# CS260 – Advanced Systems Security

Future

June 2, 2025

# Privilege Separation

- Has been promoted for some time
  - Software-Fault Isolation (1993)
  - Kernel driver isolation (1990s)
  - OpenSSH (early 2000s)
- Can be a time-consuming task
  - Automate – not there yet
- Questions
  - What is the state of automating privilege separation?
  - Do we still need it?

# Privilege Separation Goal

- What should the goal be for implementing privilege separation?

# Privilege Separation Goal

- What should the goal be for implementing privilege separation?
- Want to achieve
  - Input: Legacy program
  - Output: K separated components with minimal privilege

# Privilege Separation Goal

- What should the goal be for implementing privilege separation?
- Want to achieve
  - Minimal privilege for each code component
    - When should components be created?
    - Are there clear security properties that can drive this?
  - Without too much programmer effort
    - Automated
  - While retaining good performance
    - Statically verifiable where possible
    - Optimize trade-off of security and performance

# Privilege Separation Goal

- What should the goal be for implementing privilege separation?
- Want to achieve
  - Minimal privilege for each code component
    - When should components be created?
    - Are there clear security properties that can drive this?
  - Without too much programmer effort
    - Automated
  - While retaining good performance
    - Statically verifiable where possible
    - Optimize trade-off of security and performance

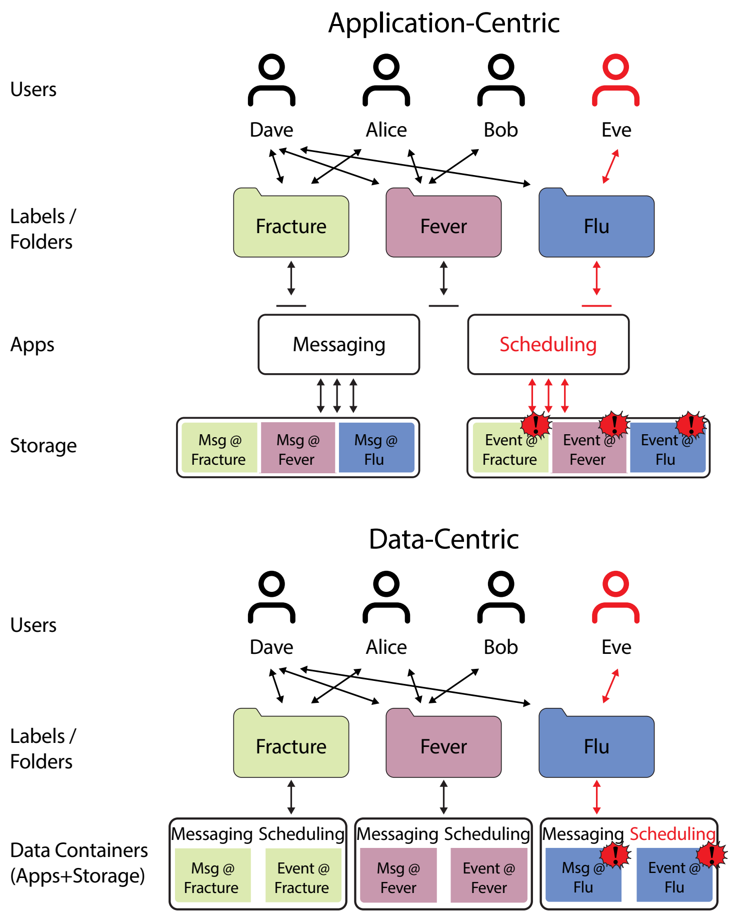
# DATS

- Proposes a security architecture for web applications

- Single use services
  - Launch service for particular user/request
    - Unique web-application instance (container)
- Access control
  - Limit each single-use service to only the user/request permissions needed
    - For the specific request
- Privilege separation
  - Isolate untrusted front-end from processing of key data
    - Within web applications, but trust the backend storage (storage declassifier)

# DATS

- Different view of sharing



# DATS

- Different view of sharing

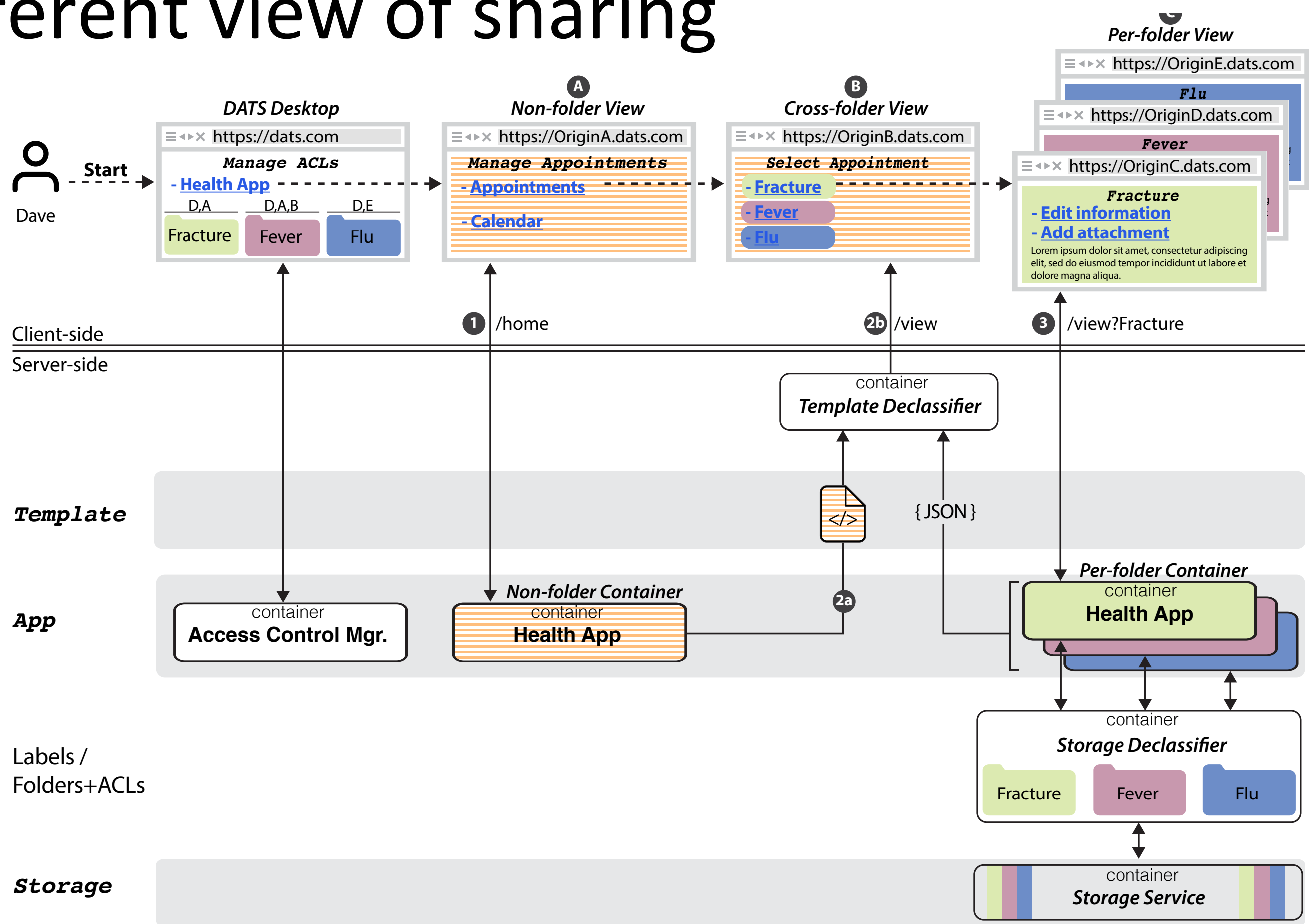



**Figure 2.** *Example web page flow from a user ("client-side"), DATS's main components, and an application's* `app-template-storage` *components (and their relation to MVC). Application code, application data, and storage services are untrusted (grayed areas and colored boxes), while DATS's trusted components (boxes with white background) enforce folder non-interference. Application components run inside OS-level containers, which can very easily enforce per-folder MAC policies. Note that the client's browser is allowed to run untrusted application code (e.g., JavaScript).*

# DATS – Take Aways

- Questions
- Do programmers know how to build web application instances?
  - Is this an automated privilege separation task?
- Can we enforce information flow guarantees comprehensively?
  - Currently SELinux
  - Should we use DIFC?
- Can we really trust the backend?  Do we really need to?
  - Is this another privilege separation problem?

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

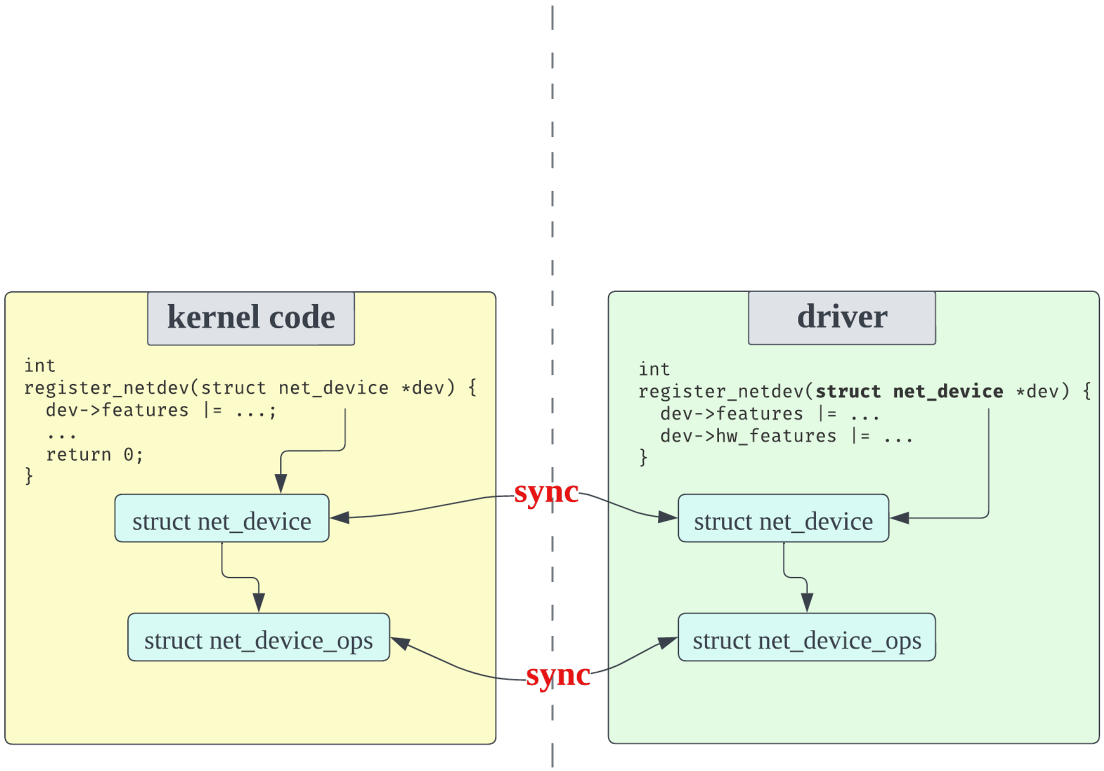
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

# KSplit: Automating Device Driver Isolation


**Yongzhe Huang[1], *Vikram Narayanan[2]*, David Detweiler[2], Kaiming Huang[1], Gang Tan[1], Trent Jaeger[1], and Anton Burtsev[2,3]**

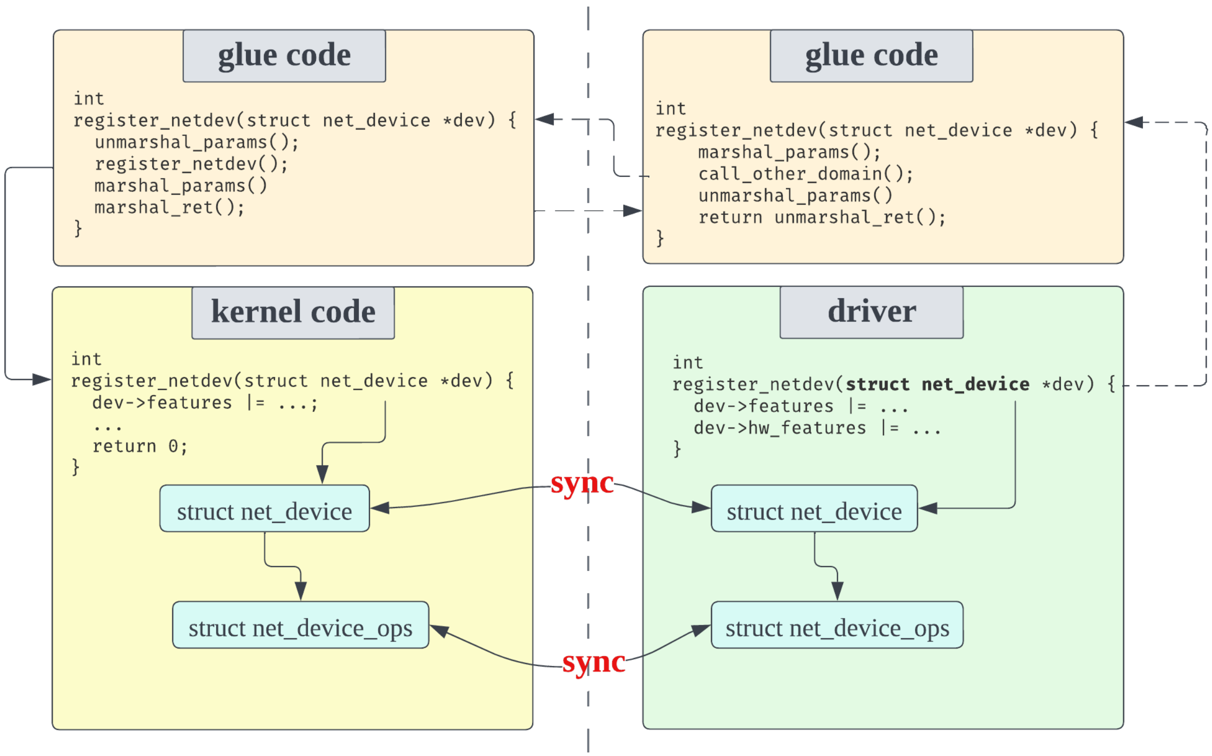
*[1]Penn State University* *[2]University of California, Irvine* *[3]University of Utah*

# Driver Isolation




- Separate memory space
  - Two copies of object hierarchies
  - Keep them synchronized

# Driver Isolation

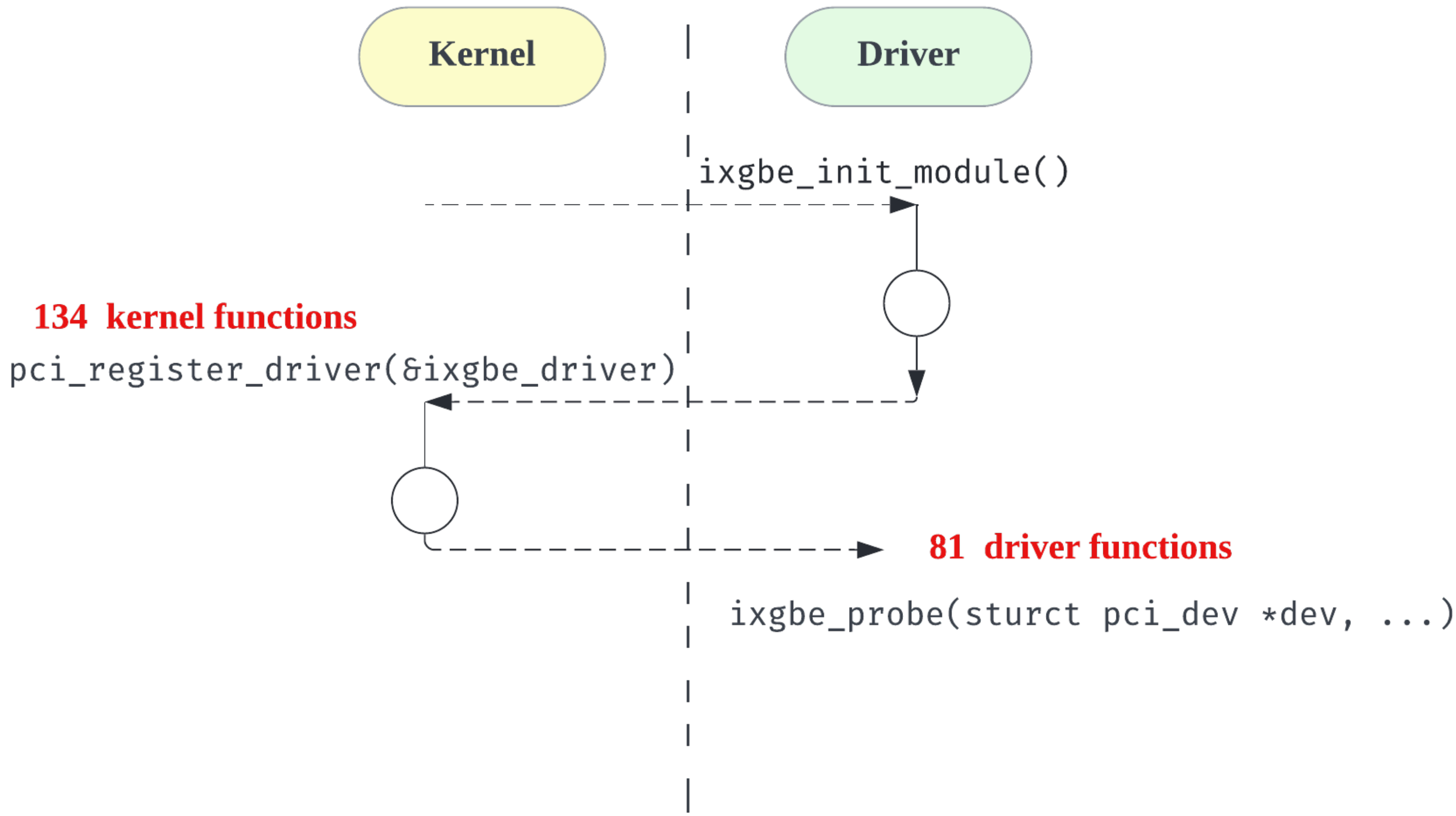



- Separate memory space
  - Two copies of object hierarchies
  - Keep them synchronized
- Glue code
  - Marshal/unmarshal params
  - Interface definition language (IDL) spec
  - Generated with IDL compiler

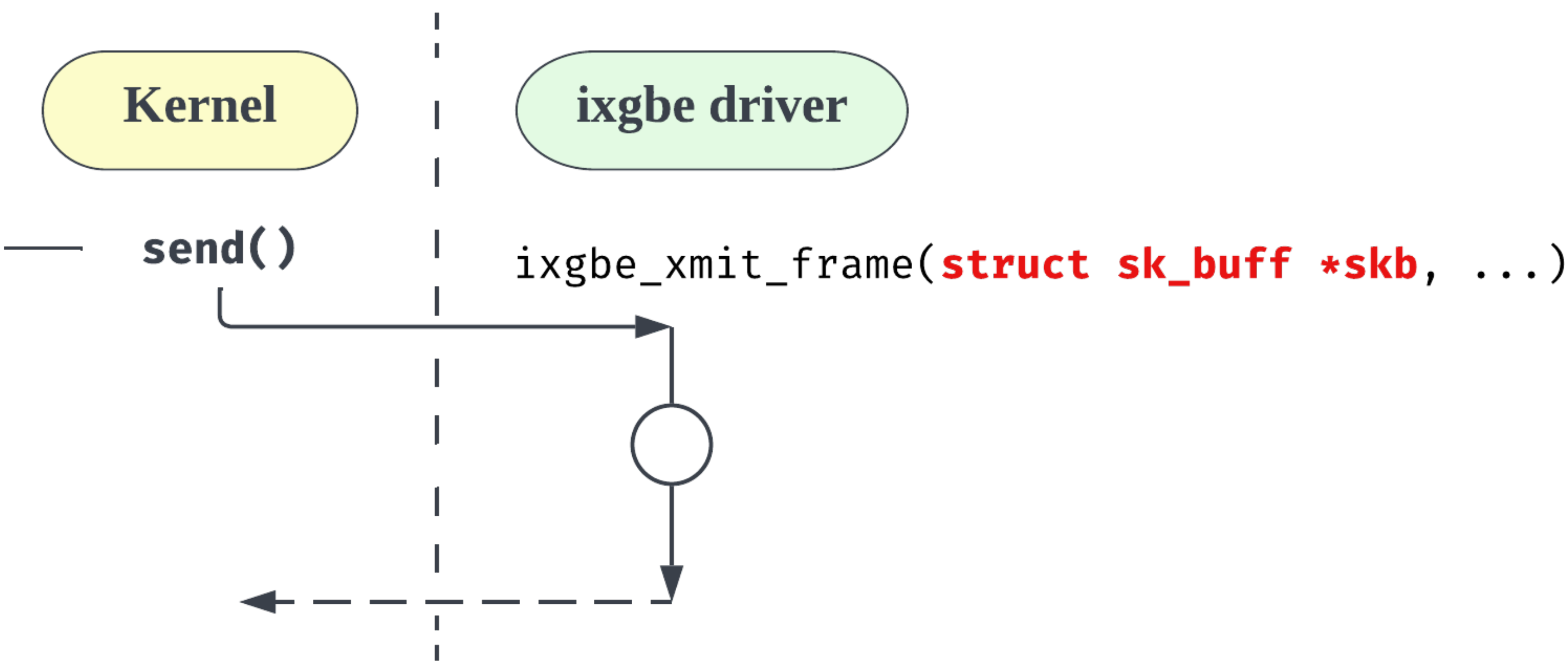
# Automating Driver Isolation

- Can driver isolation be automated?
  - What are the challenges?

# Challenge: Large Interface

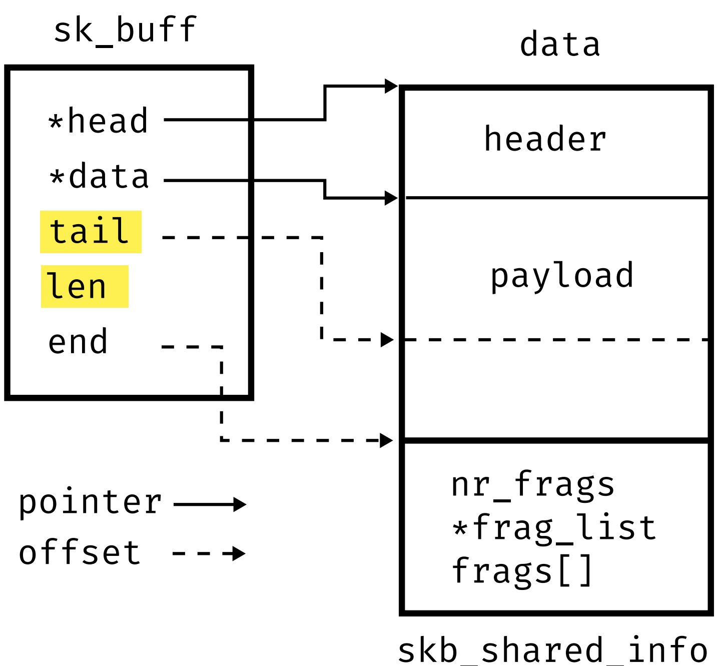


# Challenge: Complex Data Exchange



# Challenge: Complex Data Exchange

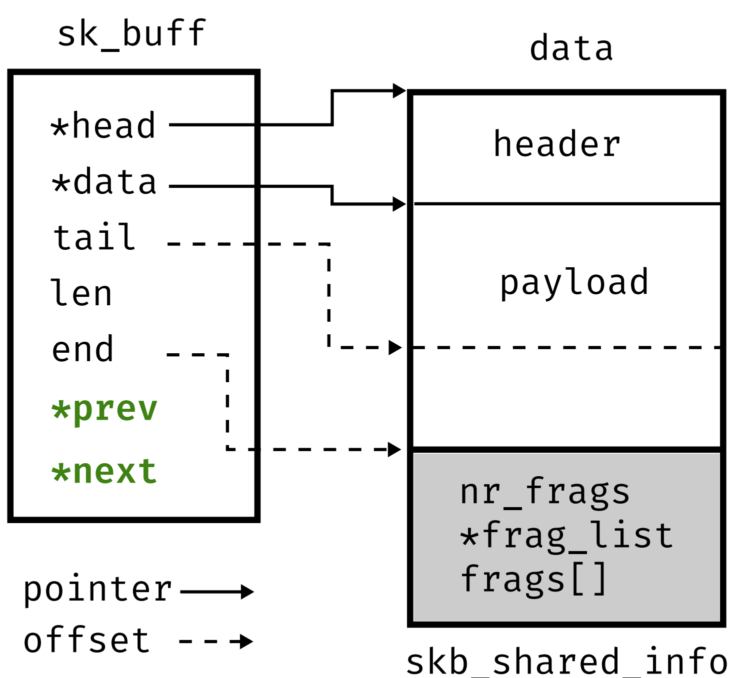
```
ixgbe_xmit_frame(struct sk_buff *skb, …)
```




- Represents a network packet
- Has 66 fields (5 pointers)
- 3,132 fields (1,214 pointers) are recursively reachable
- But only a small subset are accessed by both kernel and driver (shared)
  - 8 shared fields for this API

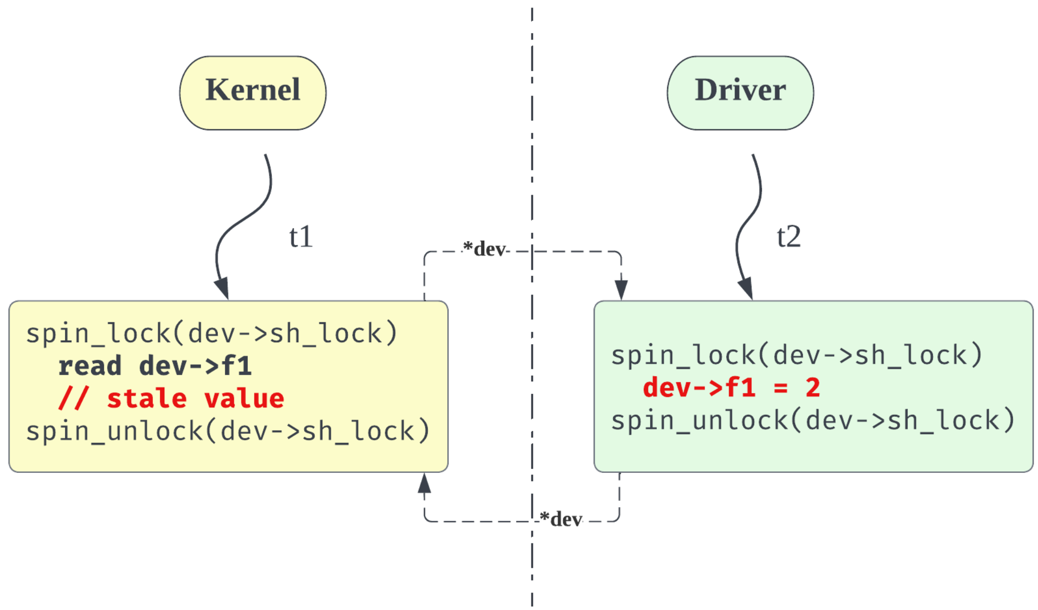
# Challenge: Kernel Idioms

```
int ixgbe_xmit_frame(struct sk_buff* skb, …)
```




- Pointers
  - Singleton, array
  - Linked list
  - Collocated data structures
- Sized and sentinel arrays
- Special pointers (e.g., `__user`, `__iomem`)
- Tagged unions
- Return error as ptr (e.g., `ERR_PTR`)

# Challenge: Concurrency

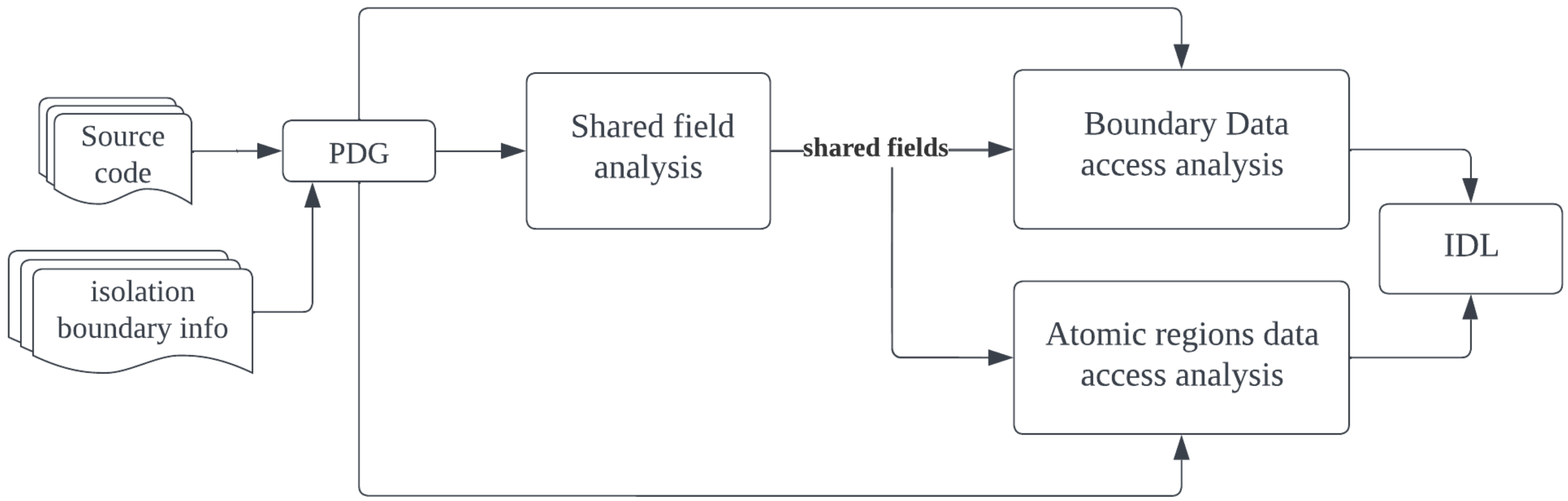



- spin/mutex lock
- driver specific lock, e.g., rtnl_lock
- atomic operations, e.g., set_bit
- read-copy update (RCU)
- sequential lock

# KSplit Goals

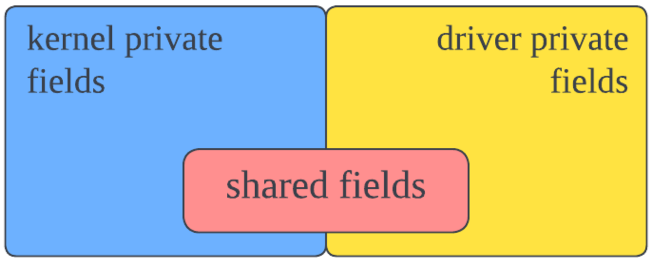
- Build a set of static analyses to generate the component-switch code (IDL) automatically (mostly) to
  - Isolate the complete driver
  - Identify shared/private data on the large interface boundary
  - Ensure each domain has the updated copy of the data structure
  - Identify marshaling requirements for the kernel idioms
  - Identify atomic regions that access shared data
- Prior work
  - Microdrivers (isolated the control plane of the driver)

# KSplit Workflow



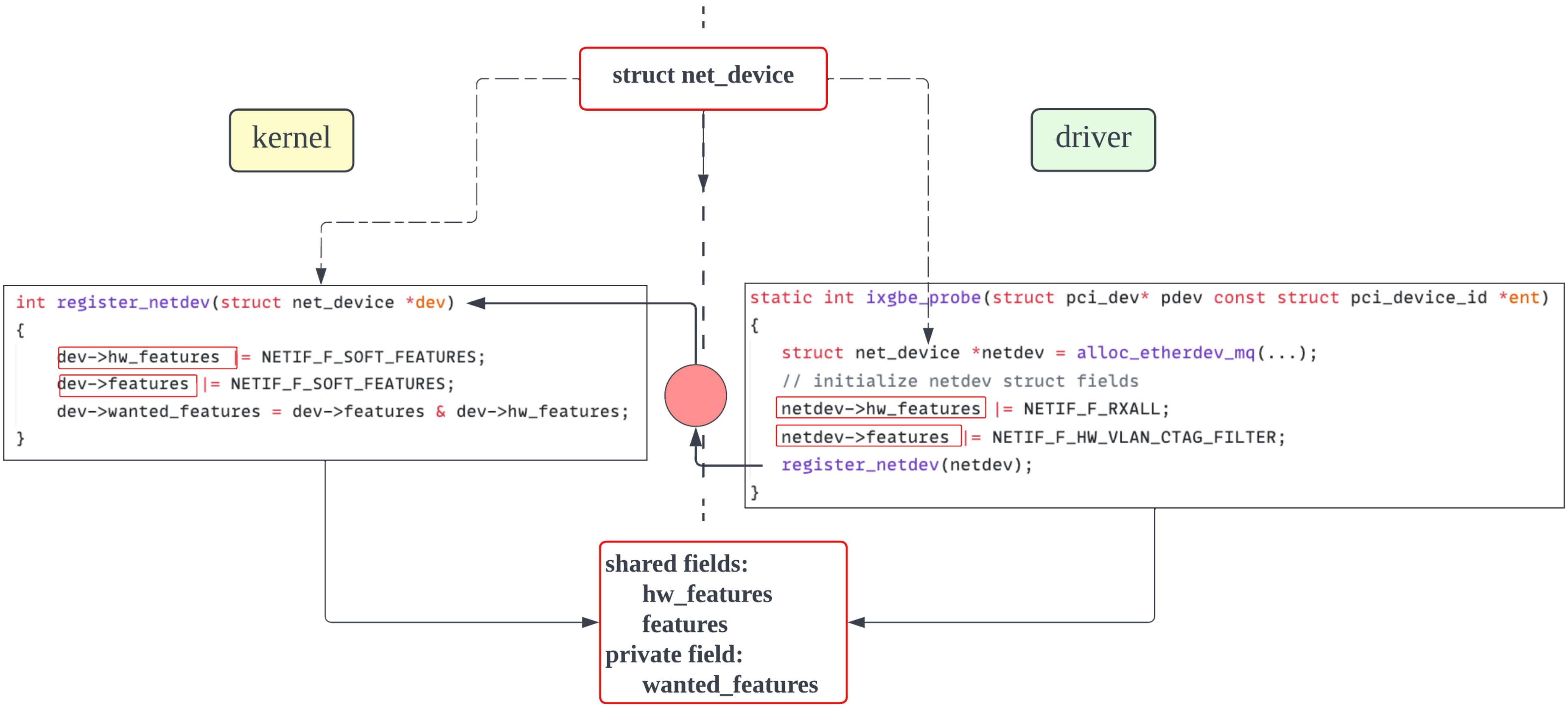

- **Input**: source code of kernel and target isolated driver
- **Output**: IDL file that specifies the communication interfaces and data synchronization requirements

# Shared Field Analysis

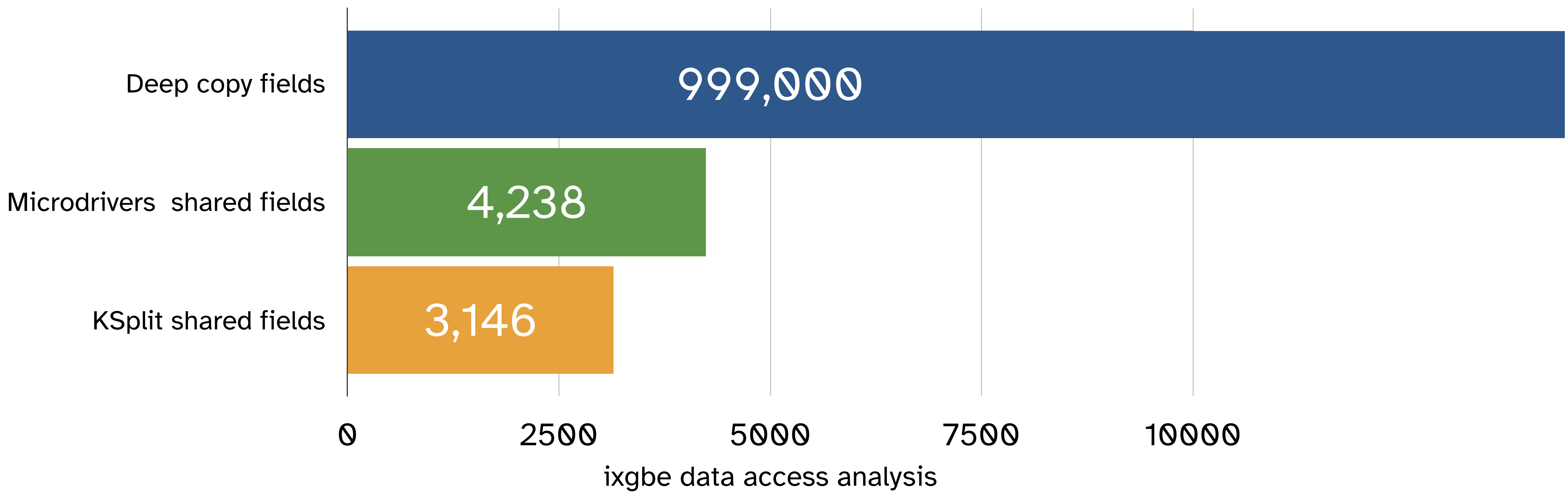



- **Input:**
  - data structure types on all the interface functions for the driver under analysis
- **Output:**
  - the set of struct fields accessed by both the kernel and this driver

# Shared Field Analysis



# Case Study: ixgbe driver



# Case Study: ixgbe driver

| | singleton | array | string | wild pointer (void) | wild pointer (other) |
|---|---|---|---|---|---|
| manual | 0 | 27 | 0 | 1 | 3 |
| handled | 1261 | 92 | 2 | 142 | 1 |

# Case Study: ixgbe driver

- Source code - 27,000 lines
- Generated IDL spec - 2000 lines
- Pointer misclassifications - 7
- Warnings - 65 (33 anonymous unions, 16 arrays, wild pointers)
  - IDL (changes) - 53 lines
  - Driver (changes) - 19 lines

# OptiSan: Using Multiple Spatial Error Defenses to Optimize Stack Memory Protection within a Budget

Rahul George[1], Mingming Chen[2], Kaiming Huang[2], Zhiyun Qian [1], Tom La Porta[2], Trent Jaeger[1]

1 UC Riverside

2 Penn State University

- Stack Spatial Memory errors *persist*
  - Existing C/C++ code

**200+ Stack CVEs over 3 years**

- Consider Apache Web Server (httpd)

```
static remoteip_parse_status_t
remoteip_process_v1_header(.. ,
proxy_header *hdr ,..)
{
    char *host;
    char buf[sizeof(hdr->v1.line)];
    ...........
    // Stack memory access
    strcpy(buf, hdr->v1.line);
    .........
    ..........
}
```


```
static remoteip_parse_status_t
remoteip_process_v1_header(.. ,
proxy_header *hdr ,..)
{
    char *host;
    char buf[sizeof(hdr->v1.line)];
    ...........
    // Stack memory access
    strcpy(buf, hdr->v1.line);
    .........
    ..........
}
```

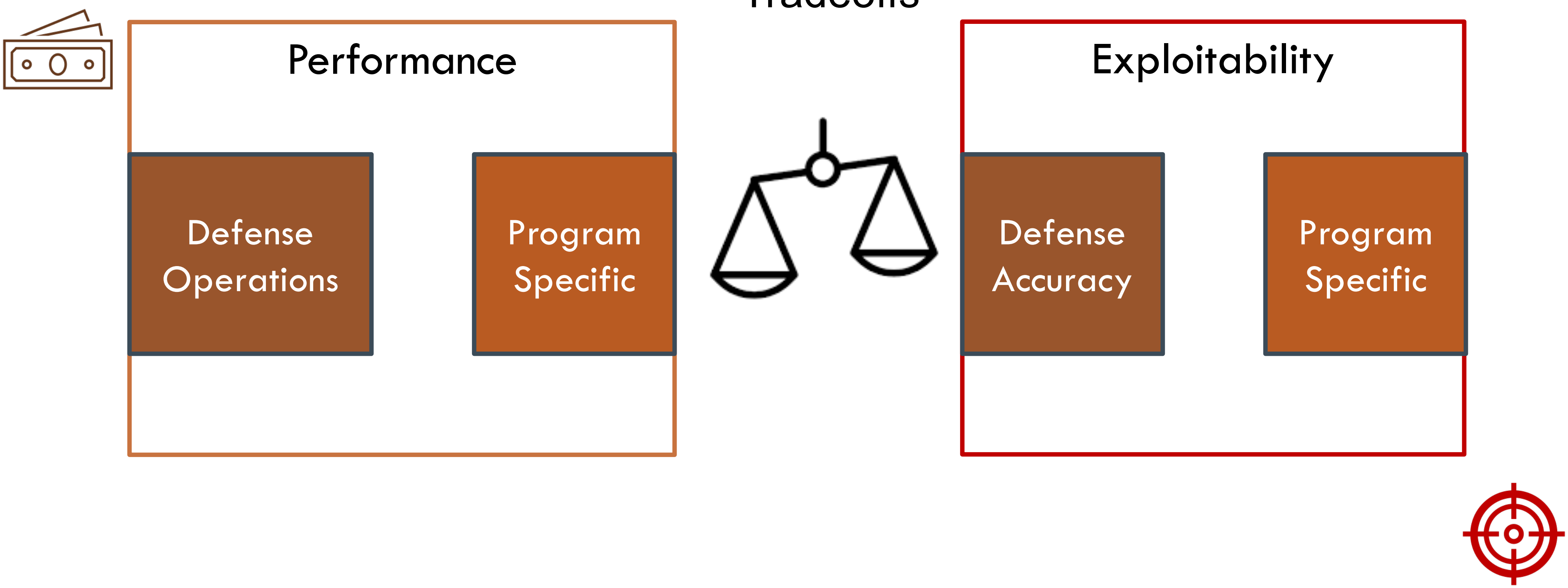
```
static remoteip_parse_status_t
remoteip_process_v1_header(.. ,
proxy_header *hdr ,..)
{
    char *host;
    char buf[sizeof(hdr->v1.line)];
    ...........
    // Stack memory access
    strcpy(buf, hdr->v1.line);
    .........
    ..........
}
```


**2,969 unsafe operations**

A memory access that may go outside of the memory region of the intended object

- *Q*: How do we *best* protect software?

# Tradeoffs

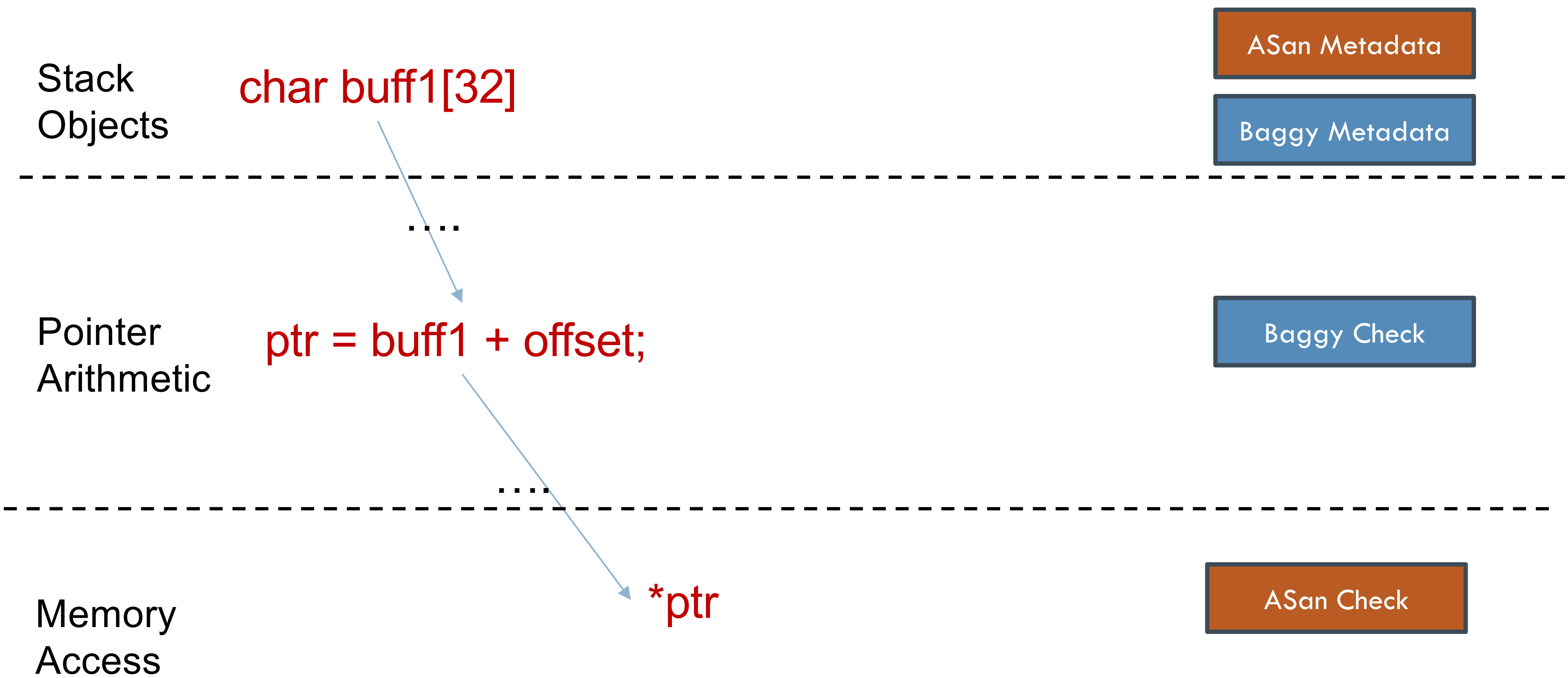

Performance
Defense Operations
Program Specific
Exploitability
Defense Accuracy
Program Specific

## Identity-Based Defenses

- Relies on the *intended referent*
- ***Detects*** pointer value is **<u>out of bounds</u>**
- **Check** performed at ***<u>pointer arithmetic</u>***
- **Metadata** tracks ***<u>bounds</u>***
- *Soft Bounds, Baggy Bounds*

## Location-Based Defenses

- Relies on the *invalid memory*
- **Detects <u>invalid memory access</u>**
- **Check** at ***<u>memory access</u>***
- **Metadata** tracks ***<u>validity</u>*** of memory (valid, invalid)
- Limited by the size of the invalid memory and *<u>may be bypassed</u>*
- *Purify, ASan*


Stack Objects
char buff1[32]
ASan Metadata
Baggy Metadata
….
Pointer Arithmetic
ptr = buff1 + offset;
Baggy Check
….
Memory Access
*ptr
ASan Check

- **Model defense overhead** to enable **fine grained** application
- Apply **multiple defenses** considering **tradeoffs**
- **Maximize** stack **protection** by modeling exploitability and performance
- Within a **cost budget**

- **How to model defense overhead for a program to enable incremental application at the operation granularity?**
  - ASAP (IEEE S&P 2015)
    - Imprecise – overheads of check operations only (frequency)
  - Debloating Asan (Usenix '22), Fuzzan (Usenix '20)
    - Coarse grained - Not at operation granularity (different goal)

- **How to model exploitability *of a program* w.r.t. stack spatial memory errors?**

- **How to *apply* multiple defenses considering these tradeoffs for a given *budget*?**
  - Several cost-based approaches apply a single defense without considering exploitability
    - ASAP (IEEE S&P 15), SanRazor (OSDI 21), PartiSan (RAID) …

- **Idea** – Treat applying spatial error defenses defenses – e.g., ASan and Baggy Bounds – as an **optimization problem** considering performance and exploitability

- We model the **cost of spatial memory error defenses** that captures the cost of different operations - metadata and check operations

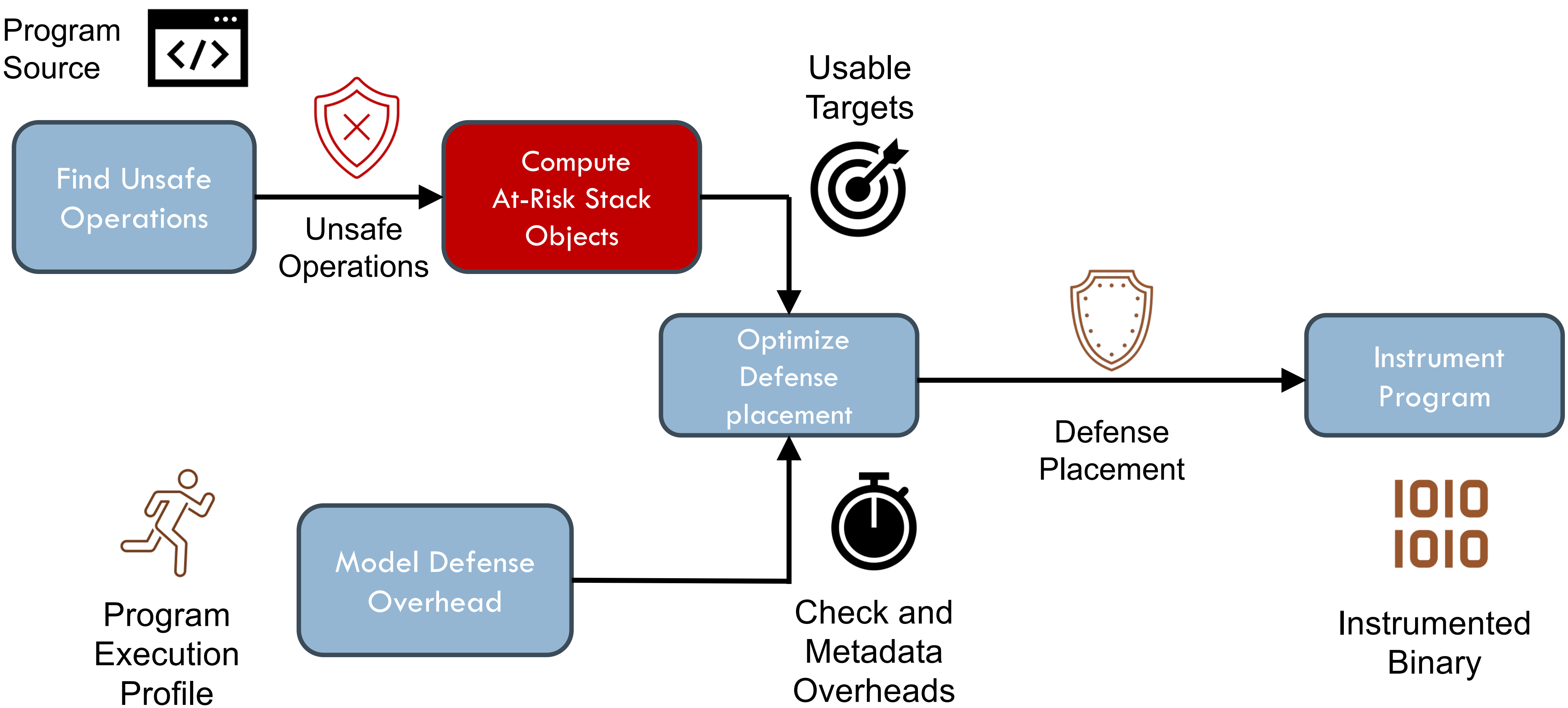
$$\max \sum_{i=1}^{m} g_i$$

- We develop a novel **mixed-integer non-linear program** (MINLP) formulation to maximize the protection of stack objects within a cost budget

$$\sum_{p=1}^{o} c_p \sum_{k=1}^{q} f_k \cdot y_{k,p} \leq B$$

$$\forall i \in \{1, \ldots, m\}, \quad g_i - \frac{\sum_{j \in T_i} \sum_{p=1}^{o} a_{j,p} \cdot x_{j,p}}{|T_i|} = 0$$

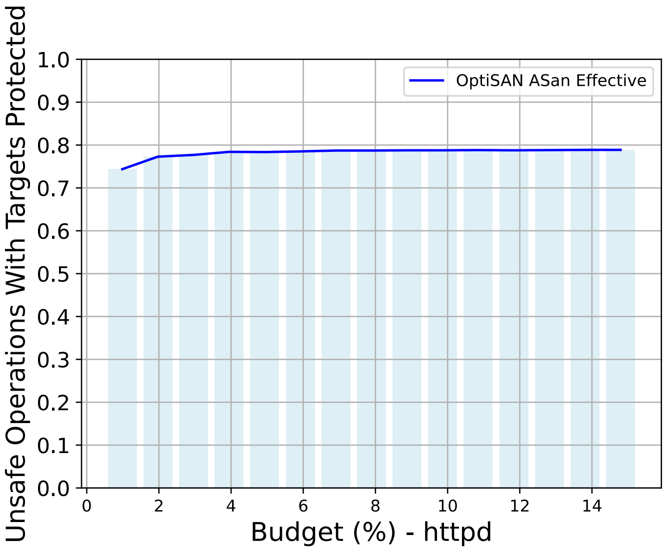
- We develop an **instrumentation pipeline** to instrument programs with the defenses at the **operation granularity** as computed

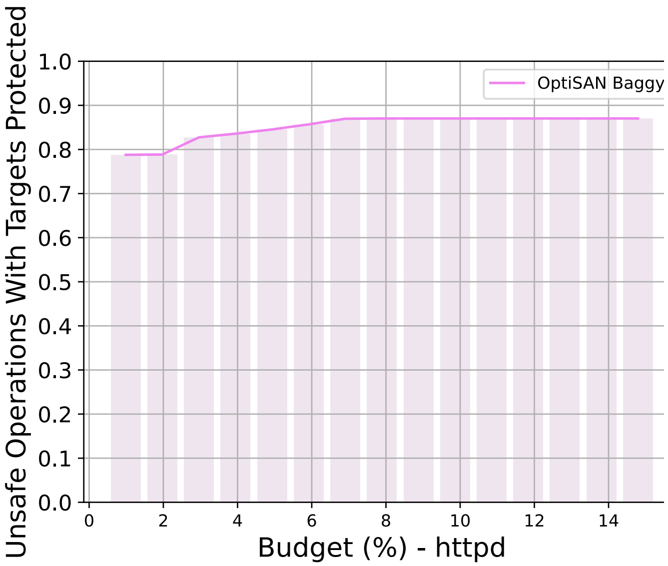

Program
Source
Find Unsafe
Operations
Unsafe
Operations
Compute
At-Risk Stack
Objects
Usable
Targets
Optimize
Defense
placement
Defense
Placement
Instrument
Program
Instrumented
Binary
Model Defense
Overhead
Check and
Metadata
Overheads
Program
Execution
Profile
1010
1010

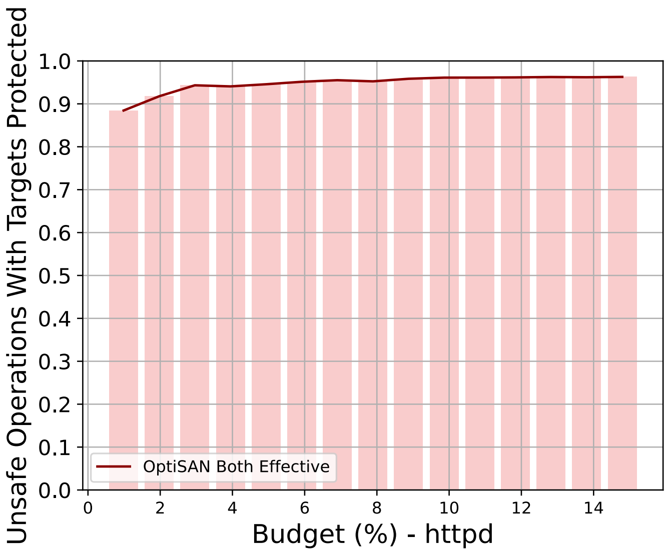
- Does applying ***multiple defenses improve security***?
  - Up to **52%** more unsafe operations covered than Baggy Bounds

- How are the ***defenses used together***?
  - Baggy Bounds is used **85%** on average

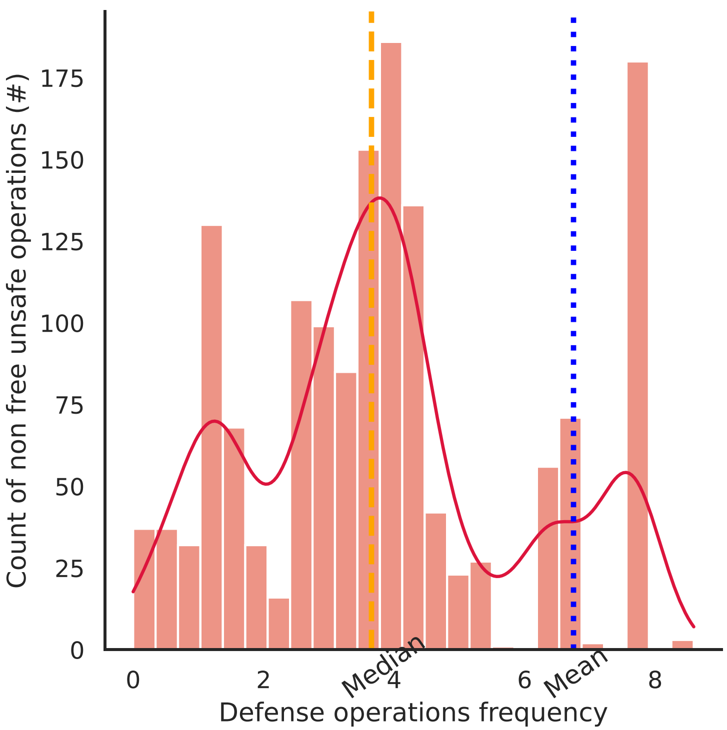
- Can OPTISAN produce an ***optimal placement*** with the ***desired overhead budget***?




Both defenses protect **18% more unsafe operations** on average than Baggy Bounds
For SPEC CPU 2006, 2017, httpd, openssl, redis, sqlite3




**Frequency Distribution - Apache**

| ***Program*** | ***ASan (%)*** | ***Baggy (%)*** | ***ASan is Slower (%)*** | ***Baggy Improves (%)*** |
|---|---|---|---|---|
| httpd | 13.15 | 19.66 | 2.43 | 16.76 |
| sqlite3 | 0.45 | 0.63 | 0.84 | 12.69 |
| sjeng | 33.08 | 53.08 | 5.45 | 12.10 |
| povray | 5.50 | 6.63 | 30.22 | 49.45 |
| gcc | 8.94 | 14.17 | 1.02 | 17.37 |
| perlbench | 10.75 | 20.95 | 3.05 | 2.33 |

The first two columns contain the performance overhead to protect all unsafe operations using ASan and Baggy Bounds, respectively. The third and fourth columns show the percentage of cases where ASan is slower and the percentage reduction in cost if Baggy Bounds is used instead.

**Performance Tradeoffs between ASan and Baggy Bounds**

# Questions