The Benefits of Performing Comprehensive Memory Safety Validation

Trent Jaeger, UC Riverside

Lead Student: Kaiming Huang
With Mathias Payer (EPFL), Zhiyun Qian (UCR), Jack Sampson (PSU), Gang Tan (PSU)

Problem of Memory Safety

- Unsafe programming languages (e.g., C/C++), distinguish memory objects from memory references (i.e., pointers).

- Allow pointers to reference any object and be changed independently
  - E.g., char buf[100]; char *p = buf; p++;

- This separation provides flexibility to programmers, but often leads to **memory errors**, when **pointer definitions become inconsistent with the objects to which they are intended to reference**.
  - E.g., p = p+200;

Vulnerabilities Due to Memory Errors

Despite being known since the 1970s, these memory errors are still common

- Google and Microsoft report independently that over 70% of their vulnerabilities are due to memory errors

```
int
im_vips2dz( IMAGE *in, const char *filename ){
  char *p, *q;
  char name[FILENAME_MAX];
  char mode[FILENAME_MAX];
  char buf[FILENAME_MAX];
  ...

  im_strncpy( name, filename, FILENAME_MAX );
  if( (p = strchr( name, ':' )) ){
    *p = '\0';
    im_strncpy( mode, p + 1, FILENAME_MAX );
  }

  strcpy( buf, mode );
  p = &buf[0];
  ...
}
```

Fig. 5: Case Study of CVE-2020-20739

Vulnerabilities Due to Memory Errors

Still, objects are not protected from illicit accesses due to memory errors

- Defenses aim to detect overwrites later (e.g., when the function returns) or make exploiting them harder, but there is a significant attack window

```
int
im_vips2dz( IMAGE *in, const char *filename ){
  char *p, *q;
  char name[FILENAME_MAX];
  char mode[FILENAME_MAX];
  char buf[FILENAME_MAX];
  ...

  im_strncpy( name, filename, FILENAME_MAX );
  if( (p = strchr( name, ':' )) ){
    *p = '\0';
    im_strncpy( mode, p + 1, FILENAME_MAX );
  }

  strcpy( buf, mode );
  p = &buf[0];
  ...
}
```

Fig. 5: Case Study of CVE-2020-20739

UC RIVERSIDE

Vulnerabilities Due to Memory Errors

Even for data that is never accessed unsafely by any of its aliases

- Even if no memory operation on `name` or `q` can possibly violate memory safety, they are at risk from unsafe accesses to other objects

```
int
im_vips2dz( IMAGE *in, const char *filename ){
  char *p, *q;
  char name[FILENAME_MAX];
  char mode[FILENAME_MAX];
  char buf[FILENAME_MAX];
  ...

  im_strncpy( name, filename, FILENAME_MAX );
  if( (p = strchr( name, ':' )) ){
    *p = '\0';
    im_strncpy( mode, p + 1, FILENAME_MAX );
  }

  strcpy( buf, mode );
  p = &buf[0];
  ...
}
```

Fig. 5: Case Study of CVE-2020-20739

Vulnerabilities Due to Memory Errors

This bothers me a lot

- ❑ Shouldn't we protect data whose accesses can be proven to be "safe" from memory errors?
- ❑ How much "safe" data do programs have?
- ❑ How hard is it to protect "safe" data from illicit access?
- ❑ How does identifying "safe" data impact the protection of "unsafe" data?

UC RIVERSIDE

So What?

Isn't C going to be replaced by memory safe languages (e.g., Rust)?

- Rust consists of a combination of **safe and unsafe code blocks**
- Even safe Rust code has runtime checks – incurs non-trivial overhead
  - Zhang et al., Towards Understanding the Runtime Performance of Rust, ASE 2022
- Unsafe Rust needs runtime checks – vulnerabilities may impact safe Rust
- **Fundamental question**: What is the optimal way to achieve comprehensive memory safety?

UC RIVERSIDE

Memory Error Classes

There are three classes of memory errors

- **Spatial errors** : pointer accesses to an object may be outside its memory region (bounds) – i.e., the one in the example

  Overwrite (overflow) and ***overread*** (disclosure)

- **Type errors**: pointer accesses to an object may use incompatible type semantics (e.g., interpret data as a pointer) – ***type confusion errors***
- **Temporal errors**: pointer accesses may occur before initialization (***use-before-initialization***) or after its referent is deallocated (***use-after-free***)

Insight (3-Cs)

Memory error defenses must balance along three dimensions to be effective

- All three **classes** of memory errors
- The **cost** of enforcing the defense
- The **coverage** of objects protected

Most research aims full coverage of objects using one defense for a subset of memory error classes – but costs are often too high for adoption

As a result, we are left with ad hoc and incomplete defenses in practice (e.g., canaries, ASLR, etc.)

UC RIVERSIDE

Is There Another Way?

Memory error defenses must balance along three dimensions to be effective

- All **classes** of memory errors
- The **cost** of enforcing the defense
- The **coverage** of objects protected

Identify objects that can be protected for all classes of memory errors for low cost

Then, explore how to combine defenses to address memory safety for unsafe operations

UC RIVERSIDE

Inspiration #1 – Memory Safety Validation

***CCured system* (Necula 2002) identifies the pointers whose uses cannot violate spatial and type safety**

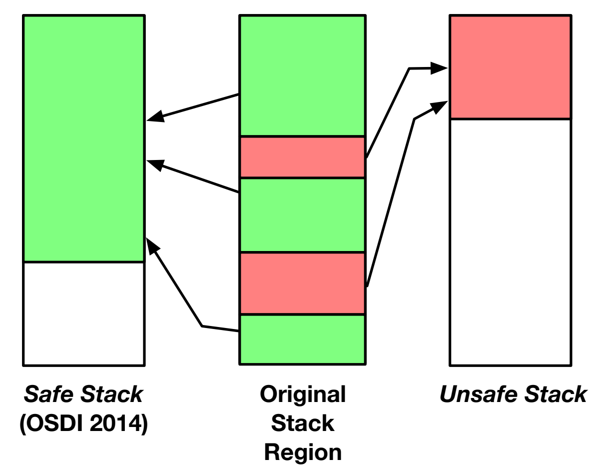
- A pointer cannot violate **spatial safety** unless it is used in **pointer arithmetic operation**
- A pointer cannot violate **type safety** unless it is used in a **type cast operation**
- They found about 90% of pointers are **never used in either operation**
- However, they did not address **temporal safety**

UC RIVERSIDE

Inspiration #2 – Multi-Stack/Heap

Separate objects with different memory safety properties into distinct stacks/heaps (e.g., Safe Stack)

- **Safe Stack** system separates objects referenced by **compiler-generated pointers** (safe) from address-taken objects (unsafe)
- Generally, protects **safe objects from spatial errors**, but protection from **type and temporal errors is incomplete**
- Some objects that **may have type and/or temporal errors** are still placed on the safe stack



UC RIVERSIDE

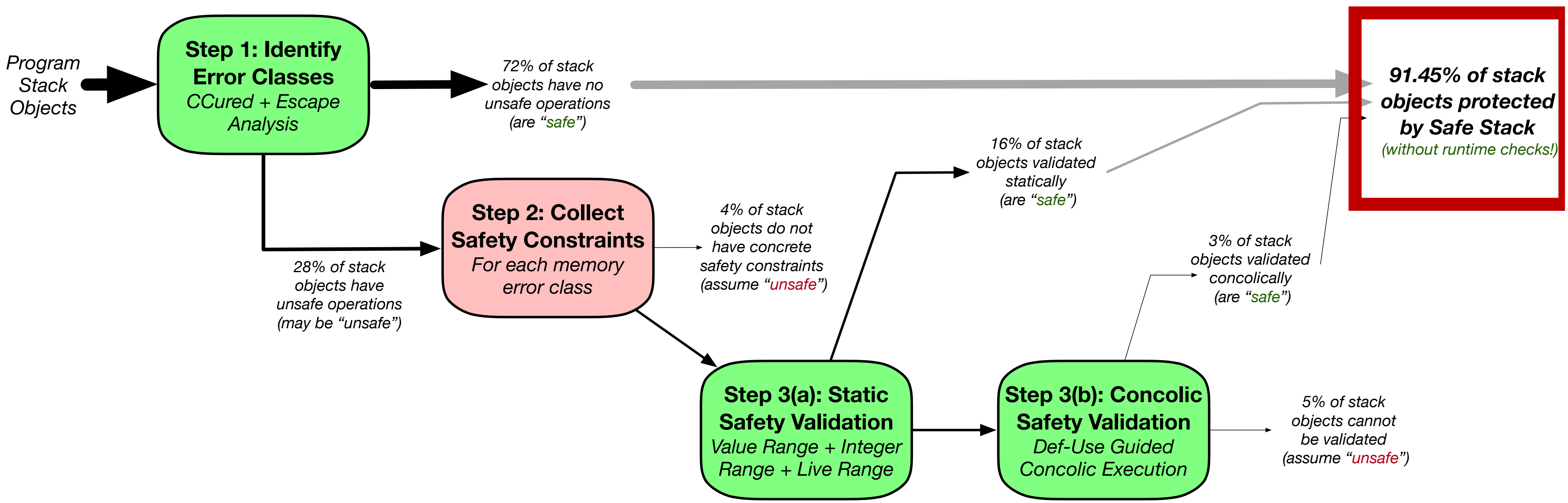
Hypotheses

It is possible to validate the memory objects whose accesses can never violate any of three classes of memory errors – memory safety validation

- A **large fraction of memory objects** whose accesses can be validated **statically** to satisfy memory safety (i.e., **are** "**safe**")
  - For both **stack** (all 3 classes) **and heap memory** (spatial and type safety, with a form of temporal safety enforced at runtime) regions
- **These objects can be protected** from memory errors in accesses to unsafe objects **cheaply**

Secondary Hypothesis: **Memory safety validation can provide insight into how to address memory safety enforcement for unsafe cases**

DataGuard – Comprehensive Memory Safety Validation for the Stack



UC RIVERSIDE

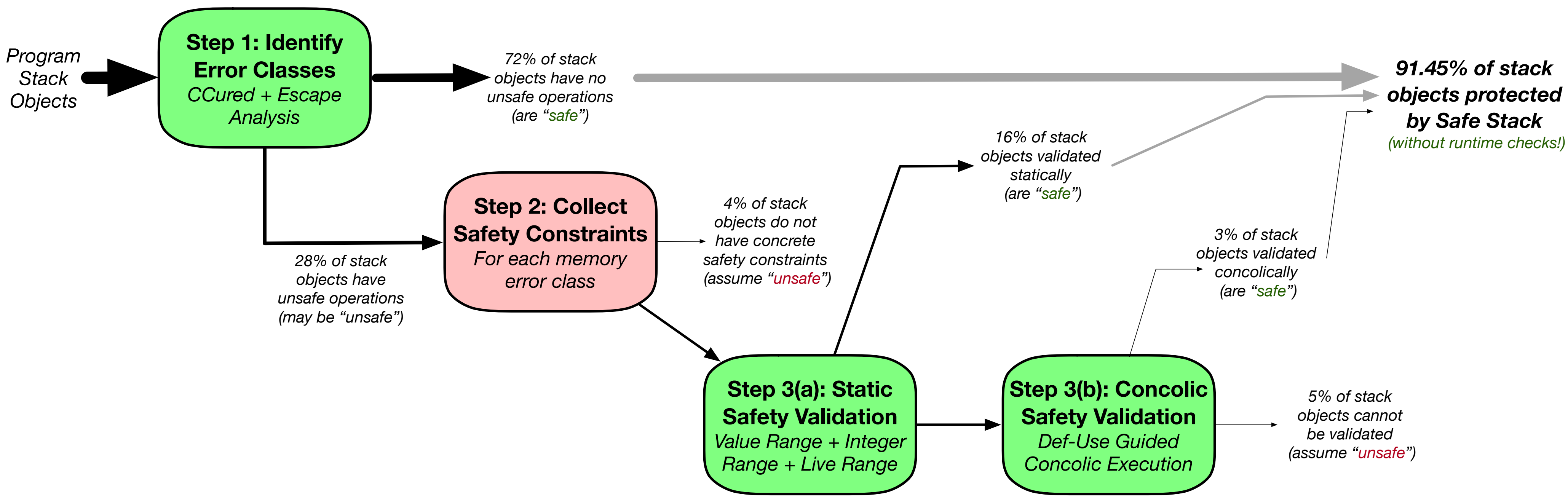
DataGuard Validation - Approach

A stack object is "safe" if all pointers that may-alias the object are only used in memory operations that must satisfy memory safety

- **Static analysis** to validate that all may-alias pointers are only used in safe operations relative to the **safety constraints**
  - **Spatial safety**: Concrete size and offsets – pointer's **value range** is in bounds
  - **Type safety**: For integers only, casts **must not change the integer's value**
  - **Temporal safety**: The def/use of all aliases are within its **live range**
- **Use directed concolic execution (along def-use chains found statically) to validate cases that are not provable statically**

UC RIVERSIDE

DataGuard – Comprehensive Memory Safety Validation for the Stack



UC RIVERSIDE

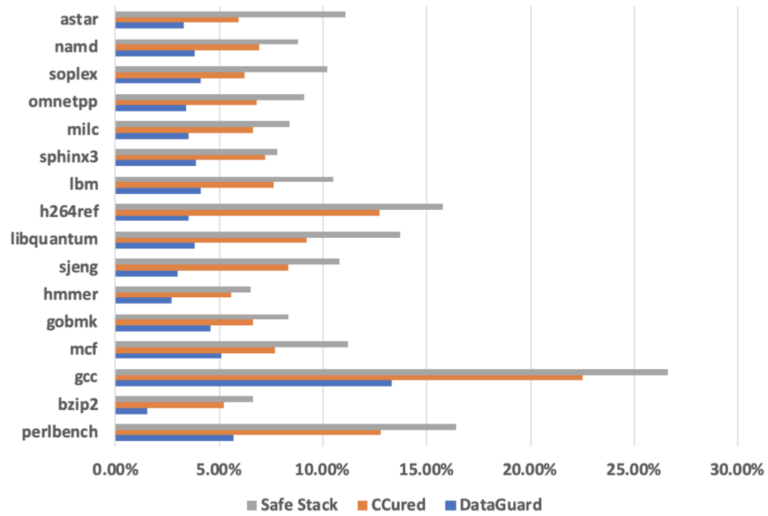
DataGuard Comparison

	CCured-default	CCured-min	Safe Stack-default	Safe Stack-min	DataGuard	Total
nginx	14,573 (79.52%)	14,496 (79.10%)	13,047 (71.20%)	12,375 (67.53%)	16,684 (91.05%)	18,324
httpd	61,915 (73.06%)	60,526 (71.42%)	49,523 (58.44%)	46,833 (55.27%)	78,266 (92.36%)	84,741
proftpd	14,521 (81.66%)	14,189 (79.79%)	12,837 (72.19%)	12.513 (70.37%)	16,190 (91.04%)	17,782
openvpn	48,379 (76.58%)	47,662 (75.45%)	40,627 (64.31%)	39,145 (61.97%)	57,693 (91.33%)	63,171
opensshd	20,238 (79.45%)	20,062 (78.75%)	18,176 (71.35%)	17,712 (69.53%)	23,871 (93.71%)	25,474
perlbench	52,738 (91.61%)	51,165 (88.57%)	42,398 (73.65%)	42,014 (72.98%)	52,324 (90.89%)	57,567
bzip2	1,293 (92.29%)	1,162 (82.94%)	1,057 (75.44%)	1,049 (74.87%)	1,238 (88.39%)	1,401
gcc	123,427 (73.34%)	120,856 (71.82%)	96,796 (57.52%)	91,344 (54.28%)	152,452 (90.59%)	168,283
mcf	580 (90.34%)	569 (88.63%)	441 (68.69%)	436 (67.91%)	602 (93.77%)	642
gobmk	34,376 (85.53%)	33,969 (84.52%)	26,229 (65.26%)	26,013 (64.72%)	38,552 (95.92%)	40,191
hmmer	20,133 (75.84%)	19,874 (74.87%)	13,873 (52.26%)	13,629 (51.34%)	25,674 (96.71%)	26,546
sjeng	3,461 (85.62%)	3,415 (84.49%)	2,798 (69.22%)	2,712 (67.10%)	3,741 (92.55%)	4,042
libquantum	2,576 (66.80%)	2,521 (65.38%)	2,036 (52.80%)	1,878 (48.70%)	3,214 (83.35%)	3,856
h264ref	19,525 (87.70%)	19,283 (86.61%)	14,418 (64.76%)	14,339 (64.40%)	20,177 (90.63%)	22,264
lbm	448 (82.96%)	442 (81.85%)	376 (69.63%)	369 (68.33%)	506 (93.70%)	540
sphinx3	2,744 (72.90%)	2,713 (72.10%)	2,058 (54.67%)	1,962 (52.13%)	3,398 (90.28%)	3,764
milc	4,325 (81.50%)	4,233 (79.76%)	3,887 (73.24%)	3,794 (71.49%)	4,680 (88.19%)	5,307
omnetpp	20,572 (83.44%)	20,264 (82.19%)	16,967 (68.82%)	16,283 (66.04%)	22,091 (89.60%)	24,655
soplex	14,253 (72.80%)	14,072 (71.87%)	11,044 (56.41%)	9,513 (50.12%)	16,368 (83.60%)	19,579
namd	21,676 (85.17%)	21,352 (83.90%)	18,389 (72.26%)	18,213 (78.34%)	23,249 (91.36%)	25,448
astar	4,016 (87.36%)	3,977 (86.51%)	3,606 (78.44%)	3,524 (76.66%)	4,206 (91.49%)	4,597

- 91.45% of stack objects are shown to be safe by DataGuard w.r.t. spatial, type, and temporal safety
- 79.54% and 64.48% of stack objects classified as safe by CCured and Safe Stack, respectively
- **50% and 70% unsafe stack objects** by CCured and Safe Stack, respectively, are **found safe** by DataGuard
- **3% and 6.3% safe stack objects** found by CCured and Safe Stack, respectively, are not provably safe in DataGuard

UC RIVERSIDE

DataGuard Performance



- **Runtime performance**: **4.3% for DataGuard**, 8.6% for CCured, 11.3% for Safe Stack.
  - All using the same Safe Stack defense implementation (based on ASLR)
- **DataGuard finds 76.12% of functions** have only safe stack objects
  - CCured and Safe Stack find 41.52% and 31.33%, respectively.

DataGuard – Broader Studies

Linux Ubuntu Package Study

	# of Packages	*# of SLOC*
Analyzed	1,245 (76.7%)	266,497,755 (77.8%)
Total	1,623	342,451,612

TABLE I: **Statistics of Linux Packages**

	Total	*DataGuard-Safe*
Object	14,627,355	12,484,971 (85.4%)
Control Data	451,839	412,725 (91.3%)
Function	1,152,744	747,391 (64.8%)
Parameter	1,904,262	1,622,867 (85.2%)

TABLE II: **Statistics of DataGuard Analysis on Linux Packages.**

Longitudinal Study

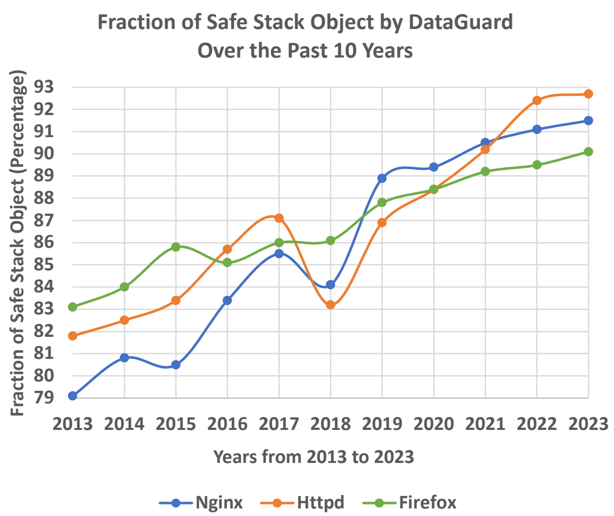
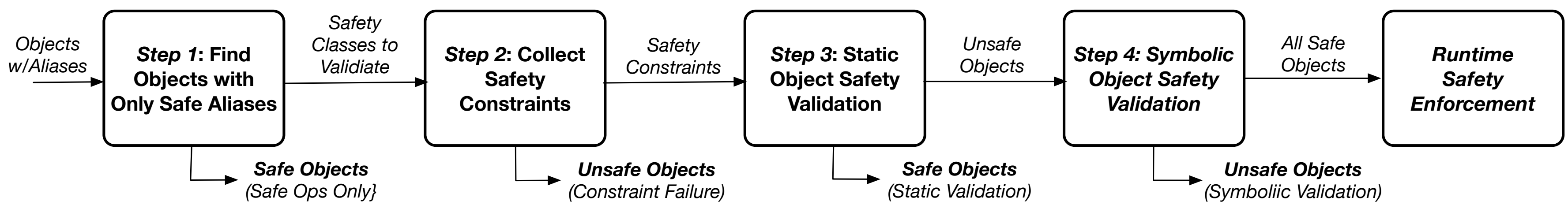


FIGURE 4. **Fraction of Safe Stack Objects by DataGuard**

Uriah – Using Memory Safety Validation for the Heap



10,000 Foot View is Similar

UC RIVERSIDE

Uriah Challenges

A heap object is "safe" if all pointers that may-alias the object are only used in memory operations that must satisfy spatial and type safety – enforce temporal safety

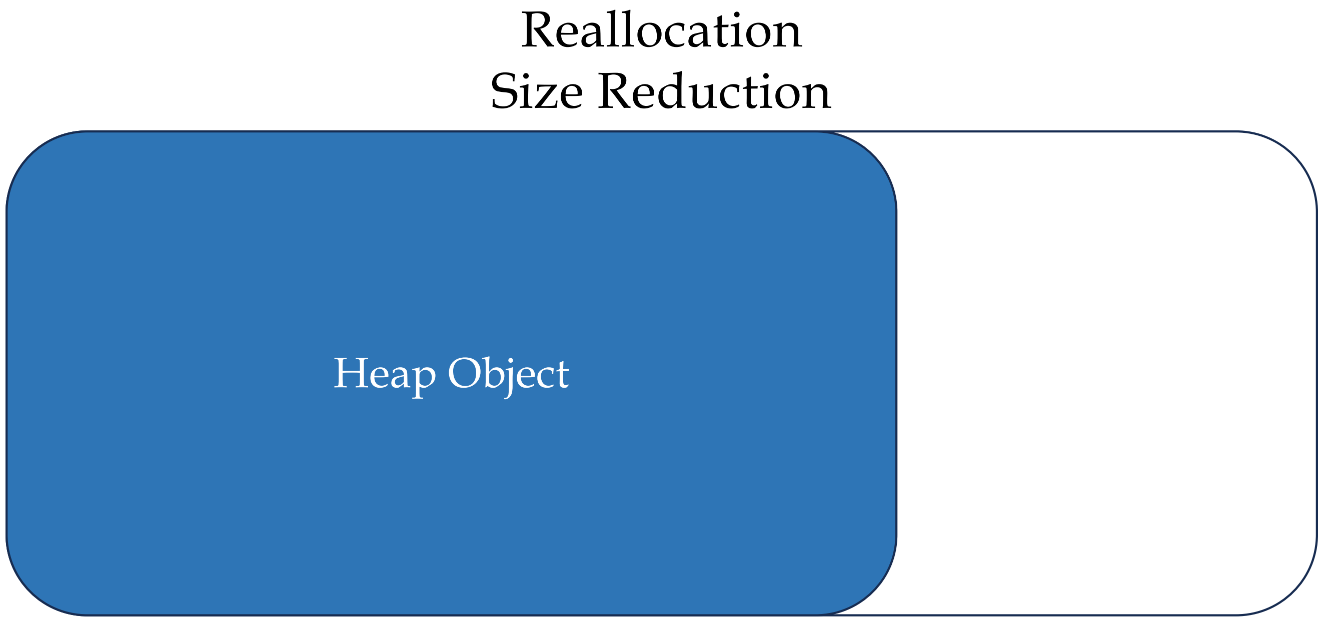
- **Static analysis** to validate heap objects must consider **several complexities**
  - **Reallocation**: Heap objects may be resized
  - **Threads**: Heap objects may be accessed by multiple threads
  - **Compound Types**: Heap objects are often complex, user-defined types
  - **Temporal**: No general algorithm to determine safety for heap objects
- **Aliasing: A significant fraction of false aliasing for heap objects could lead to many objects being falsely considered unsafe**

Reallocations

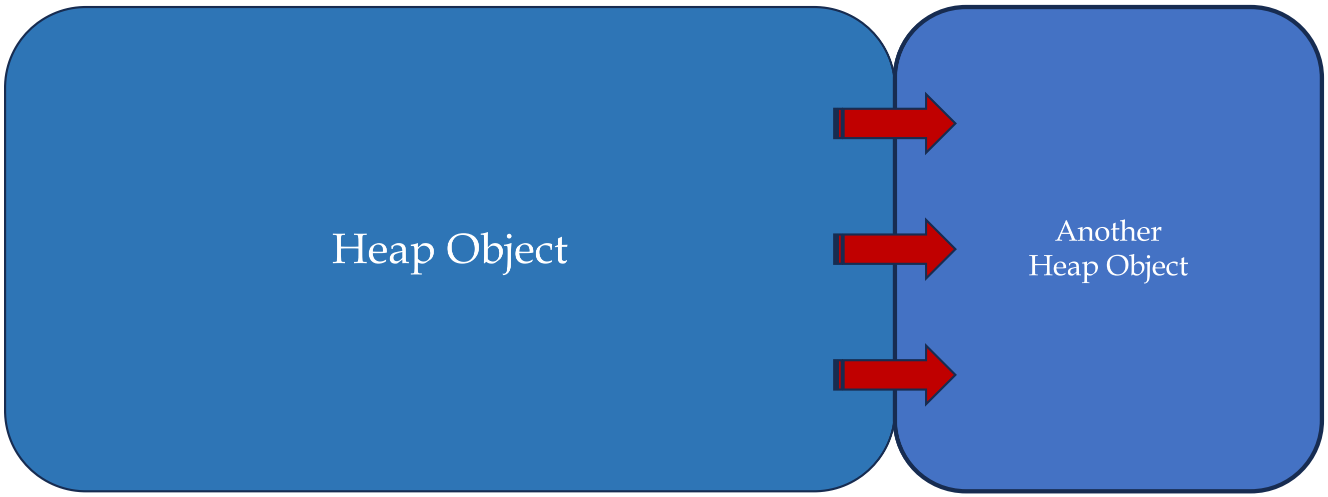
- **Spatial**: Either increase or decrease the size of the object
- **Type**: Change the format by changing the set of fields or their sizes
- **Temporal**: The object may be moved, leaving a dangling pointer

Reallocation

Consider Spatial Safety: For existing aliases

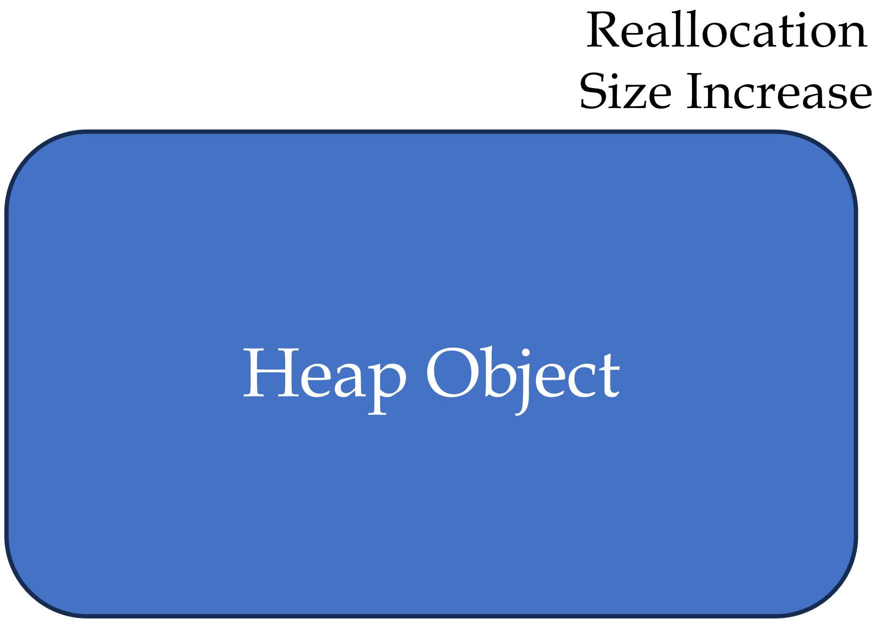

Reallocation
Size Reduction
Heap Object

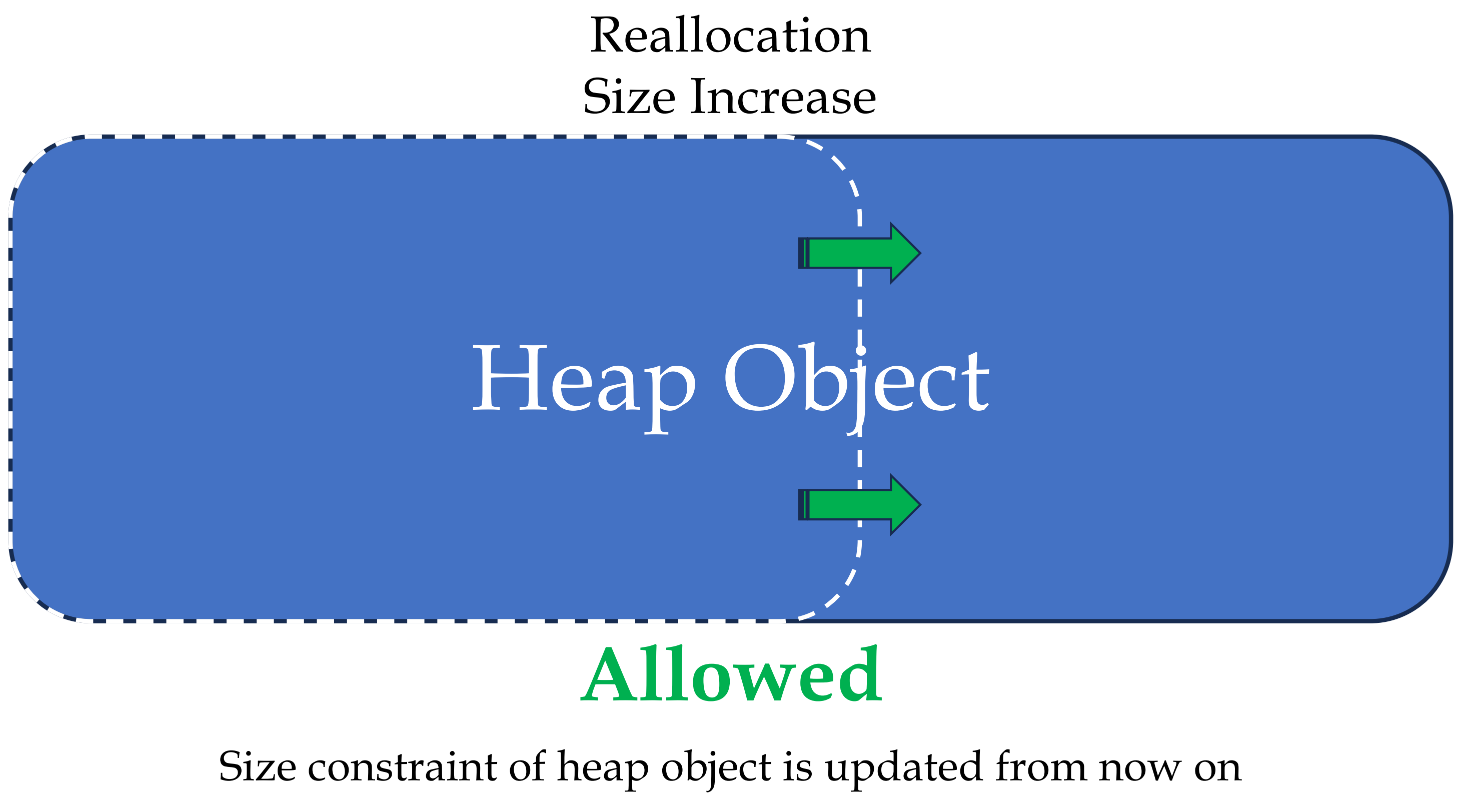
Reallocation
Size Reduction



Unsafe

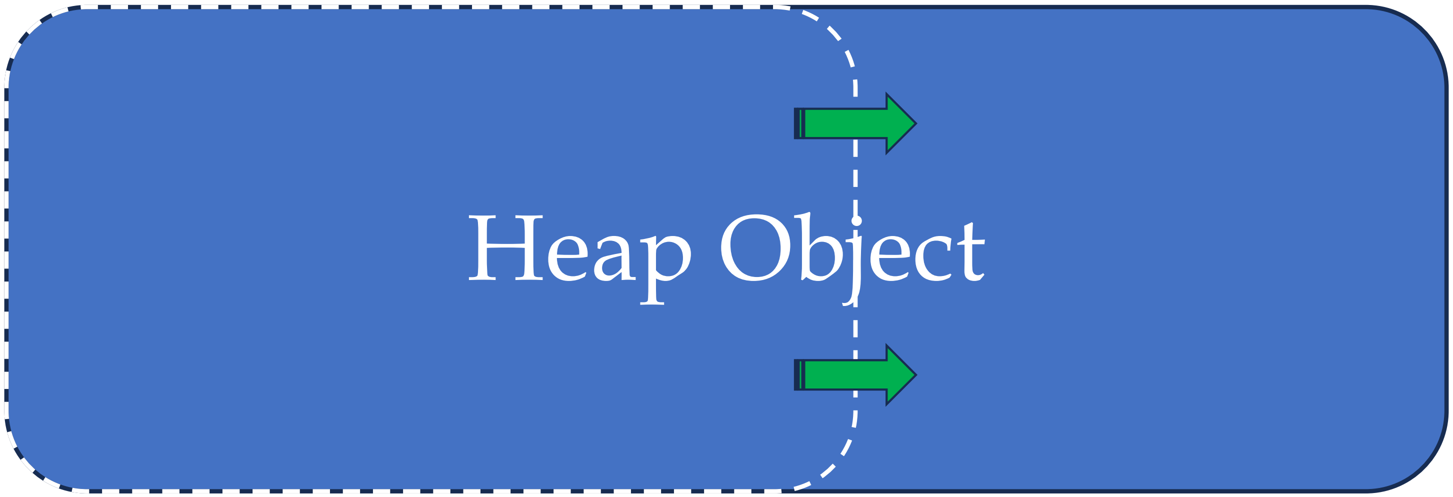
Cannot Guarantee all following operations obey the new size


Reallocation
Size Increase
Heap Object

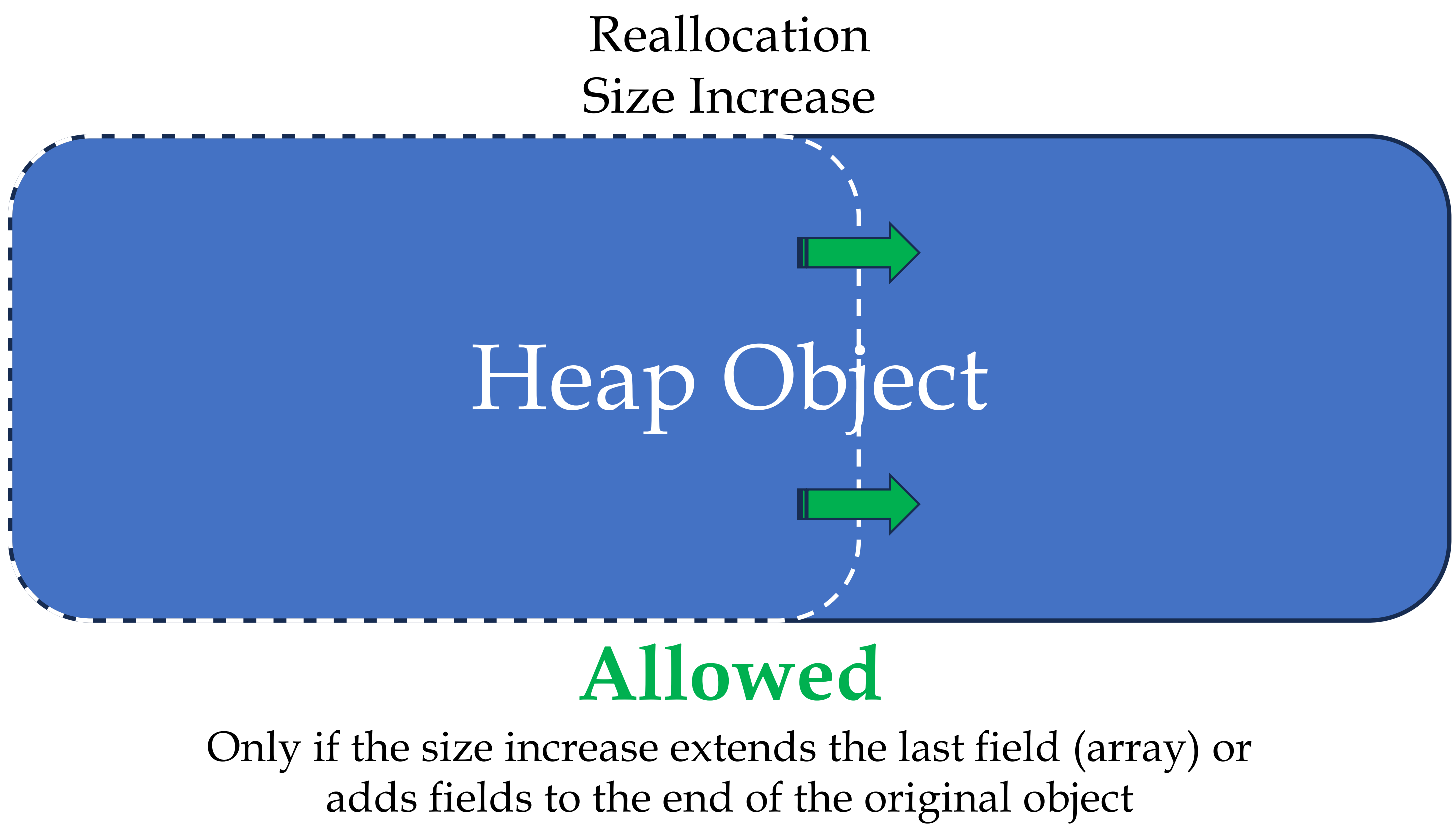

Reallocation
Size Increase
Heap Object
Allowed
Size constraint of heap object is updated from now on

Reallocation

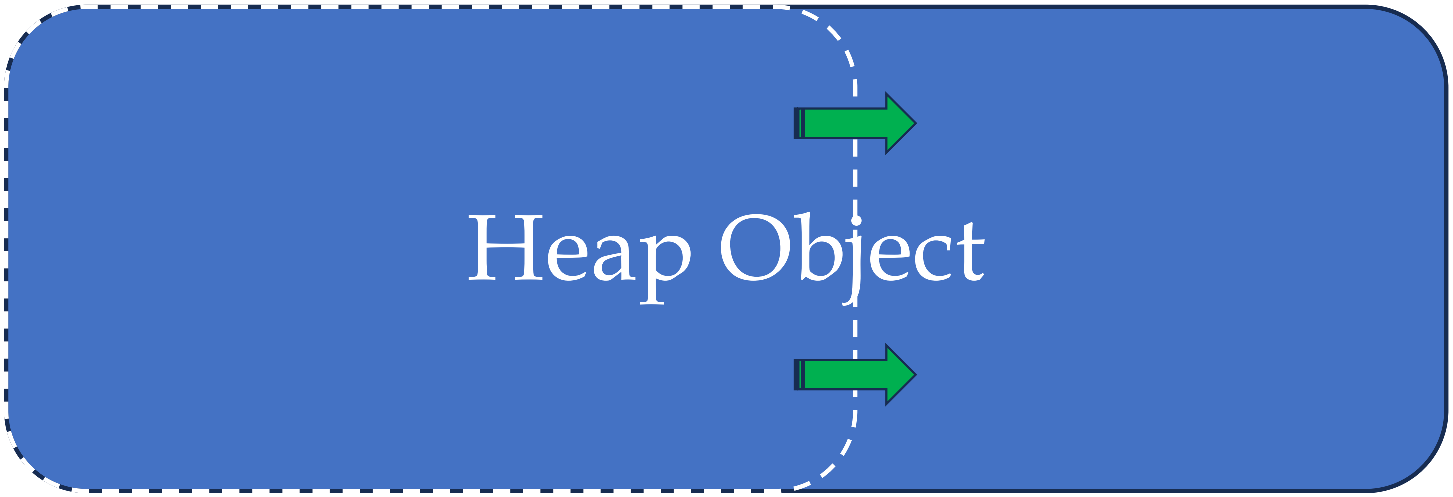
Consider Type Safety: For existing aliases



Does the extension of the object change its layout in memory?
In a way that would lead to unsafe accesses?


Reallocation
Size Increase
Heap Object
Allowed
Only if the size increase extends the last field (array) or
adds fields to the end of the original object

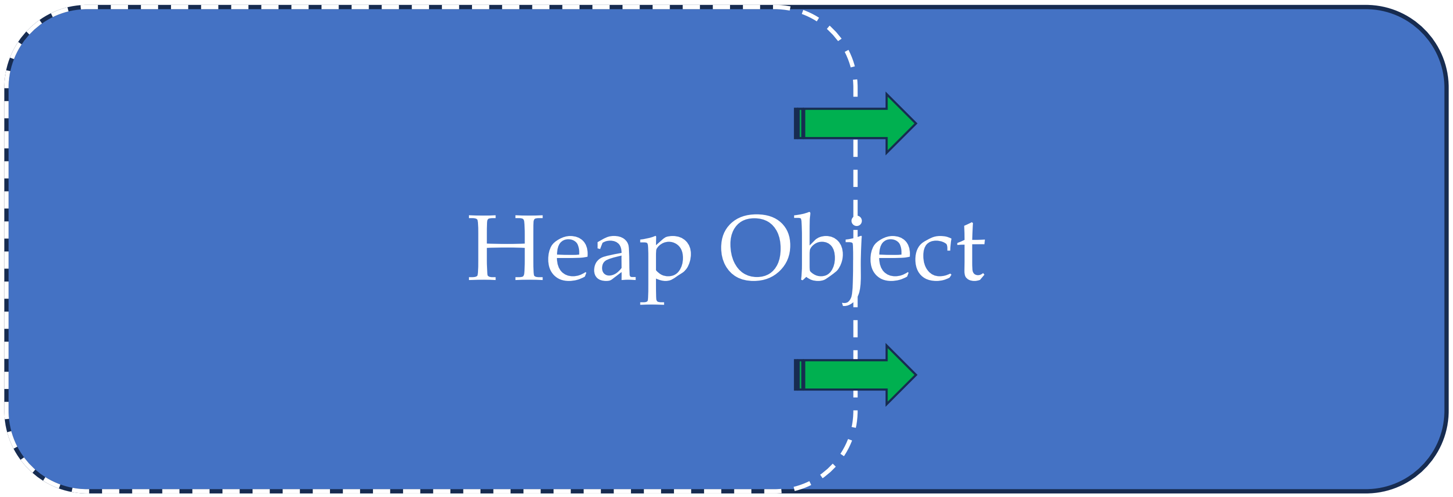
Reallocation
Consider Temporal Safety: For existing aliases



Reallocation will move the larger object to a new location
Leaving dangling pointers to the original location

Reallocation

Consider Temporal Safety: For existing aliases



Prior works restrict allocations to same size, but
Uriah enforces ***temporal allocated type safety*** – only objects of exactly the same size and format can be allocated in a location

Uriah Validation – Concolic Execution

Problem: Sound aliasing may produce many aliases that cannot actually point-to an object, and these aliases may be used in unsafe operations

- **Use directed concolic execution to invalidate infeasible unsafe aliases**
- **Infeasible definitions**: an object cannot be assigned (i.e., defined) to an alias on a path with an unsafe operation
- **Infeasible use**: an object cannot be used by an alias in an unsafe operation
- **Infeasible path**: the path cannot be executed in a manner that causes the unsafe operation

Uriah Comparison

	Total	*VR-Spatial*	*Uriah-Spatial*	*CCured-Type*	*CTCA-Type*	*Uriah-Type*	*VR-Spatial+ CCured-Type*	*Uriah-Spatial+ Uriah-Type*
Firefox	26,162	19,857 (75.9%)	20,432 (78.1%)	14,101 (53.9%)	19,700 (75.3%)	20,040 (76.6%)	12,270 (46.9%)	18,392 (70.3%)
nginx	954	705 (73.9%)	785 (82.3%)	585 (61.3%)	766 (82.3%)	819 (85.5%)	521 (54.6%)	744 (78.0%)
httpd	1,074	662 (61.6%)	816 (76.0%)	825 (76.8%)	918 (85.5%)	942 (87.7%)	575 (53.5%)	760 (70.8%)
proftpd	1,707	1,275 (74.7%)	1,380 (80.8%)	596 (34.9%)	1,201 (70.4%)	1,366 (80.0%)	458 (26.8%)	1,174 (68.8%)
sshd	378	270 (71.4%)	310 (82.0%)	170 (45.0%)	284 (75.1%)	304 (80.4%)	144 (38.1%)	274 (72.5%)
sqlite3	761	614 (80.7%)	655 (85.7%)	382 (50.2%)	567 (74.5%)	587 (77.1%)	316 (41.5%)	513 (67.4%)
perlbench	319	186 (58.3%)	241 (75.5%)	206 (64.6%)	258 (80.9%)	271 (85.0%)	154 (48.3%)	230 (72.1%)
bzip2	5	5 (100%)	5 (100%)	2 (40.0%)	4 (80.0%)	5 (100%)	2 (40.0%)	4 (80.0%)
mcf	4	4 (100%)	4 (100%)	0 (0.0%)	4 (100%)	4 (100%)	0 (0.0%)	4 (100%)
gobmk	29	19 (65.5%)	23 (79.3%)	10 (34.5%)	15 (51.7%)	19 (65.5%)	9 (31.0%)	16 (55.2%)
hmmer	350	238 (68.0%)	282 (80.6%)	73 (20.9%)	215 (61.4%)	256 (73.1%)	65 (18.6%)	240 (68.6%)
sjeng	12	10 (83.3%)	10 (83.3%)	3 (25.0%)	9 (75.0%)	9 (75.0%)	3 (25.0%)	9 (75.0%)
libquantum	19	13 (68.4%)	15 (78.9%)	7 (36.8%)	16 (84.2%)	16 (84.2%)	5 (26.3%)	14 (73.7%)
h264ref	103	76 (73.8%)	81 (78.6%)	29 (28.2%)	87 (84.5%)	87 (84.5%)	22 (21.4%)	75 (72.8%)
lbm	7	4 (57.1%)	5 (71.4%)	7 (100%)	7 (100%)	7 (100%)	4 (57.1%)	5 (71.4%)
sphinx3	138	66 (47.8%)	78 (56.5%)	59 (42.8%)	113 (81.9%)	120 (87.0%)	43 (31.2%)	70 (50.7%)
milc	55	41 (74.5%)	47 (85.5%)	8 (14.5%)	47 (85.5%)	49 (89.1%)	8 (14.5%)	45 (81.8%)
omnetpp	859	578 (67.3%)	600 (69.8%)	402 (46.8%)	713 (83.0%)	735 (85.6%)	342 (39.8%)	525 (61.2%)
soplex	242	165 (68.2%)	172 (71.1%)	137 (56.6%)	190 (78.5%)	202 (83.5%)	115 (47.5%)	161 (66.5%)
namd	29	22 (75.9%)	24 (82.8%)	7 (24.1%)	24 (82.8%)	24 (82.8%)	7 (24.1%)	24 (82.8%)
astar	48	28 (58.3%)	39 (81.2%)	15 (31.3%)	36 (75.0%)	38 (79.2%)	11 (23.0%)	34 (71.0%)
AVERAGE	—-	**71.7%**	**79.5%**	**42.3%**	**79.4%**	**83.9%**	**33.8%**	**71.9%**

- **71.9% of heap allocation sites** are validated by Uriah to only create safe objects w.r.t. spatial and type safety
- Correlates to **73.0% of allocated objects** for SPEC CPU2006 programs
- 33.8% of heap allocation sites are found safe for spatial and type safety by current best methods
- Extended TcMalloc to enforce temporal type safety **for 2.9% overhead** on SPEC CPU2006
  - Can isolate from unsafe accesses via SFI for <1% more.

DataGuard and Uriah – Broader Studies

Impact on Overhead for Unsafe

	SPEC CPU2006		SPEC CPU2017	
	Native	*w/* URIAH	*Native*	*w/* URIAH
TDI	8.4% / 15.5%	2.5% / 3.7%	12.5% / 18.6%	4.4% / 7.1%
CAMP	54.9% / 237.7%	16.8% / 72.3%	21.3% / 127.5%	8.2% / 40.6%

Table 8: Overhead Reduction of Applying TDI and CAMP to URIAH Unsafe Heap. Overhead is represented using the form "(runtime) / (memory)".

70% reduction in objects that need runtime protection leads to ~70% reduction in overhead

Uriah Longitudinal Study

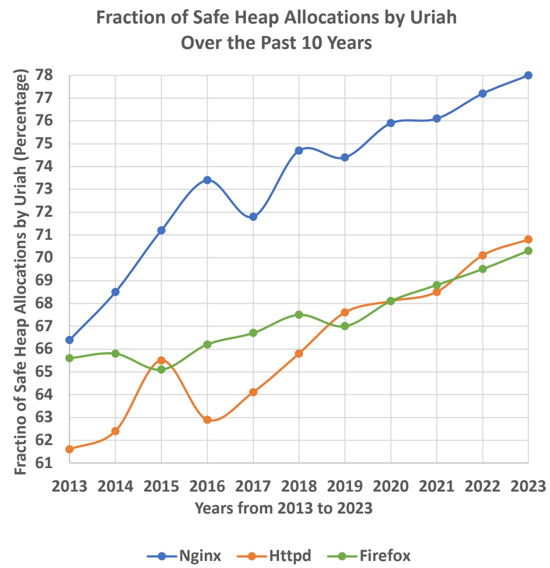


FIGURE 5. Fraction of Safe Heap Allocations by Uriah

The Future – How Can Memory Safety Validation Help?

UC RIVERSIDE

Leveraging Validation – Information Flow

Information Flow Validation

Information flow validation has long been used for programs to avoid inadvertent leaks

But could not detect **flaws like Heartbleed**, in C/C++ code

Since memory errors create data flows outside of program, current tools cannot be applied to C/C++

Leveraging Validation – Information Flow

Information Flow Validation for C/C++

But, if such a high fraction of objects are actually memory safe, **can we apply information flow usefully within this subset**?

Reconsider, Heartbleed: protect keys (safe objects) from unsafe accesses (Heartbleed bug) by construction and detect any Illegal information flows on safe

Leveraging Validation – Make C/C++ More Like Rust

Rust Memory Safety Is Explicit

Compare C/C++ to Rust, where some safety enforcement is done automatically (spatial checks via fat pointers) and some is required of programmers (temporal ownership) – but **unsafe code in Rust is explicitly identified**

Leveraging Validation – Make C/C++ More Like Rust

Memory Safety Validation

Can we make **memory safety** (safe/unsafe) **explicit in C/C++**, apply defenses automatically and efficiently?

UC RIVERSIDE

Leveraging Validation – Make C/C++ More Secure

OptiSan – Choose The Right Sanitizer

Location-based (red zones) and **identity-based** (fat pointers) sanitizers have different causes of overhead. We profile the program to determine which to apply at an operation (rather than object) granularity to maximize security within a budget. (USENIX 2024)

Can we choose the **most efficient defense combo**?

UC RIVERSIDE

Leveraging Validation – Make Rust Safer/Efficient

Memory Safety Validation

Can we **address unsafe operations in Rust**, adding memory safety checks (for safe and unsafe code) only where needed to apply defenses automatically and efficiently?

UC RIVERSIDE

Conclusions

Memory safety validation enables efficient protection of a large fraction of C/C++ program objects

- Foundation for protection from memory errors – safety is improving
- Quantify and make explicit which code is memory safe and reduce overhead for runtime defenses for unsafe code
- To improve defenses overall – e.g., enable checks for non-memory errors in C/C++ programs (information flow)

To improve our trust in computing

UC RIVERSIDE

Questions

- Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, Trent Jaeger. Top of the Heap: Efficient Memory Error Protection for Many Heap Objects. In *Proceedings of the 2024 ACM Conference on Computer and Communications Security* (ACM CCS), November 2024.
- Rahul George, Mingming Chen, Kaiming Huang, Zhiyun Qian, Thomas La Porta, Trent Jaeger. OptiSan: Using Multiple Spatial Error Defenses to Optimize Stack Memory Protection within a Budget. In *Proceedings of the 33rd USENIX Security Symposium*, August 2024
- Kaiming Huang, Mathias Payer, Zhiyun Qian, John Sampson, Gang Tan, Trent Jaeger. Comprehensive Memory Safety Validation: An Alternative Approach to Memory Safety. *IEEE Security & Privacy*, accepted for publication March 2024 for May/June 2024 issue.
- Kaiming Huang, Jack Sampson, Trent Jaeger. Assessing the Impact of Efficiently Protecting Ten Million Stack Objects from Memory Errors Comprehensively. In *Proceedings of the 2023 IEEE Secure Development Conference* (IEEE SecDev), October 2023.
- Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, Trent Jaeger. The Taming of the Stack: Isolating Stack Data from Memory Errors. In *Proceedings of the 2022 Network and Distributed System Security Symposium* (NDSS), April 2022.

UC RIVERSIDE