The Taming of the Stack: Isolating Stack Data from Memory Errors

Kaiming Huang, Yongzhe Huang, Mathias Payer,
Zhiyun Qian, Jack Sampson, Gang Tan, Trent Jaeger

EPFL, UC Riverside, Penn State University

Accepted by NDSS 2022

PennState

Vulnerability – Memory Errors

- Still many vulnerabilities being discovered due to memory errors
  - Most famously – **buffer overflows**
  - Recent example → → → → →
    - Bad coding style
    - Use of unsafe functions
- Allow adversary to write
  - Outside of allocated buffer (e.g., buf)
  - To write other objects (e.g., mode, *p)
  - **Other variants of memory errors**
- Extensive problem
  - "Eternal War in Memory"
  - "The Neverending Story: Memory Corruption 30 Years Later"
  - Take full control over the system
  - Defeat all protection schemes

- Case Study: CVE-2020-20739

```
1   int
2   im_vips2dz( IMAGE *in, const char *filename ){
3     char *p, *q;
4     char name[FILENAME_MAX];
5     char mode[FILENAME_MAX];
6     char buf[FILENAME_MAX];
7     ...
8
9     im_strncpy( name, filename, FILENAME_MAX );
10    if( (p = strchr( name, ':' )) ){
11      *p = '\0';
12      im_strncpy( mode, p + 1, FILENAME_MAX );
13    }
14
15    strcpy( buf, mode );
16    p = &buf[0];
17    ...
18  }
```

PennState

Security via Detecting Attacks

- But, detecting attacks is becoming more difficult
  - Adversaries are skilled and have systematic tools
  - Nation-state level attacks appear to be increasing

PennState

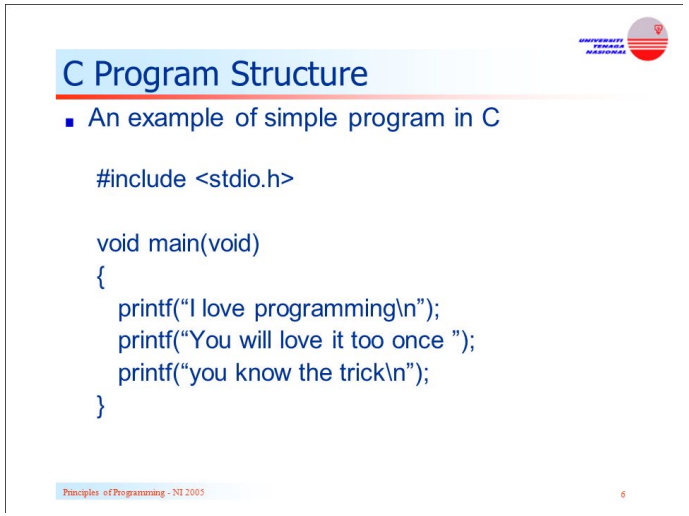
How did we get here?

- **Problem**: Systems often take risks (i.e., perform unsafe actions)
- *Internet*: enables parties worldwide to communicate
- *Firewalls*: must allow many unsafe communications
- *Access control*: cannot block any functional requirements
- *Software (part 1)*: use of unsafe languages leads to memory errors
- *Software (part 2)*: cannot validate information flows are safe in practice

PennState

Example: C programming language

- **Popular**: Still among the top-3 languages in preference in surveys
- **Lots of code**: Legacy code abounds
- **Useful**: Can write high performance code
- **Unsafe**: Makes no guarantees of memory safety



C Program Structure

- An example of simple program in C

```
#include <stdio.h>

void main(void)
{
   printf("I love programming\n");
   printf("You will love it too once ");
   printf("you know the trick\n");
}
```

Principles of Programming - NI 2005 6

PennState

What Can Go Wrong?

- What are the 3 general categories of memory error?

- What operations are needed for triggering each memory error class?

Spatial Error – Pointer Arithmetic

- Think about how to access an element in a string

```
char string[10];
string[3] = 'A';
```

- Here is what happens exactly

```
char string[10];
char *p;
p = string;
//string[3] = 'A';
p = p + 3;
*p = 'A';
```

Pointer Arithmetic

- Generally, there are 4 kinds of pointer arithmetic.
  - Increment/Decrement of a Pointer, e.g., `p++`
  - Addition of integer to a pointer, e.g., `p+3`
  - Subtraction of integer to a pointer, e,g., `p-5`
  - Subtracting two pointers of the same type, e.g., `offset=p1-p2`

What Can Go Wrong?

- Memory safety errors consist of three classes
  - **Spatial errors**: pointer accesses to an object may be outside its memory region (bounds) – unsafe pointer arithmetic operations.
  - **Type errors**: pointer accesses to an object may interpret the object using multiple types (casts) – unsafe type cast operations
  - **Temporal errors**: accesses to a pointer may occur before initialization (use-before-initialization or UBI) or after its target object is deallocated (use-after-free)

```
1  void example(int ct, char **buf) {
2     int lct = BUF_SIZE;
3     char lbuf[lct];
4     if (ct < lct){                  //(1) ct > buf's size
5       strlcpy(lbuf, *buf, (size_t) ct);     //(2) ct < 0
6     }
7     *buf = lbuf;                            //(3) temporal
8  }
```

Fig. 1: Example function demonstrates: (1) bounds error that enables overread of `buf`; (2) type error due to casting of `ct` from signed to unsigned; and (3) temporal error as `*buf` references local variable `lbuf` after return.

Reality

- **For memory safety in C**: Still only very limited protections, even just when considering stack objects
  - E.g., stack canaries to protect return addresses

- In this work, we explore an opportunity to leverage safety
- Are we almost able to protect most objects from memory errors?

PennState

Stack Is Security-Critical

Stack saves important data.

- Control data – e.g., flag variable in conditional branch, return address.
- Non-control data – e.g., user-sensitive data.

Stack suffers from variety kinds of attacks.

- Control flow hijacking – return address, function pointers.
- Data-oriented attack – Direct data manipulation (DDM), DOP.
- Block-oriented programming.
- 500+ CVEs related to stack memory errors in recent 3 years.

Stack-Based Memory Bugs Still Exist

- 500+ CVEs related to stack memory errors in recent 3 years.
- OOB writes: writes data out of the range of the intended buffer.
  - 2021-28972, 2021-24276, 2021-25178.
- OOB reads: disclose sensitive stack information.
  - 2021-3444, 2020-25624, 2020-16221.
- Type error: reference memory using different type semantics.
  - 2021-26825, 2020-15202, 2020-14147.
- Temporal error: reference memory using stale pointers.
  - 2020-25578, 2020-20739, 2020-13899.

Safe Stack Approach

Introduction

- **Protects code pointers against stack buffer overflows**
- **Multistack with two distinct, isolated regions**
  - **Safe Stack:** return addresses, function pointers, safe local variables
  - **Unsafe Stack:** everything else, e.g., buffers, address taken variables
- **Isolating Safe Stack from Unsafe Stack**
  - Ensure attack on unsafe stack object cannot corrupt Safe Stack.

Limitations

- **Security depends on classification of safe objects**
  - A safe stack object must not perform an operation that violates the security goal (i.e., no runtime checks on the safe stack)
- **Safe Stack classification is incomplete for memory errors**
  - Does not account for type errors and some temporal errors (e.g., UBI)
- **Safe Stack classification is conservative**
  - Some objects may be safe that are not placed on the safe stack

PennState

Example of Safe Stack

```
1  void example(int ct, char **buf) {
2      int lct = BUF_SIZE;
3      char safe_lbuf[lct];
4      char unsafe_lbuf[lct];
5      if (ct < lct){                                  //(1) ct > buf's size
6          strlcpy(unsafe_lbuf, *buf, (size_t) ct); //(2) ct < 0
7          ...                                         //Some safe operations on unsafe_lbuf
8          *buf = unsafe_lbuf;                         //(3) temporal
9      }
10     else{
11         strlcpy(safe_lbuf, *buf, lct-1);
12         ...                                         //Some safe operations on safe_lbuf
13         strcpy(*buf, safe_lbuf)
14     }
15 }
```

- What are unsafe objects for Safe Stack?
  - *buf, unsafe_lbuf, safe_lbuf.
- What are real unsafe objects?
  - *buf, unsafe_lbuf, ct.

PennState

Inspiration

- **For memory safety in C**: CCured system enables checking of which pointers are used only in memory-safe ways
  - For buffers that can be overflowed, there must be pointer arithmetic operations
  - For type confusion error, there must be type cast operations
  - **Safe**: No pointer arithmetic or casting operations
  - ***Results***: Estimated 90% of pointers are only used in safe operations
  - Are we almost able to protect most objects from memory errors?

PennState

Same Example Again…

```
1  void example(int ct, char **buf) {
2      int lct = BUF_SIZE;
3      char safe_lbuf[lct];
4      char unsafe_lbuf[lct];
5      if (ct < lct){                                  //(1) ct > buf's size
6          strlcpy(unsafe_lbuf, *buf, (size_t) ct); //(2) ct < 0
7          ...                                         //Some safe operations on unsafe_lbuf
8          *buf = unsafe_lbuf;                         //(3) temporal
9      }
10     else{
11         strlcpy(safe_lbuf, *buf, lct-1);
12         ...                                         //Some safe operations on safe_lbuf
13         strcpy(*buf, safe_lbuf)
14     }
15 }
```

- What are unsafe objects for CCured?
  - *buf, unsafe_lbuf, safe_lbuf, ct
- What are real unsafe objects?
  - *buf, unsafe_lbuf, ct.

PennState

Our Goals

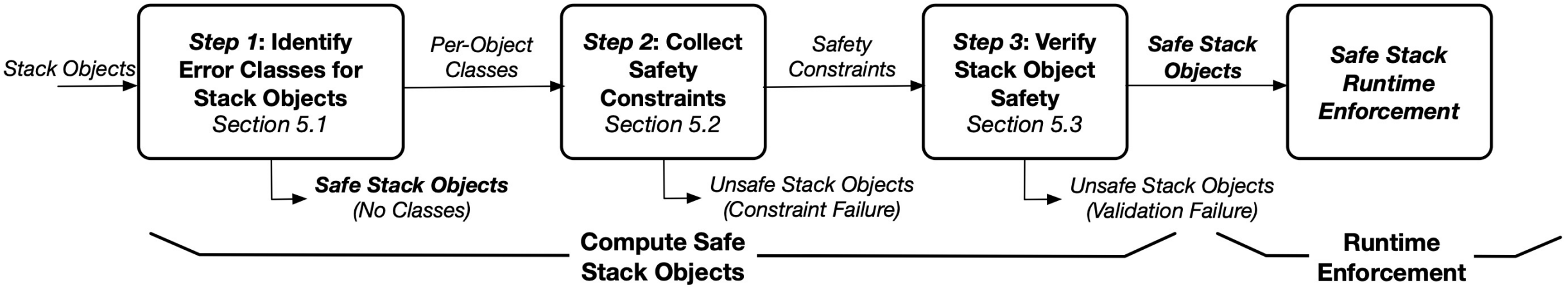
- Validate the safety of stack objects against all types of memory errors
  - Spatial, type, and temporal errors
  - Remove all unsafe objects from the safe stack
- Maximize number of stack objects found that are safe from memory errors comprehensively.
  - Add as many safe objects to the safe stack as feasible
- Ensure no unsafe stack object is ever mistakenly classified as safe
- Remove runtime checks on safe stack objects by isolating their accesses from unsafe objects
  - Same as the "safe stack" runtime defense
- **Protect more stack objects from memory errors comprehensively without runtime checks** – ultimately, leading to **better performance**

PennState

Design

PennState

Our Approach - DataGuard



PennState

Step 1 – Identifying Error Classes

- **Claim**: A stack object is safe unless it may be accessed by some pointer operation that may cause a memory error
  - May be cases where stack objects are trivially safe
  - Reduce validation effort to where needed
- **Question**: Which classes of memory errors may be possible for each stack object?

PennState

Step 1 – Identifying Error Classes

- Find the classes of memory errors possible for each stack object
  - Based on the operations performed using its pointers
- Could a pointer operation cause a spatial error?
  - **CCured**: if used in pointer arithmetic operation
- Could a pointer operation cause a type error?
  - **CCured**: if used in type cast operation
- Could a pointer operation cause a temporal error?
  - **CCured**: does not address
  - **Escape analysis**: if used prior to initialization or if escape via call/return or heap/global
- **Pointers that are not used in any such operations are "safe"**
- **Objects only aliased by safe pointers are "safe"**

Step 2 – Collecting Safety Constraints

- **Question**: For pointers used in unsafe operations, under what conditions are those operations safe?
- E.g., a pointer that is guaranteed to be used within bounds cannot cause a spatial memory error

PennState

Step 2 – Collecting Safety Constraints

- Safety Constraints
  - Spatial Constriants
  - Type Constraints
  - Temporal Constraints

• ***Declaration***: The *size* from the object's *base* must be declared as a constant value. The initial *index* is 0.

• ***Definition***: When a pointer is defined to reference the object, the reference may be *offset* to change the index. This offset must be a constant value.

• ***Use***: When a pointer is used in an operation, the pointer may be further offset to change the index. Each offset in a use must also be a constant value.

• ***Validation***: For all uses, pointer $index < size$ and $index \geq 0$

Step 2 – Collecting Safety Constraints

- Define safety validation requirements for each memory error class
  - Spatial, type, and temporal
- Collect constraints for each stack object
  - E.g., Stack object size must be declared as a constant
  - Constraints may not be found for all stack objects
- Collect constraints for each pointer
  - E.g., All pointer arithmetic operations must use constants
  - Define constraints for pointer "definitions" and "uses"
- **If safety constraints cannot be derived for a stack object or any pointers that may alias it, then safety validation is not possible and the object is "unsafe"**

Step 3 - Verifying Stack Object Safety

- **Question**: w/o running program, can we tell if all executions satisfy safety constraints?
  - Static analysis and Symbolic execution
  - Prior work (Baggy Bounds) applied value-range analysis to reduce the number of pointer operations that would require bounds checks
- **Problem**: Static analyses that over-approximate program executions may find a pointer ”unsafe” that could really be used on safely
- **Hypothesis**: A significant number of such cases exist (due to aliasing), so many stack objects may be found “unsafe” that can be proven “safe”

```
1  char str[30];
2  int v1 = read_int();
3  int v2, v3, v4 = 0;
4  if (v1 > 10){
5      v2 = 10; //v2:[10,10]
6      v3 = 20; //v3:[20,20]
7  }
8  else{
9      v2 = 15; //v2:[15,15]
10     v3 = 15; //v3:[15,15]
11 }
12 v4 = v2+v3-1; //v4:[24,34]
13 read(0, str, v4);
```

PennState

Step 3 - Verifying Stack Object Safety

- Apply the safety constraints to determine whether we can prove a stack object is only accessed via safe pointer operations
- Performed in **two steps**:
  - **(1) Static analysis**: Find all pointers that may-alias the stack object can only perform operations that comply with the safety constraints
    - **Spatial**: Value-range analysis
    - **Type**: Value-range analysis to validate that integer type casts never change value
    - **Temporal**: Live-range analysis to find that def and use of all aliases are within the live range
- **A stack object is "safe" if all pointers that may-alias it are safe**
  - I.e., all may-alias pointers are only used in safe operations relative to the safety constraints

Step 3 - Verifying Stack Object Safety

- Apply the safety constraints to determine whether we can prove a stack object is only accessed via safe pointer operations
- Performed in **two steps**:
  - **(2) Concolic execution**: Determine whether a complete execution of all operations that access the stack object only consists of safe operations
    - Only performed for stack objects with any aliases found to be "unsafe" from the static analysis
  - **Problem**: Path explosion of symbolic execution
    - Utilize def-use chain already computed to guide symbolic execution
    - Perform a limited symbolic execution for all stack objects not found safe via static analysis (e.g., limit the context depth)
- **A stack object is "safe" if all operations that access it comply with safety constraints**
  - If a complete symbolic execution cannot be performed, the object is "unsafe"

Soundness

- Must ensure that no stack object ever used in an unsafe operation may be classified as "safe"
  - Our analyses must overapproximate the program executions (i.e., be "sound")
- **Challenge**: DataGuard leverages a variety of static analyses [1,2]
  - Some claim soundness, some prove soundness
- We show that DataGuard achieves *relative soundness*
  - DataGuard's analysis is sound if all utilized analyses are sound
- By default, SE is a sound form of analysis because it follows all execution paths of a program [4]
  - However, for performance, SE analyses often make choices that render it unsound
  - DataGuard avoids such choices in SE, limiting [3]

[1] SVF: interprocedural static value-flow analysis in LLVM. Y. Sui et al, CC '06
[2] PtrSplit: Supporting general pointers in automatic program partitioning. S. Liu et al, CCS '17
[3] S2E: A Platform for In-vivo Multi-path Analysis of Software Systems, V. Chipounov et al, ASPLOS '11
[4] A Survey of Symbolic Execution Techniques. R. Baldoni et al, 2018.

Evaluation

PennState

How Does DataGuard Impact the Security of Safe Stack Object Compared with Previous Work?

	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 soundly by DataGuard w.r.t. spatial, type and temporal errors.
- 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 are found safe by DataGuard.
- 3% and 6.3% safe stack objects by CCured and Safe Stack are found unsafe by DataGuard.

PennState

How Frequently are Pointers Used in Unsafe Operations for Each Error Class?

	Total	*Spatial*	*Type*	*Temporal*	*Safe*
nginx	11,679	1,555 (13.31%)	555 (4.75%)	1,401 (11.99%)	8,785 (75.22%)
httpd	58,572	12,116 (20.69%)	2,905 (4.96%)	16,232 (27.71%)	37,899 (64.70%)
proftpd	10,354	1,332 (12.86%)	488 (4.71%)	1,156 (11.16%)	8,155 (78.76%)
openvpn	38,065	7,061 (18.55%)	2,326 (6.11%)	8,734 (22.93%)	26,020 (68.36%)
opensshd	15,067	2,185 (14.50%)	479 (3.18%)	1,924 (12.77%)	11,798 (78.30%)
perlbench	33,241	2,255 (6.78%)	454 (1.37%)	5,571(16.76%)	27,345 (82.30%)
bzip2	778	52 (6.68%)	9 (1.16%)	146 (18.76%)	616 (79.17%)
gcc	103,285	22,661 (21.94%)	6,012 (5.82%)	19,476 (18.85%)	69,863 (67.64%)
mcf	384	28 (7.29%)	7 (1.82%)	57 (14.84%)	303 (78.90%)
gobmk	22,363	2,959 (13.23%)	170 (0.76%)	5,302 (23.71%)	15,522 (69.40%)
hmmer	16,257	3,759 (23.12%)	203 (1.25%)	2,803 (17.24%)	11,126 (68.43%)
sjeng	2,449	348 (14.20%)	74 (3.02%)	420 (17.14%)	1,768 (72.19%)
libquantum	2,182	524 (24.01%)	162 (7.42%)	343 (15.72%)	1,387 (63.57%)
h264ref	13,246	1,535 (11.59%)	91 (0.69%)	2,192 (16.55%)	10,109 (76.32%)
lbm	307	35 (11.40%)	8 (2.61%)	56 (18.24%)	226 (73.62%)
sphinx3	2,143	478 (22.30%)	135 (6.30%)	509 (23.75%)	1,320 (61.60%)
milc	2,943	338 (11.48%)	117 (3.98%)	314 (10.67%)	2,326 (79.03%)
omnetpp	13,780	1,247 (9.05%)	848 (6.15%)	1,832 (13.29%)	10,636 (77.18%)
soplex	11,941	1,910 (16.00%)	1,482 (12.41%)	2,453 (20.54%)	7,107 (59.51%)
namd	14,026	1,780 (12.69%)	154 (1.10%)	2,325 (16.58%)	10,852 (77.37%)
astar	2,571	193 (7.51%)	71 (2.76%)	414 (16.10%)	1,925 (74.87%)

- 14.24% spatial, 3.92% type, 17.39% temporal.
- 72.70% of stack pointers are free from any class of memory errors.

PennState

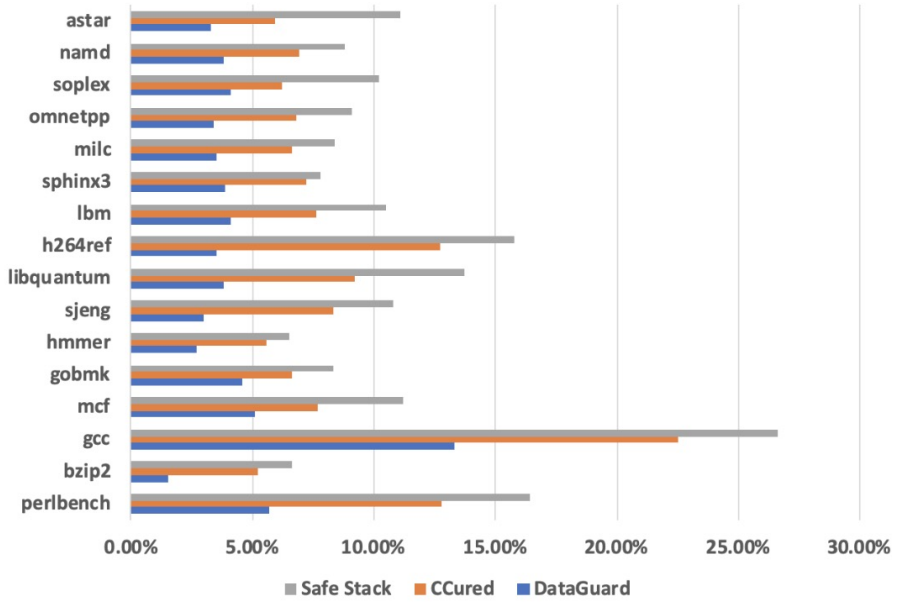
How Much Does the Two-Stage Validation Improve the Ability to Identify Safe Stack Objects over Prior Work?

	Safe Pointer	*Diff. from CCured*
CCured	10,124 (86.68%)	0 (00.00%)
Symbolic Exec (SE)	10,501 (89.91%)	377 (24.24%)
Value Range	11,085 (94.91%)	961 (61.80%)
Value Range+SE	11,498 (98.45%)	1,374 (88.36%)

	Safe Pointer	*Safe Address-Taken*
Safe Stack	10,278 (88.00%)	0 (00.00%)
Error Class (EC)	11,244 (96.27%)	966 (68.95%)
Liveness (LV) + EC	11,463 (98.15%)	1,185 (84.58%)
SE+LV+EC	11,586 (99.20%)	1,308 (93.36%)

- For Nginx
- 88.36% of pointers classified as unsafe for spatial errors by CCured are found as safe by DataGuard.
  - DataGuard's use of "Value range+SE" finds more (413) safe pointers than "SE alone" (377).
- 93.36% of pointers classified as unsafe for temporal errors by Safe Stack are found as safe by DataGuard.

How Does the Increase in Safe Stack Objects Impact Performance?



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

PennState

Does DataGuard Enhance the Security of Programs and Prevents Real-World Exploits?

- Attack Mitigation
  - Exploit objects are classified as unsafe.
  - Target object are classified as safe.
- CGC Binaries
  - 87 binaries have stack-related memory bugs.
  - Directly mitigates 95 of 118 exploits.
  - Successfully classifies all targets objects of the steppingstone objects for the remaining 23.
- Impact on Control Data

	Control Data	*Safe-Stack-Safe*	*DataGuard-Safe*
nginx	1,023	632 (61.78%)	946 (92.47%)
httpd	2,276	1,431 (62.87%)	2,108 (92.62%)
proftpd	1,214	576 (47.45%)	1,128 (92.92%)
openvpn	3,482	1,965 (56.43%)	3,289 (94.46%)
opensshd	1,458	862 (59.12%)	1,326 (90.95%)

  - 92.68% of control data on stack are safe.
  - Much more than Safe Stack approach.

- Case Study: CVE-2020-20739

```
1   int
2   im_vips2dz( IMAGE *in, const char *filename ){
3     char *p, *q;
4     char name[FILENAME_MAX];
5     char mode[FILENAME_MAX];
6     char buf[FILENAME_MAX];
7     ...
8
9     im_strncpy( name, filename, FILENAME_MAX );
10    if( (p = strchr( name, ':' )) ){
11      *p = '\0';
12      im_strncpy( mode, p + 1, FILENAME_MAX );
13    }
14
15    strcpy( buf, mode );
16    p = &buf[0];
17    ...
18  }
```

PennState

Conclusion

- Hypothesis
  - We can improve security enforcement if we focus on validating safety accurately
- **DataGuard**
  - Validated >90% of stack objects are from safe spatial, type and temporal errors
  - More complete definition of memory safety than prior work, improving security
  - More accurate analysis finds as safe 70% of the objects classified as unsafe by Safe Stack
  - Average overhead reduced from 11.3% to 4.3% for SPEC 2006 benchmarks.
  - Applicable to real-world programs and prevents real exploits.
  - Will be available open source soon
- DataGuard shows that a ***comprehensive*** and ***accurate*** analysis can both increase the scope of stack data protection and reduce overheads.
  - Safety validation gets us more security for lower cost!

PennState