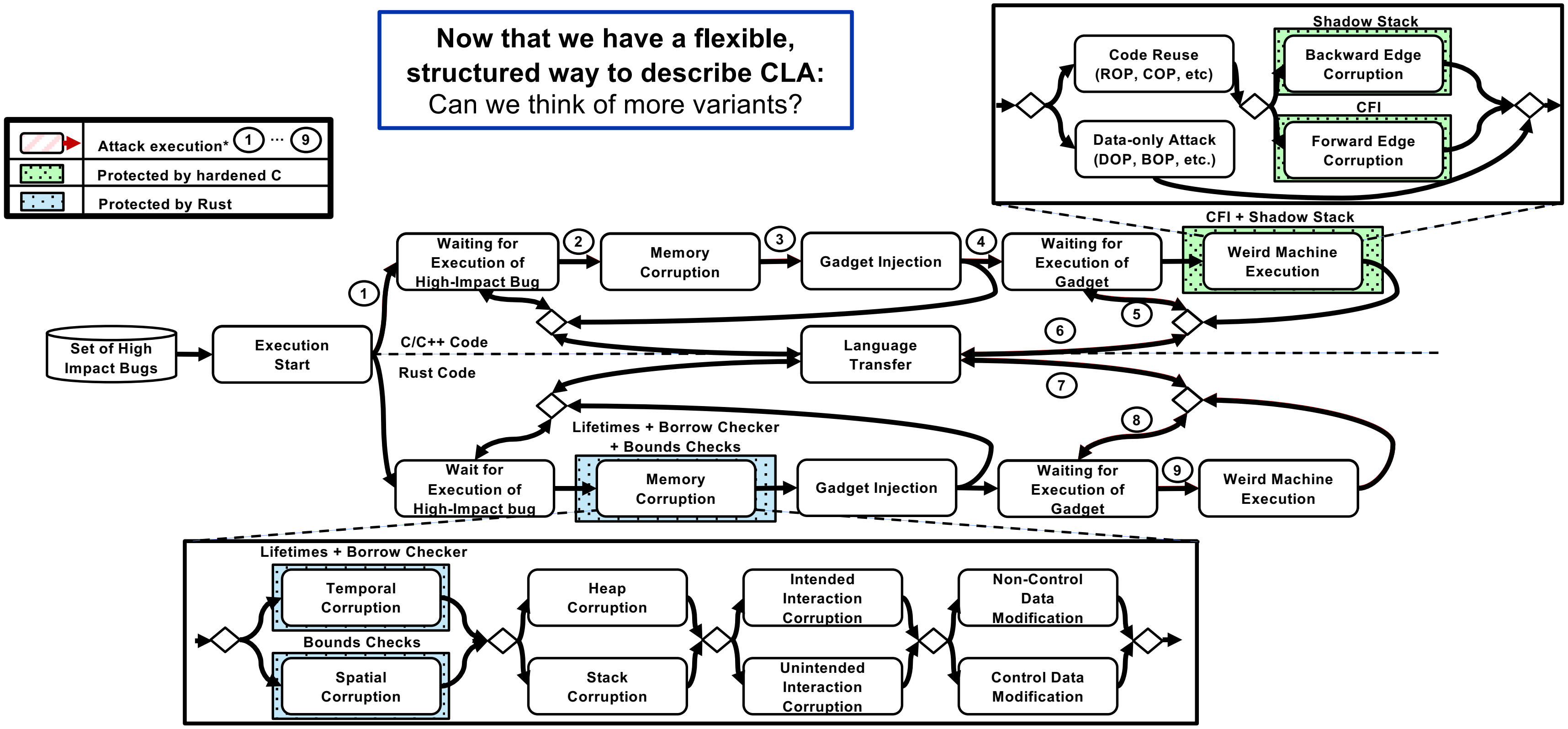
Problem Statement

- All C/C++ code cannot be immediately ported to Rust
- Real codebases ***incrementally*** port to Rust
- Rust code often exists alongside other languages, primarily C/C++
- Examples: Mozilla (Firefox), DropBox, Microsoft, Amazon, Discord, Facebook, etc.



CLA Attack Construction

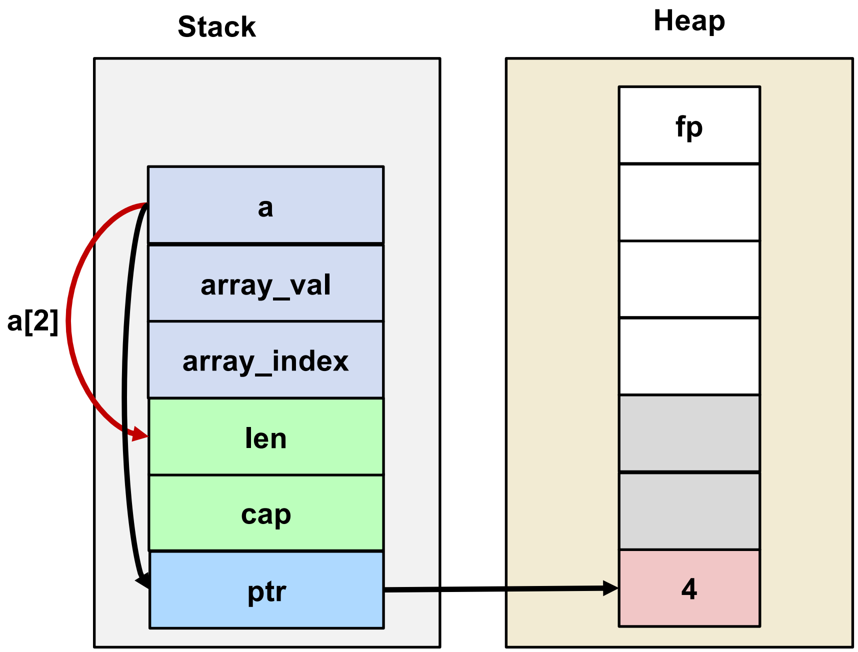
Now that we have a flexible, structured way to describe CLA:
Can we think of more variants?



Our graphical model can represent many forms of CLA

*Papaevripides, Michalis, and Elias Athanasopoulos. "Exploiting mixed binaries." *ACM Transactions on Privacy and Security (TOPS)* 2021

Variants of CLA: Corrupting Rust Dynamic Bounds



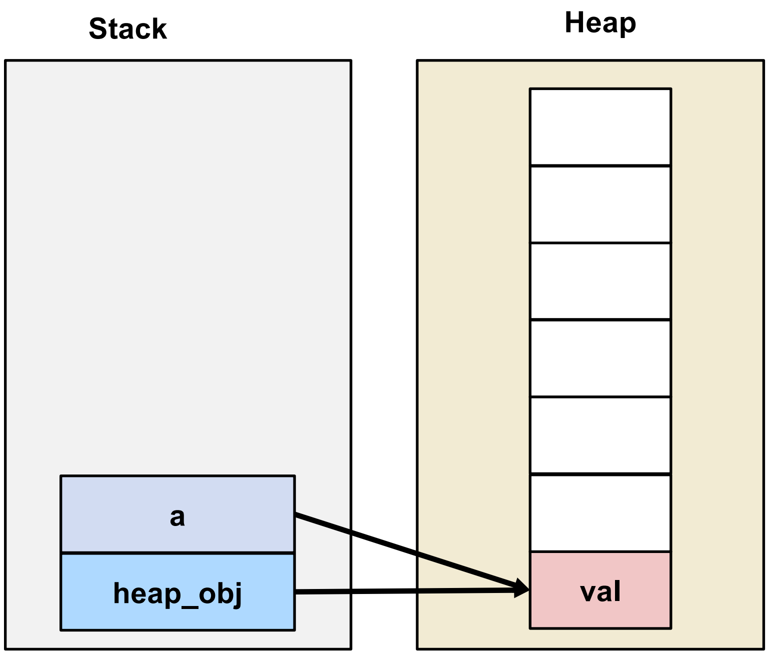

```
fn rust_fn(cb_fptr: fn(&mut i64)) {
    //Rust vectors have dynamic bounds
    let mut vecs: vec![4];

    unsafe{ vuln_fn(/*Ptr to vecs*/) }

    // C++ changed vecs size to 128!
    let vec_fp_addr: i64 = x.vecs[55];
}
```

C/C++ can corrupt the saved length of the vector to corrupt Rust dynamic checks

```
void vuln_fn(int64_t vec_ptr_addr) {
    // These values are set by a corruptible
    // source, e.g., user input
    int64_t array_index = 2;
    int64_t array_value = 128;

    int64_t* a = (void *)vec_ptr_addr;
    a[array_index] = array_value;
}
```

CLA can corrupt Rust's spatial safety

Variants of CLA: Corrupting Rust Lifetimes




```
fn rust_fn(cb_fptr: fn(&mut i64)) {
    let heap_obj: /* Rust heap allocation */

    unsafe{ vuln_fn(/*Ptr to heap_obj*/) }

    heap_obj[0] += 5; // UaF
}
```

```
// Frees object it does not own
void vuln_fn(int64_t obj_ptr_addr) {
    int64_t* a = (void *)obj_ptr_addr;

    //C/C++ frees Rust allocated object!
    free(a);
}
```

C/C++ can corrupt Rust's automatic memory management

CLA can corrupt Rust's <u>temporal</u> safety

Evaluation

Main security questions:

RQ1: How prevalent are language transitions?

RQ2: Are language transitions uniformly distributed or centralized?

We analyze Mozilla Firefox for our evaluation

Methodology and Metrics

- Call Sites
  - When a function is the *caller* of another function
    - Transfer Points: From one language to another
    - Indirect Calls: Through a register
    - Dynamic Calls: Through the program lookup table (PLT)
- Invocations
  - When a function is the *callee* of another function
    - Visitor Points: From one language to another
- Heavy Hitters
  - Investigate the distribution of language transitions across functions

Our measurements analyze the general extent of the problem

Results: Call Site Analysis

Each cell:

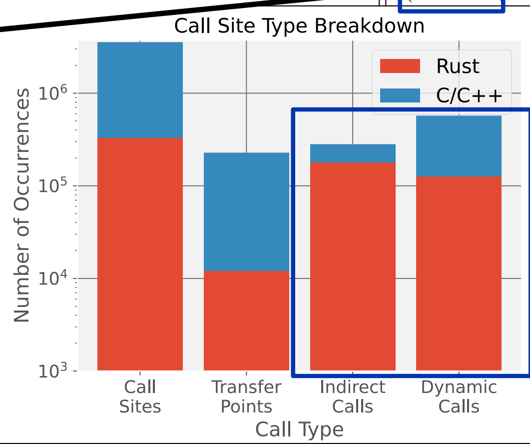
Raw Magnitude (Column %, Row %)

	Rust	C/C++	Entire Binary
Call Sites	**327,653** (100%, 9.23%)	**3,220,415** (100%, 90.77%)	**3,548,068** (100%, 100%)
Transfer Points	**12,118** (3.70%, 5.32%)	**215,778** (6.70%, 94.68%)	**227,896** (6.42%, 100%)
Indirect Calls	**179,598** (54.81%, 64.04%)	**100,843** (3.13%, 35.96%)	**280,441** (7.90%, 100%)
Dynamic Calls	**126,710** (38.67%, 22.15%)	**445,418** (13.83%, 77.85%)	**572,128** (16.13%, 100%)

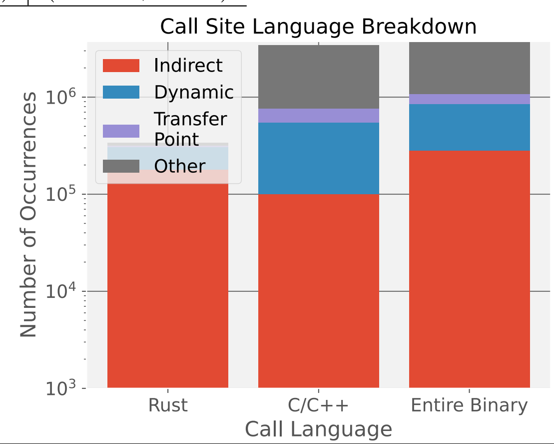
Many Rust indirect calls

Many Rust dynamic calls

Rust Transfer Points % looks small but magnitude is large



Especially compared to C/C++ behavior



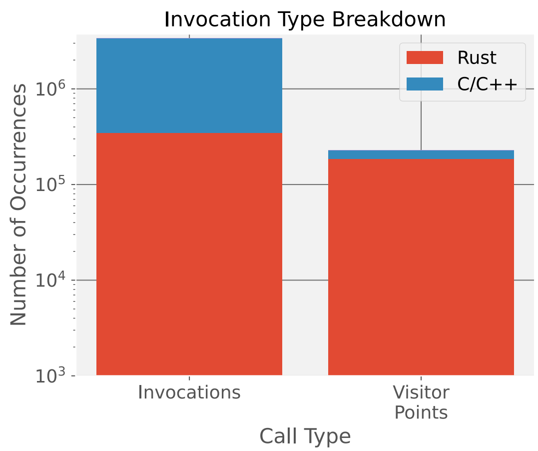
Abundant opportunities for CLA against Rust

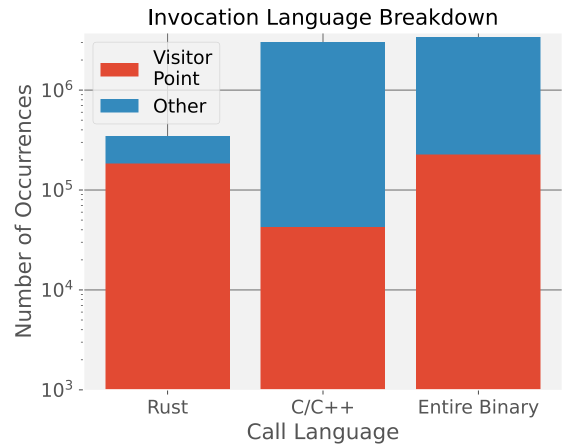
Results: Invocation Analysis

The majority of Rust invocations come from memory unsafe languages

	Rust	C/C++	Entire Binary
Invocations	**346,469** (100%, 10.25%)	**3,032,583** (100%, 89.75%)	**3,379,052** (100%, 100%)
Visitor Points	**184,799** (53.34%, 81.09%)	**43,097** (1.42%, 18.91%)	**227,896** (6.74%, 100%)

Most language transitions go from C/C++ to Rust





Rust mostly acts as a service module for C/C++

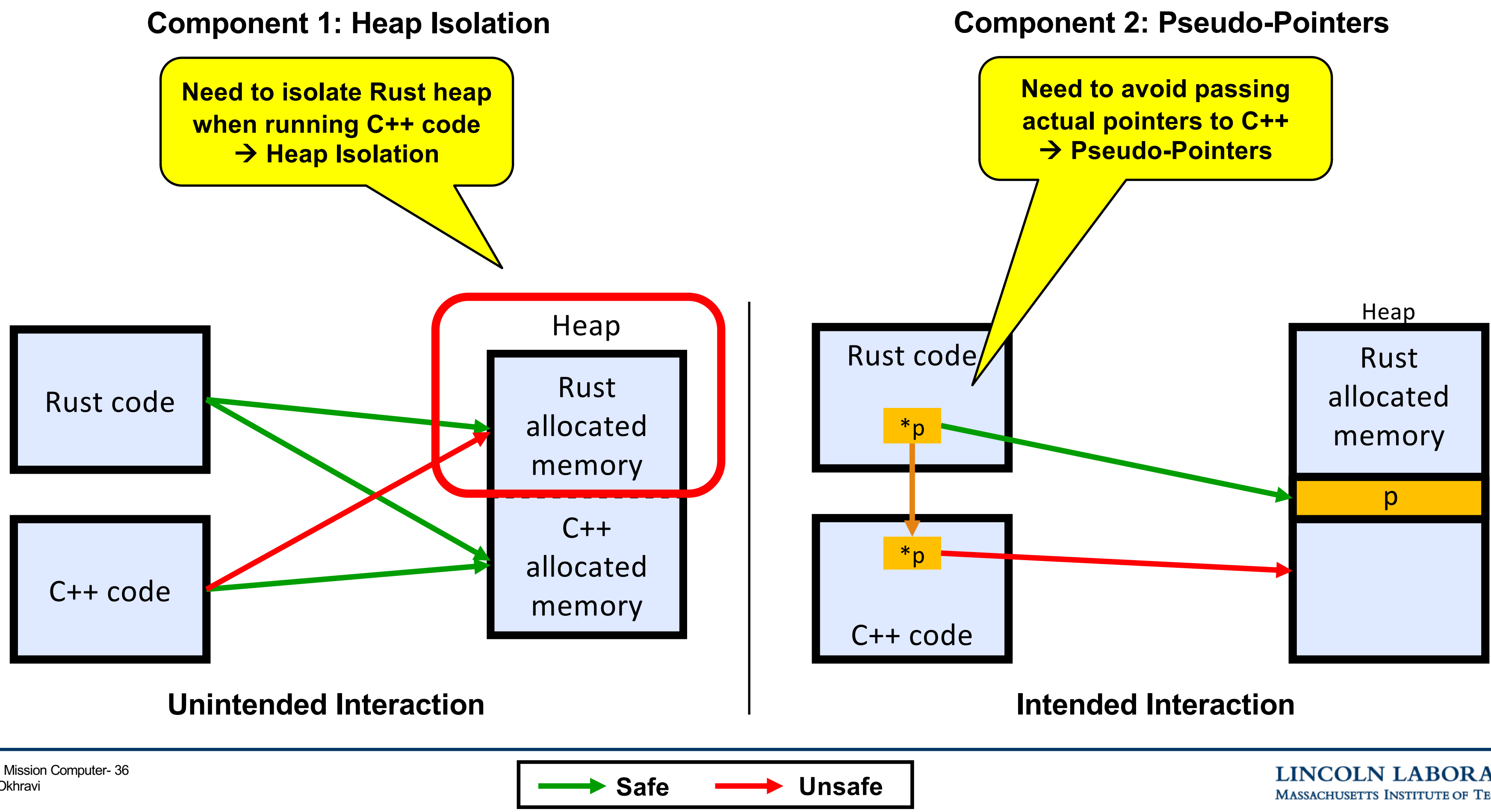
Results:
Heavy Hitters Analysis

	Rust	C/C++
Top Functions with Call Sites	1. assert_initial_values_match@libxul (**588**) 2. get_longhand_property_value<alloc>@libxul (**464**) 3. get_longhand_property_value<nsstring>@libxul (**459**)	1. CreateInstance@libxul (**1,631**) 2. generateBodyEv@libxul (**1,160**) 3. run@libxul (**846**)
Top Functions with Transfer Points	1. main@crashreporter (**55**) 2. main@modutil (**25**) 3. main@logalloc-replay (**24**)	1. Unified_cpp_protocol_http3@libxul (**84**) 2. UIShowCrashUI@crashreporter (**54**) 3. nsWindow@libxul (**49**)
Top Functions with Invocations	1. as_bytes@libxul.so (**930**) 2. state@libxul (**554**) 3. _Unwind_Resume@plt (**520**)	1. AnnotateMozCrashReason@libxul (**134,254**) 2. ReportAssertionFailure@libxul (**131,545**) 3. Array_RelocateUsingMemutil@libxul (**17,475**)
Top Functions with Visitor Points	1. _Unwind_Resume@std (**488**) 2. as_str_unchecked@libxul (**25**) 3. qcms_transform_data@libxul (**24**)	1. __assert_fail@GLIBC (**4388**) 2. ostream@GLIBC (**3326**) 3. strlen@GLIBC (**1294**)

Rust to C/C++ transfers most often are calls to libc → **May want to focus future defensive work in this area**

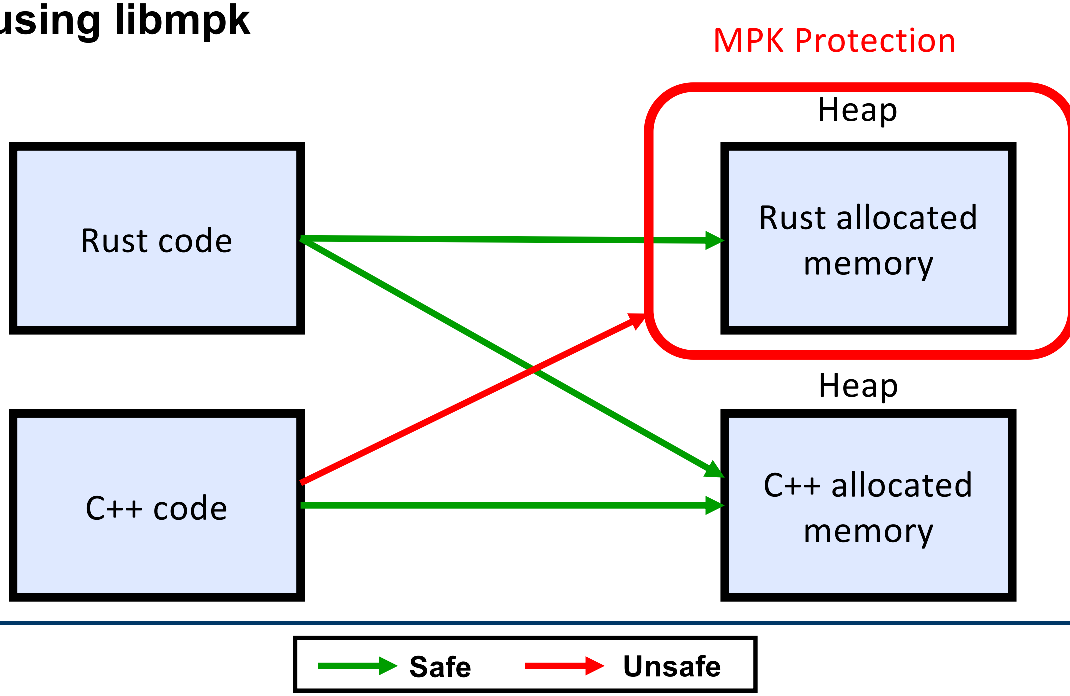
Most transfers from Rust → libc

Preventing CLAs



Preventing Unintended Interactions: Heap Isolation

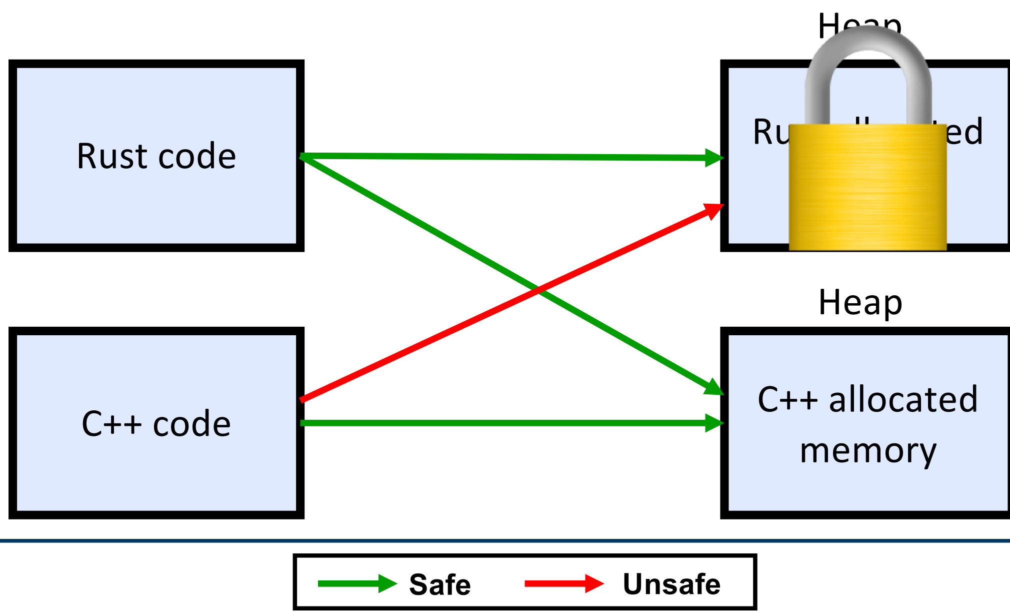
- Uses Intel Memory Protection Keys (MPK) to isolate Rust heap from C++ heap
- Modified Rust standard allocator
- Code to switch permission included around all external call sites
- Implemented using libmpk



Heap Isolation Implementation

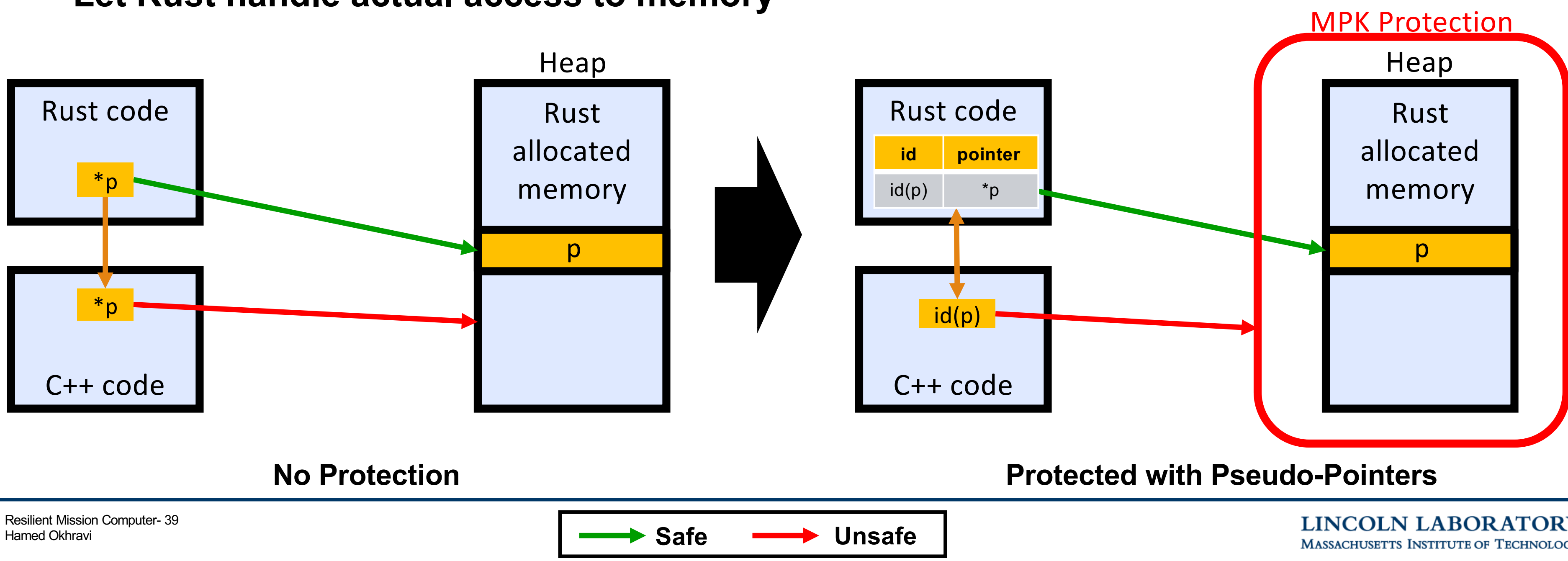
Permission Switching Code

```
asm! (" rdpkru ", in(" ecx") ecx , lateout (" eax") eax , lateout (" edx") _);
eax = ( eax & !PKRU_DISABLE_ALL ) | PKRU_ALLOW_READ ;
asm! (" wrpkru ", in(" eax") eax , in(" ecx") ecx , in(" edx ") edx );
```



Securing Intended Interactions: Pseudo-Pointers

- Replace real pointers with pseudo-pointers (identifiers)
- Pass pseudo-pointers to C++
- Replace C++ pointer operations with calls to getter/setter methods (an LLVM pass)
- Let Rust handle actual access to memory



Pseudo-Pointer Implementation

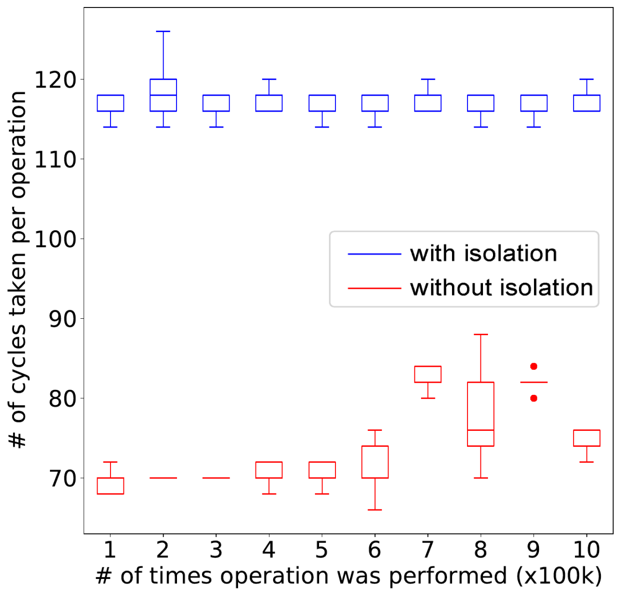
```
int add5 ( MyStruct * const p) {
      p->x += 5;
}
```

No Protection

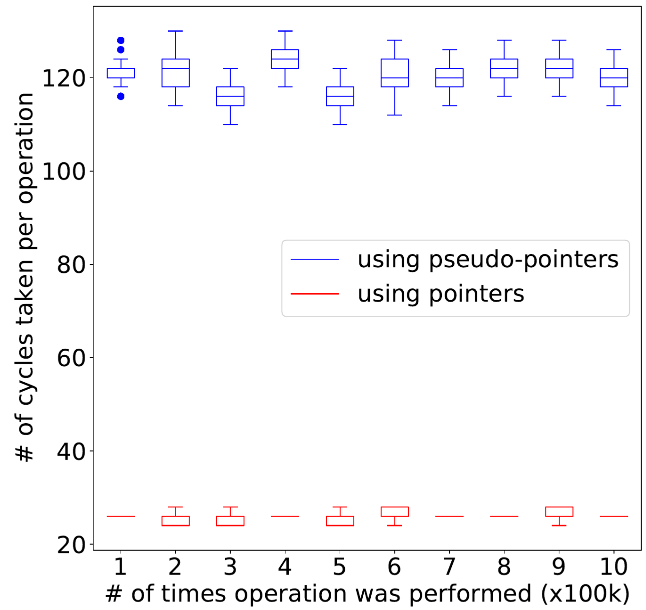
```
int add5 (ID < MyStruct > const p) {
      x = get_x_in_MyStruct (p);
      set_x_in_MyStruct (p, x +5);
}
```

Protected with Pseudo-Pointers

Evaluation: Micro-Benchmarking



Heap Isolation
Average ~50 cycles



Pseudo-Pointers
Average ~100 cycles

Publications

1. [**NDSS**] Derrick McKee, Yianni Giannaris, Carolina Ortega, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow, "Preventing Kernel Hacks with HAKC," NDSS, San Diego, CA, 2022
   Distinguished Paper Award
2. [**NDSS**] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi, "Cross-Language Attacks," NDSS, San Diego, CA, 2022
3. [**CSUR**] Nathan Burow, Bryan Ward, Richard Skowyra, Roger Khazan, Howard Shrobe, and Hamed Okhravi, “TAG: Tagged Architecture Guide”, May 2022
4. [**IEEE Security & Privacy**] Hamed Okhravi, “A Cybersecurity Moonshot”, IEEE Security & Privacy, Vol. 19, No. 3, 2021
5. [**ACSAC**] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow, "Keep Safe Rust Safe with Galeed ," ACSAC, December 2021
6. [**IEEE Security & Privacy**] Hamed Okhravi, et al. “Perspectives on the SolarWinds Hack”, IEEE Security & Privacy, Vol. 19, No. 2, 2021
7. [**DSN**] Chad Spensky, Nathan Burow, and Hamed Okhravi, et al., “Glitching Demystified”, DSN, 2021
8. [**AsiaCCS**] Chad Spensky and Hamed Okhravi, et al., “Conware: Automated Modeling of Hardware Peripherals”, AsiaCCS, 2021

\+ many more theses and reports

Our vision article featured on the cover of prestigious IEEE Security & Privacy May/June 2021

Conclusion

- Modern computer systems are hard-to-secure because of their legacy design
- RMC seeks to rethink the computer design with security as its central
- Two of our contributions:
  - A practical approach for enforcing compartmentalization on Linux on commodity processors
  - Understanding cross-language attacks and securing applications against them
- Future research goals: compartmentalization in other SW stack layers, enforcement on processors without security extensions, and designing for least privilege (new languages, app design process)