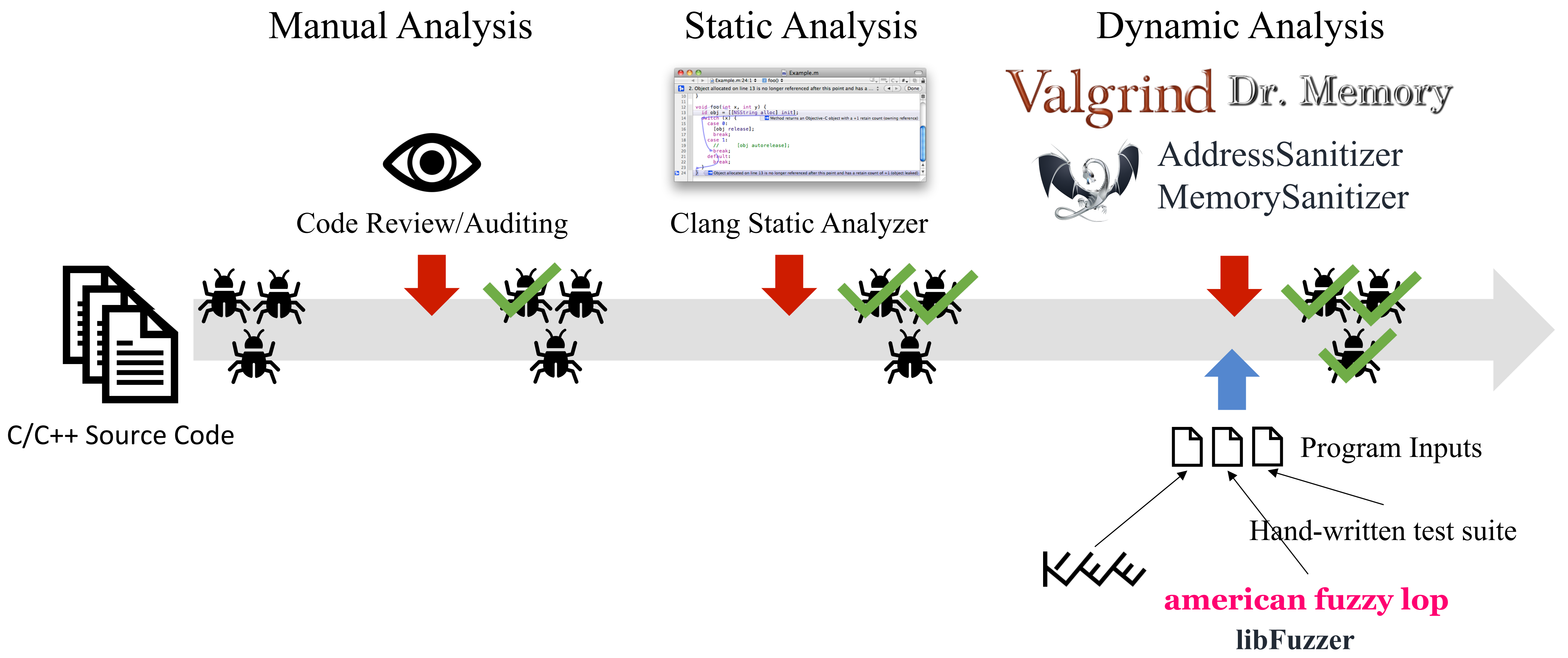
SoK: Sanitizing for Security

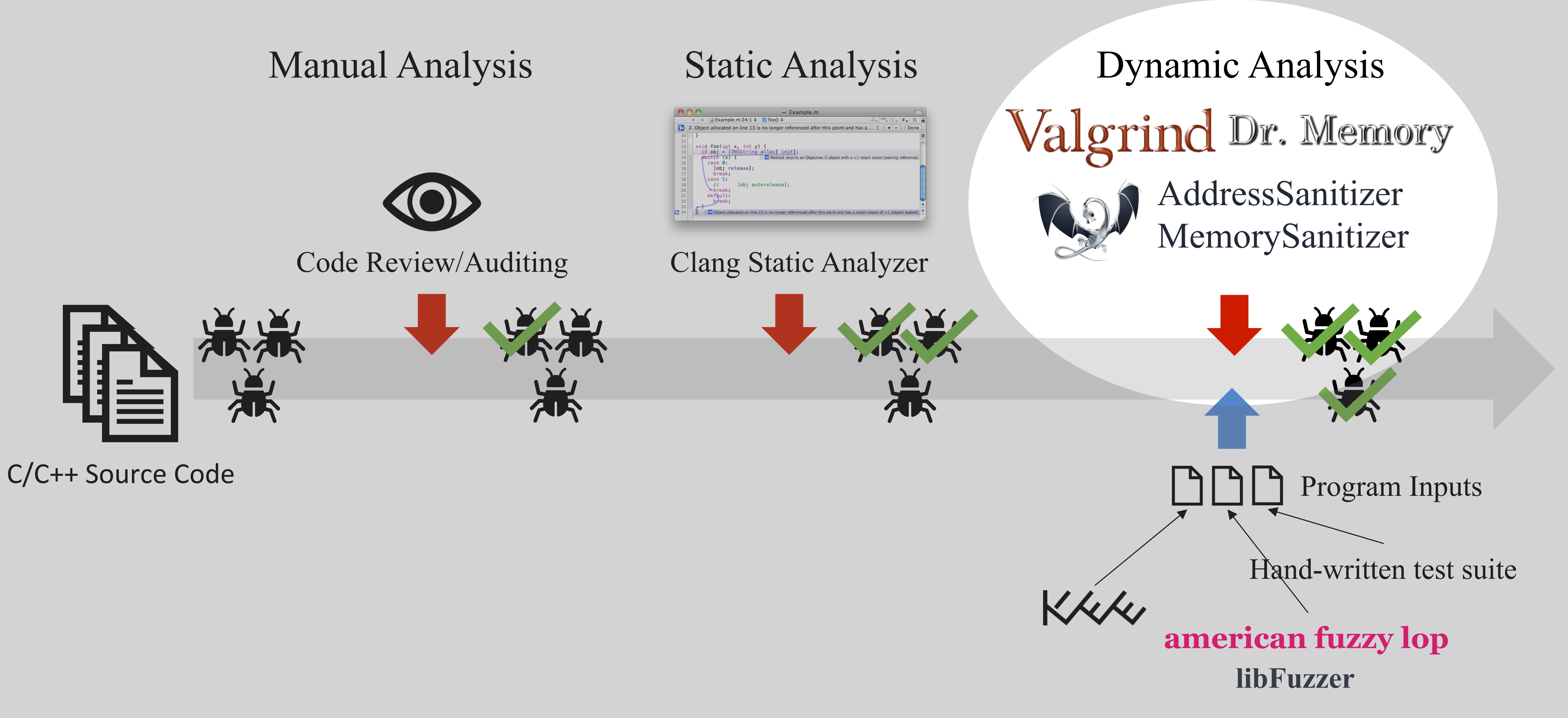
Dokyung Song, Julian Lettner, Prabhu Rajasekaran,
Yeoul Na, Stijn Volckaert, Per Larsen, Michael Franz

University of California, Irvine

Finding Bugs in C/C++

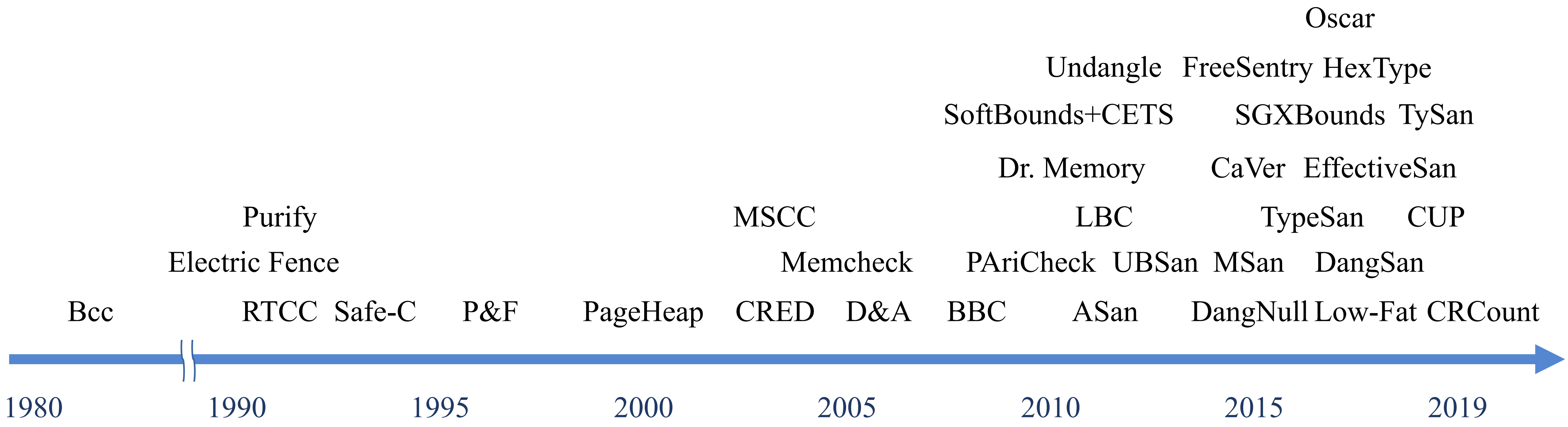


Finding Bugs in C/C++



Dynamic Analysis Tools for C/C++

- More than 35 years of research in Dynamic Analysis Tools – often-called "*Sanitizers*" – that find vulnerabilities specific to C/C++

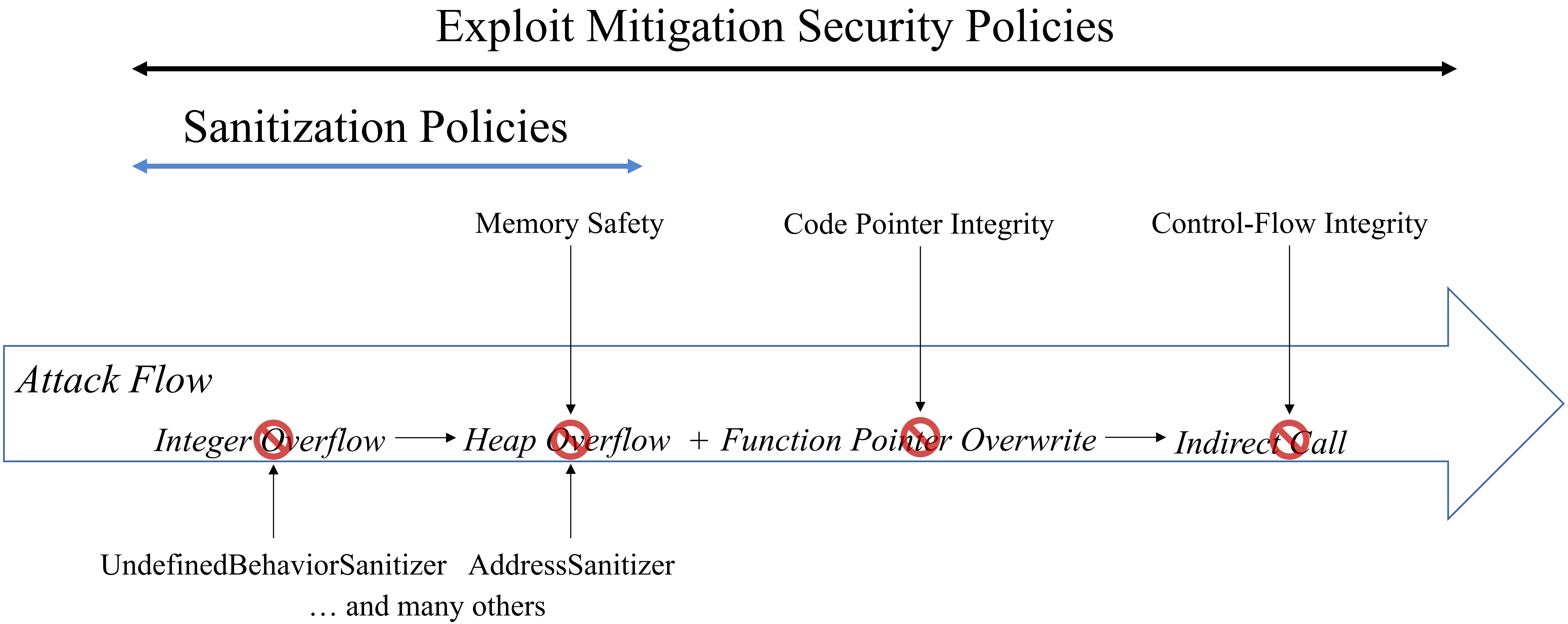


Exploit Mitigation vs. Sanitization (1/2)

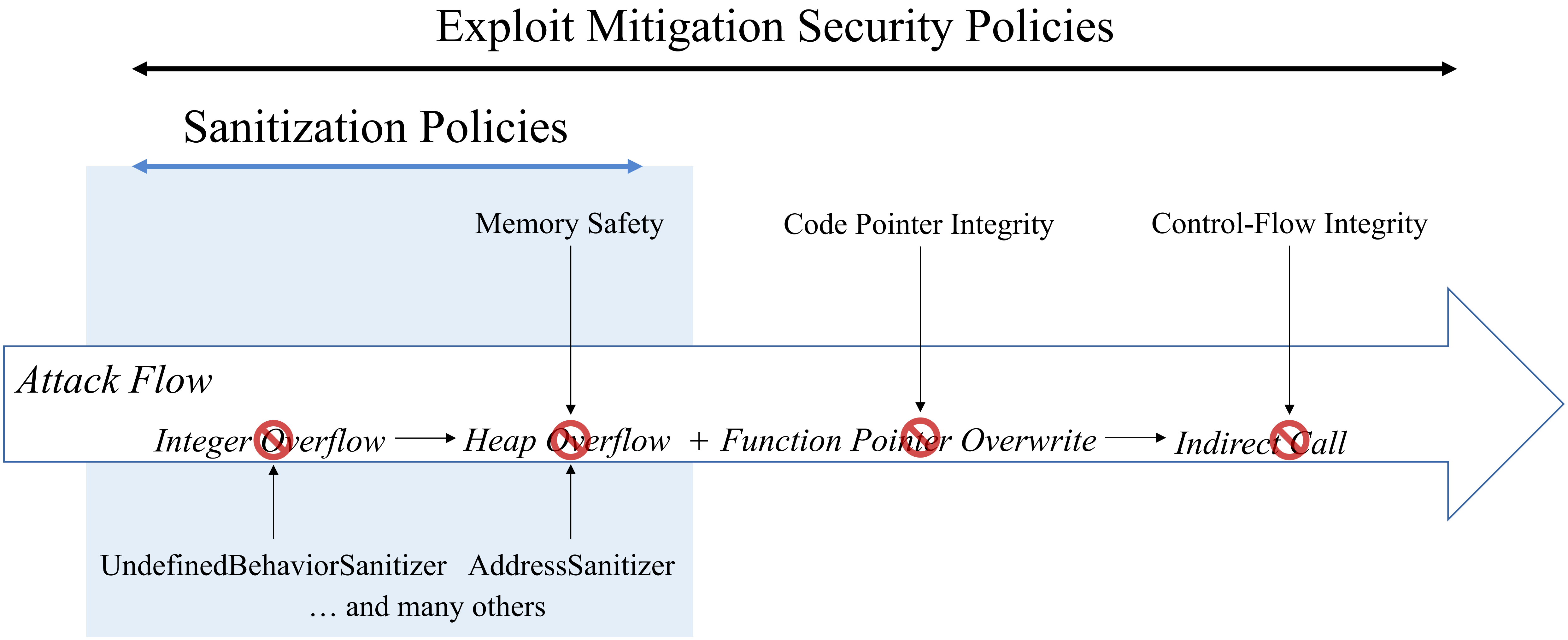
Attack Flow

Integer Overflow → *Heap Overflow* + *Function Pointer Overwrite* → *Indirect Call*

Exploit Mitigation vs. Sanitization (1/2)



Exploit Mitigation vs. Sanitization (1/2)



Exploit Mitigation vs. Sanitization (2/2)

	Exploit Mitigation	**Sanitization**
The goal is to …	Mitigate attacks	Find vulnerabilities
Used in …	Production	Pre-release
Performance budget is …	Very limited	Much higher
Policy violation leads to …	Program termination	Problem diagnosis
Violations triggered at location of bug	Sometimes	Always
Tolerance for FPs is …	Zero	Somewhat higher
Surviving benign errors is …	Desired	Not desired

Undefined Behavior in C/C++

- Buffer overflow
- Use-after-free
- Type errors
- Format string bug
- Signed integer overflow
- Null pointer dereferences
- etc.

J.2 Undefined behavior

The behavior is undefined in the following circumstances: …

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

…

— An object is referred to outside of its lifetime (6.2.4).

…

— A pointer is used to call a function whose type is not compatible with the referenced type.

…

— An object has its stored value accessed other than by an lvalue of an allowable type (6.5).

…

Undefined Behavior in C/C++

- Buffer overflow
- Use-after-free
- Type errors
- Format string bug
- Signed integer overflow
- Null pointer dereferences
- etc.

→ Well-known Security Vulnerabilities

J.2 Undefined behavior
The behavior is undefined in the following circumstances: …

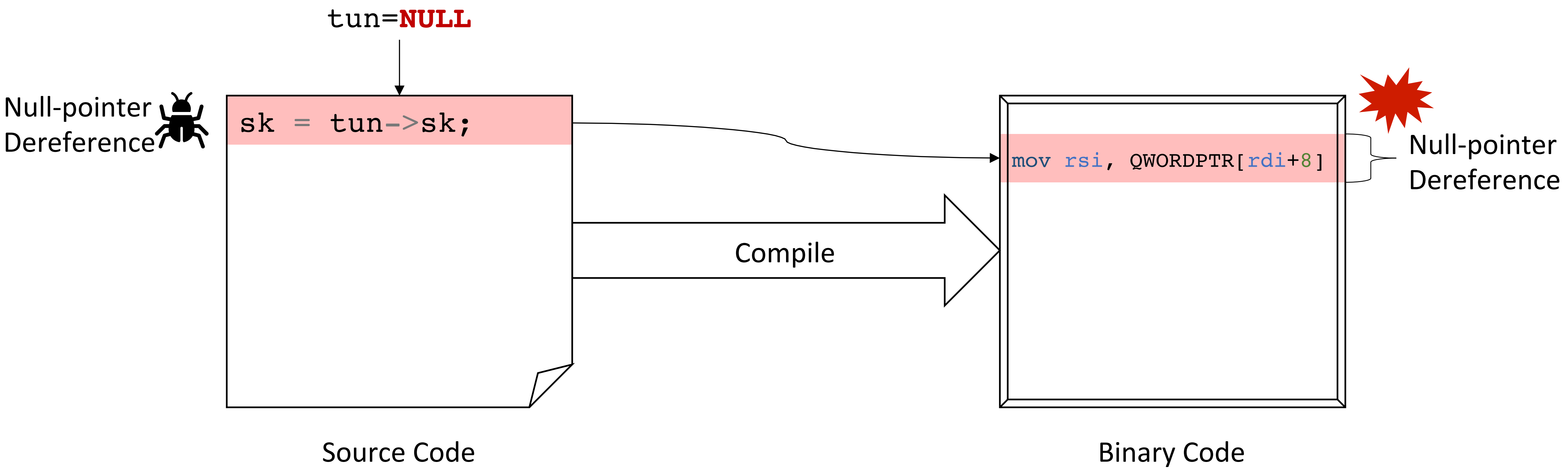
— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).
…

— An object is referred to outside of its lifetime (6.2.4).
…

— A pointer is used to call a function whose type is not compatible with the referenced type.
…
— An object has its stored value accessed other than by an lvalue of an allowable type (6.5).
…

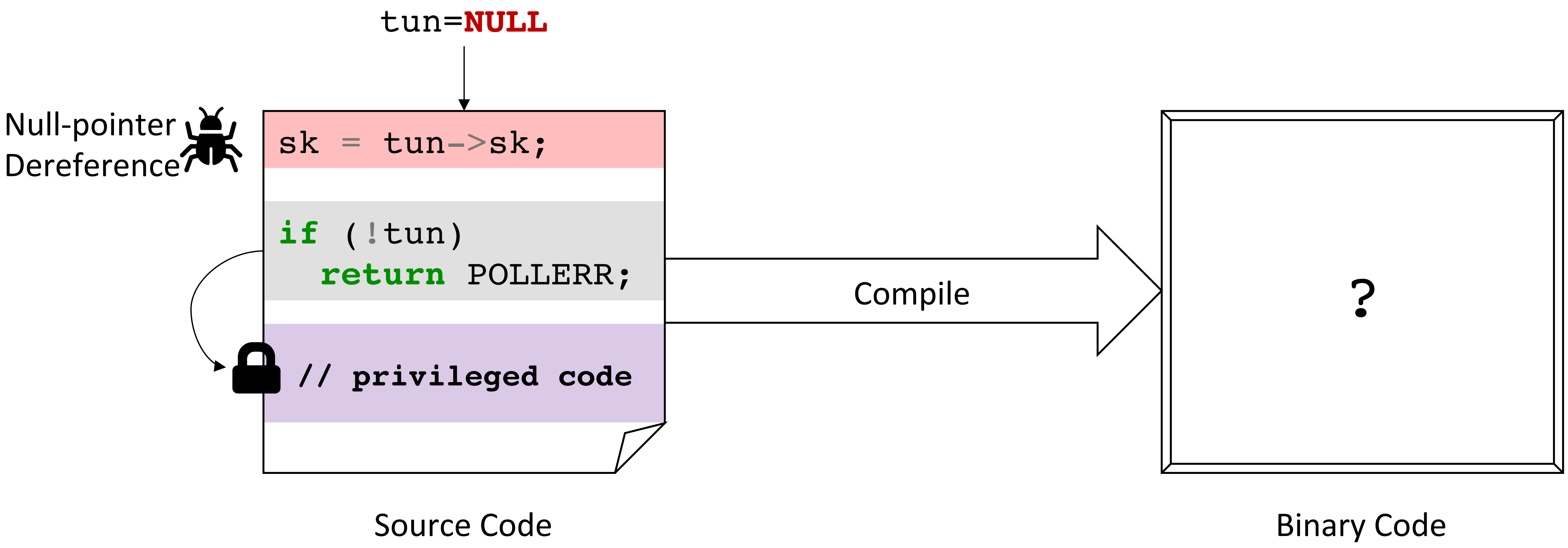
Security Implications of Undefined Behavior in C/C++ (1/2)

1. Memory and type safety violations vulnerable to memory exploits



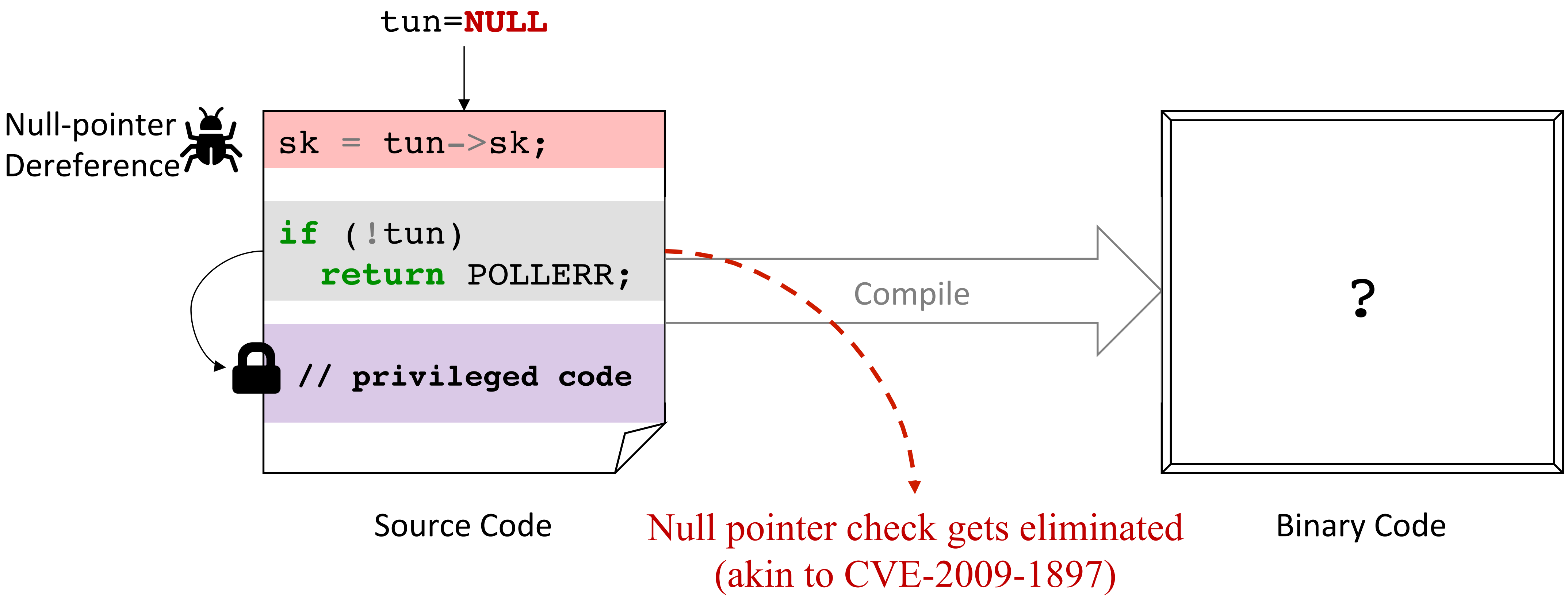
Security Implications of Undefined Behavior in C/C++ (2/2)

2. Compilation of a program having UBs may result in vulnerable code



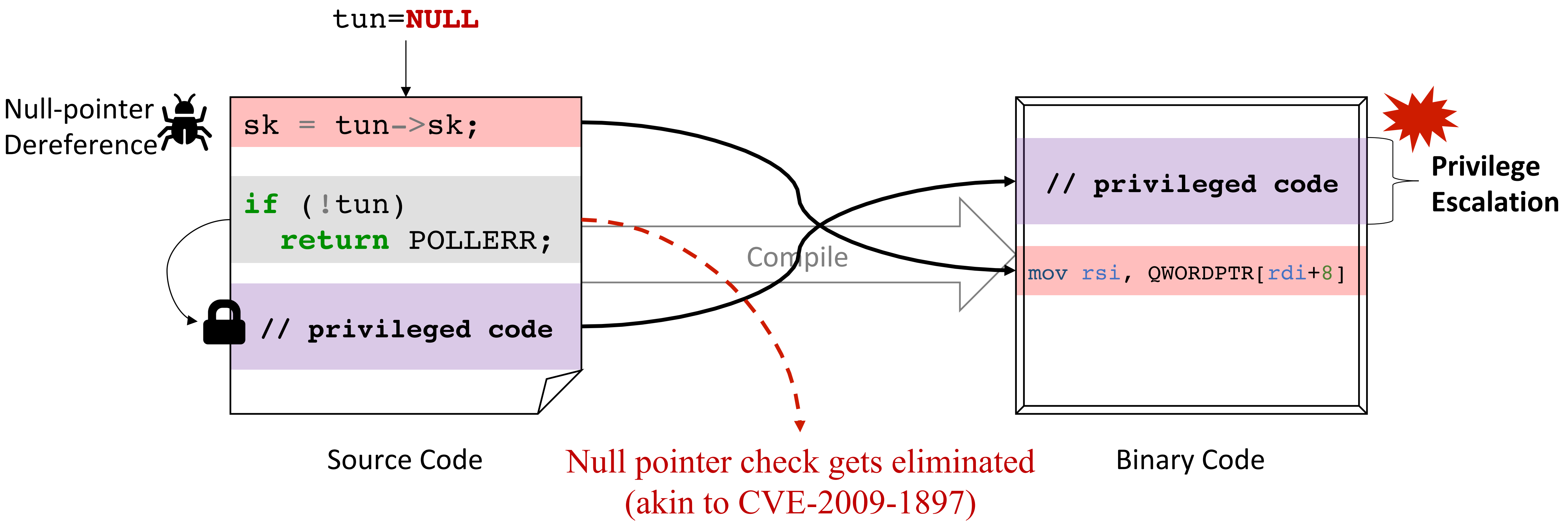
Security Implications of Undefined Behavior in C/C++ (2/2)

2. Compilation of a program having UBs may result in vulnerable code

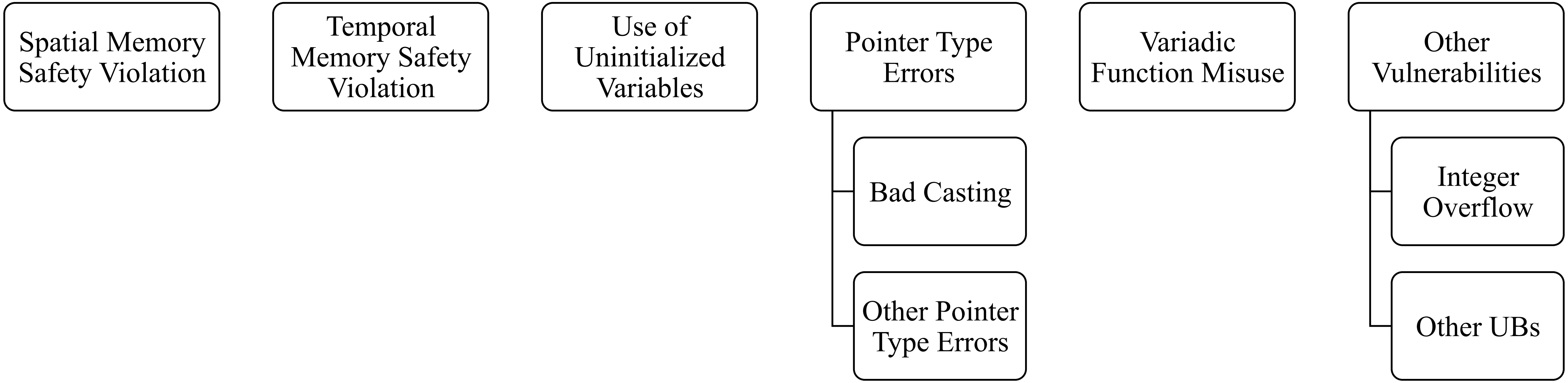


Security Implications of Undefined Behavior in C/C++ (2/2)

2. Compilation of a program having UBs may result in vulnerable code

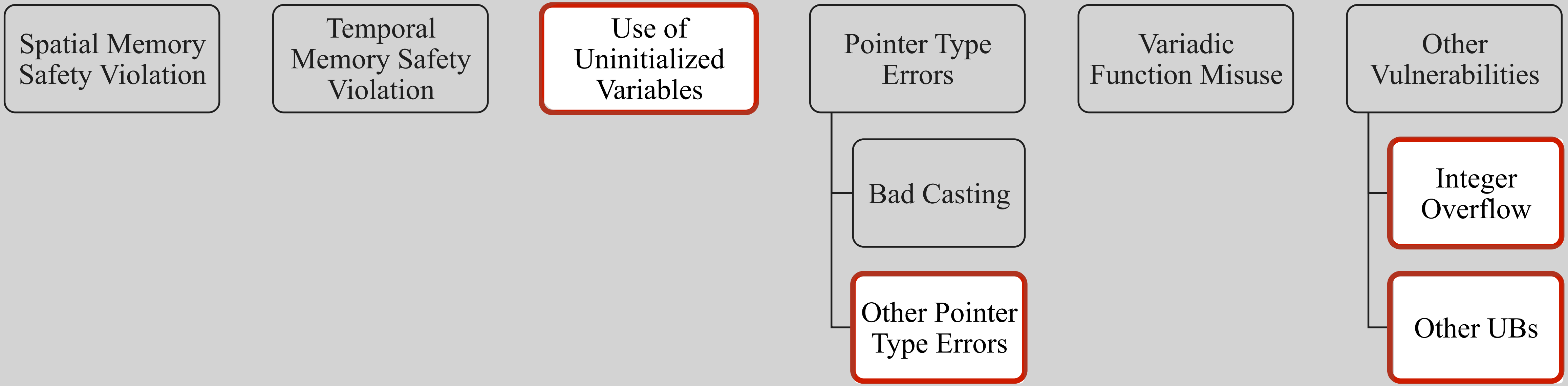


Low-Level Vulnerabilities in C/C++ (1/2)



- **Most of these vulnerabilities** can manifest as memory and type safety violations.

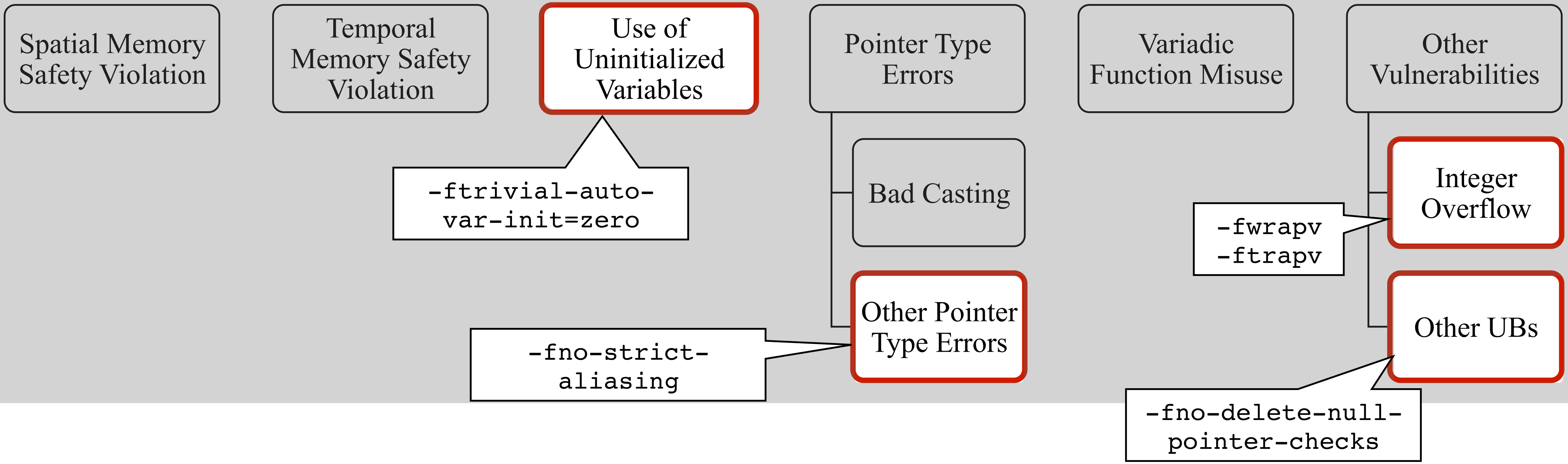
Low-Level Vulnerabilities in C/C++ (2/2)



- **Some UBs** may lead to unsafe code generation today.
- And, *things can change* as compiler optimizations evolve – called **time-bombs***.

* W. Dietz, P. Li, J. Regher, and V. Adve; “Understanding integer overflow in C/C++.” In *ICSE*, 2012.

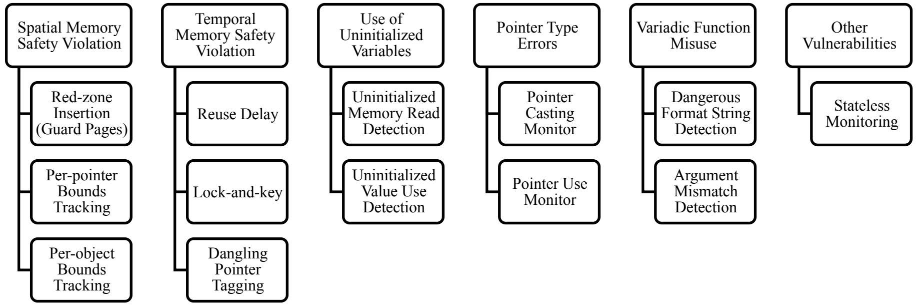
Low-Level Vulnerabilities in C/C++ (2/2)



- **Some UBs** may lead to unsafe code generation today.
- And, *things can change* as compiler optimizations evolve – called **time-bombs***.

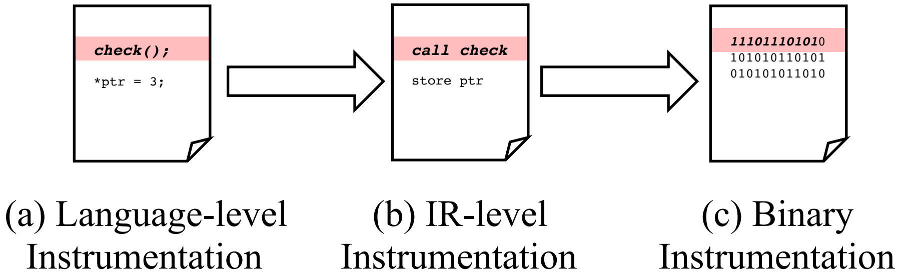
* W. Dietz, P. Li, J. Regher, and V. Adve; “Understanding integer overflow in C/C++.” In *ICSE*, 2012.

Sanitizer Design and Implementation

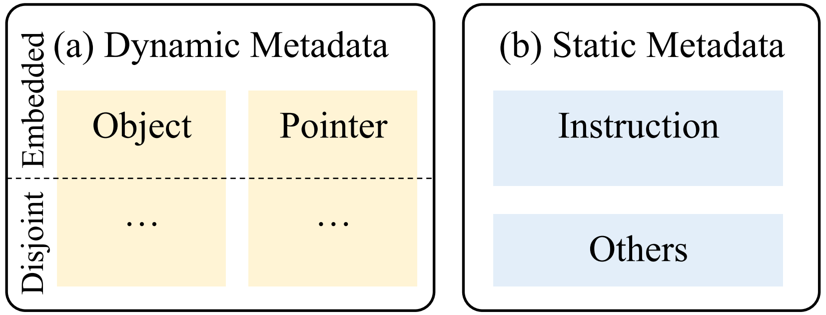


Bug Finding Technique

Program Instrumentation

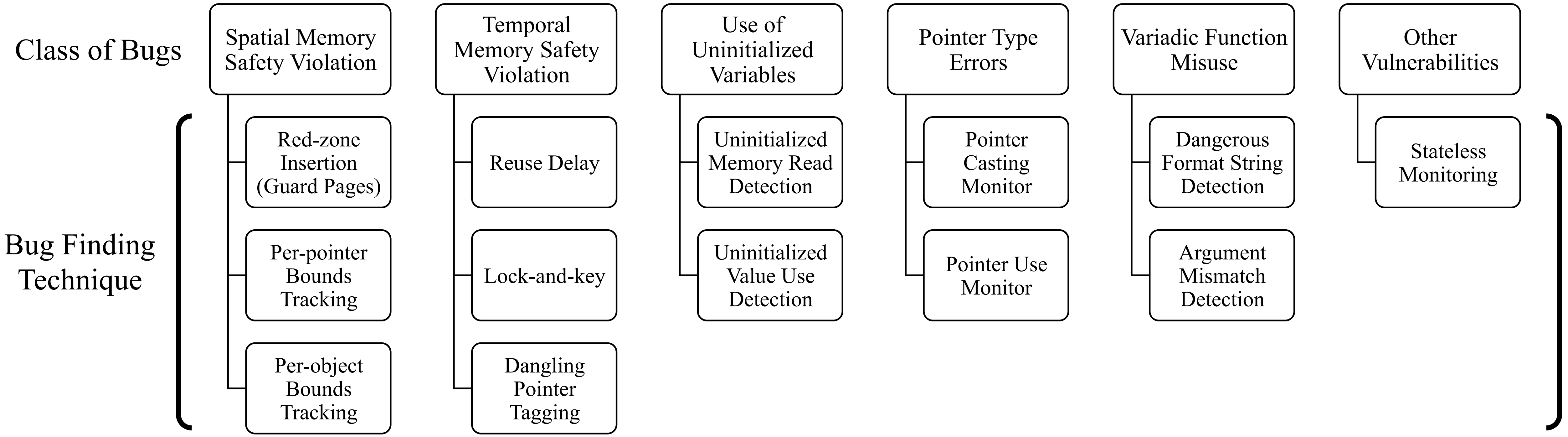


Metadata Management



Sanitizer Design and Implementation: Bug Finding Techniques

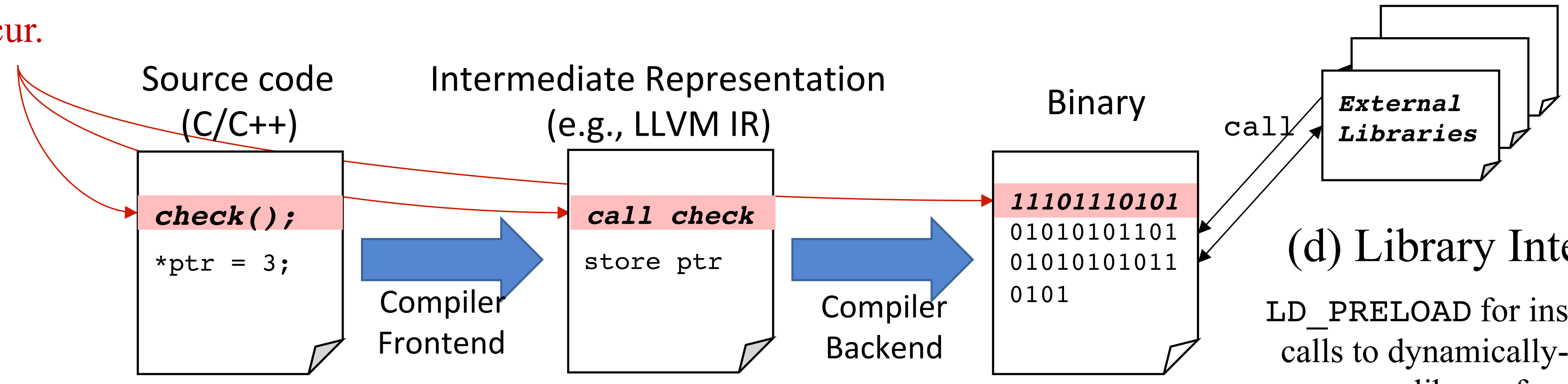
Bug Finding Techniques



Sanitizer Design and Implementation: Program Instrumentation

Program Instrumentation

Inlined Reference Monitor:
Fine-grained run-time monitoring of program behavior to detect bugs as they occur.

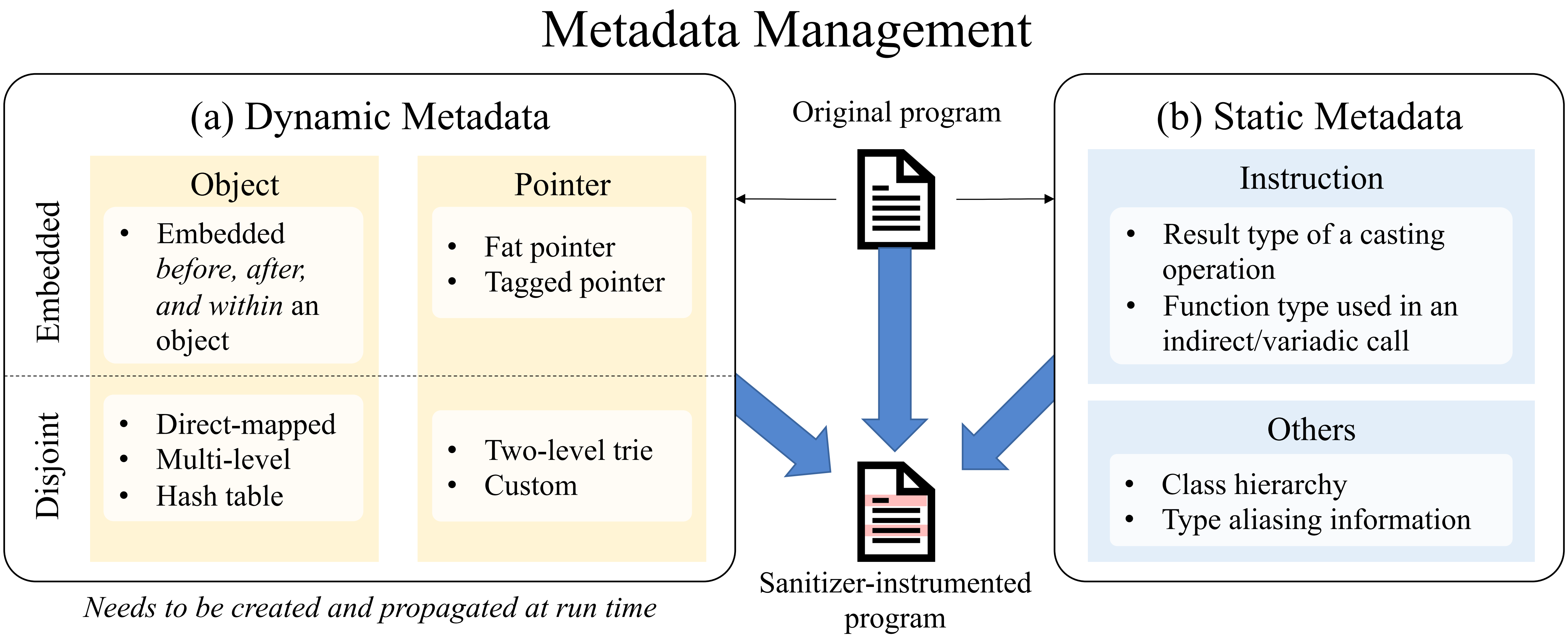


(a) Language-level (b) IR-level (c) Binary-level

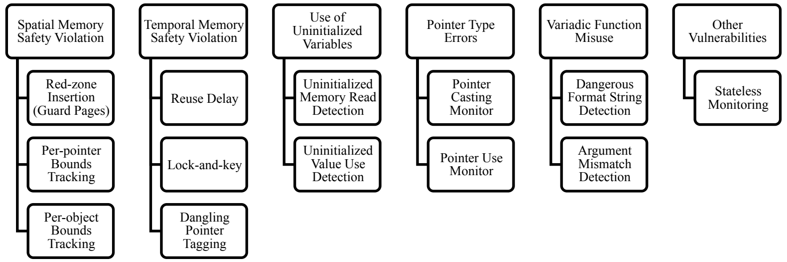
(d) Library Interposition

LD_PRELOAD for instrumenting only calls to dynamically-linked external library functions

Sanitizer Design and Implementation: Metadata Management

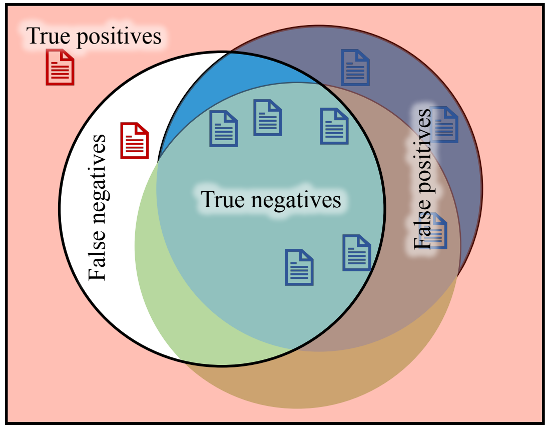


Sanitizer Design and Implementation: Precision and Overheads

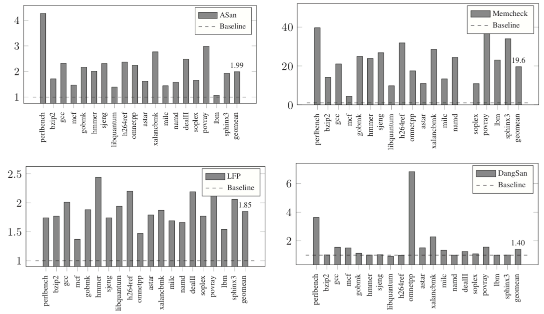


Bug Finding Technique

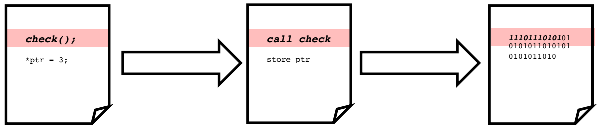
Bug Detection Precision and Compatibility



Performance and Memory Overheads



Program Instrumentation

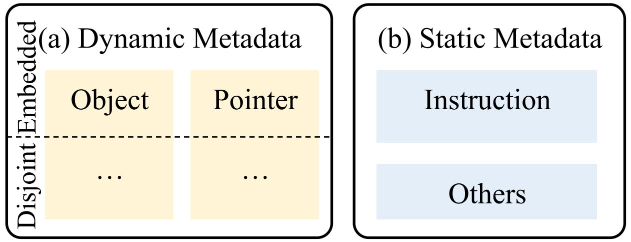


(a) Language-level Instrumentation

(b) IR-level Instrumentation

(c) Binary Instrumentation

Metadata Management



Our Analysis of Sanitizers

- Our analysis of 37 tools
  - We benchmarked 10 publicly available sanitizers on the same experimental platform (https://github.com/securesystemslab/sanitizing-for-security-benchmarks)
- Main observations
  - Performance is not a primary concern
  - Many false positives (marked as) in tools other than widely-used ones such as ASan
  - Most (◔ ◑ ◕ ly have partial coverage of bugs ( )
  - Widely deployed tools such as ASan have even smaller coverage

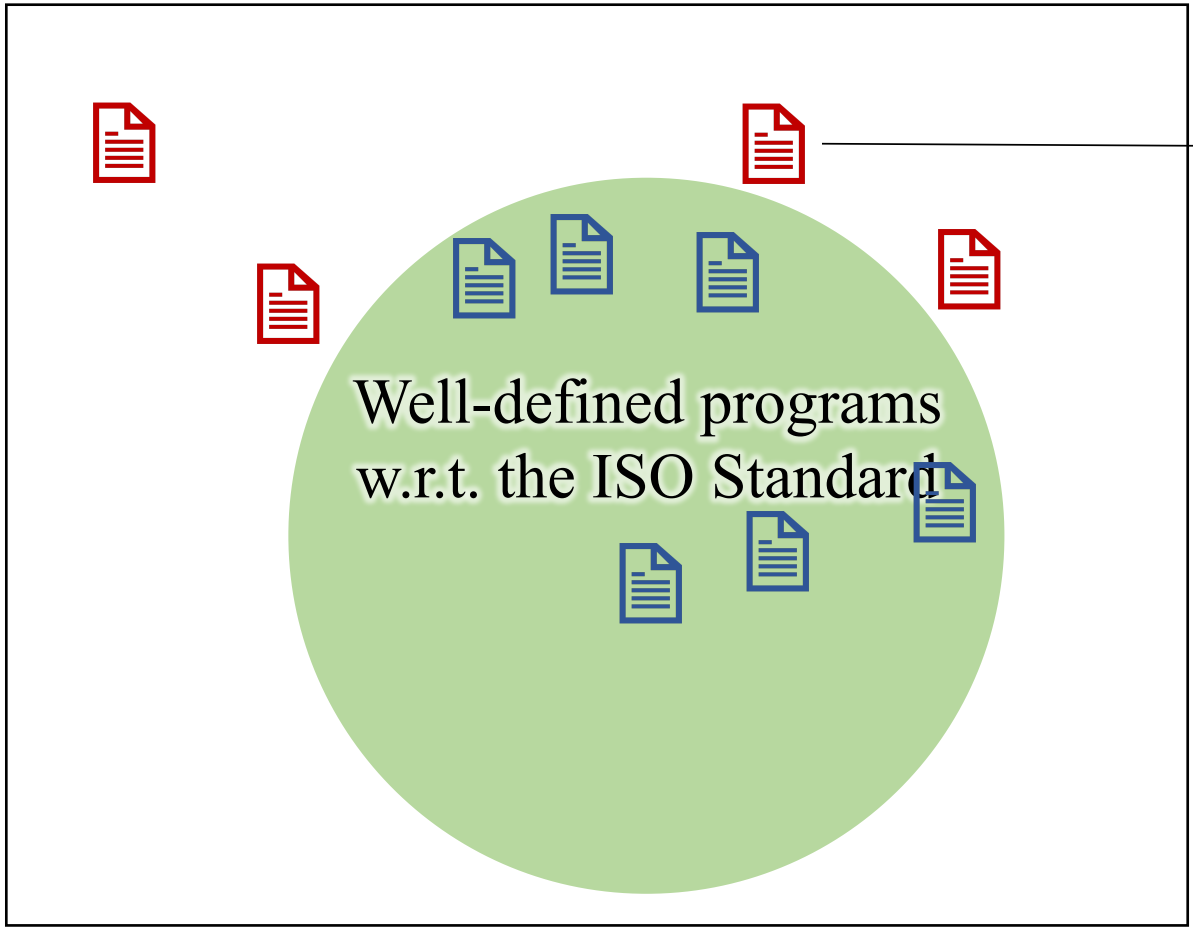
Sanitizers			IV. Bug Finding Techniques													V. Instr.				VI. Metadata Mgmt.							VIII. Analysis										
	Year Published	Actively Maintained	Red-zone Insertion	Guard Pages	Reuse Delay	Per-pointer Bounds Tracking	Per-object Bounds Tracking	Lock-and-key	Dangling Pointer Tagging	Uninit. Mem. Read Detection	Uninit. Value Use Detection	Pointer Casting Monitor	Pointer Use Monitor	Variadic Arg. Mismatch Detection	Stateless Monitoring	Language-level Instr.	IR-level Instr.	Binary Instr.	Library Interposition	Embedded Metadata	Direct-mapped Shadow	Multi-level Shadow	Custom Data Structure	Fat/Tagged Pointers	Disjoint Per-pointer Metadata	Static Metadata	Spatial Safety Violation (III-A1)	Temporal Safety Violation (III-A2)	Use of Uninit. Variables (III-B)	Bad-casting (III-C)	Load Type Mismatch (III-C)	Func. Call Type Mismatch (III-C)	Variadic Func. Misuse (III-D)	Signed Integer Overflow (III-E)	Other Undef. Behavior (III-E)	Performance Overhead	Memory Overhead
Purify [27]	'92		✔		✔					✔								✔			✔						◔	◑	●							◕	●
Memcheck [28]	'05	✔	✔		✔						✔							✔				✔					◔	◑	◑							○*	n/a
Dr. Memory [29]	'11	✔	✔		✔						✔							✔				✔					◔	◑	◑							○	n/a
LBC [30]	'12		✔													✔						✔					◕									◕	◑
ASan [31]	'12	✔	✔		✔												✔				✔						◕	◑								◕*	◑
Electric Fence [32]	'93			✔	✔														✔								◔	◕								n/a	○
PageHeap [33]	'00			✔	✔														✔								◔	◕								n/a	○
D&A Dangling [35]	'06			✔	✔												✔											◕								◕	n/a
Oscar [36]	'17			✔	✔														✔	✔								◕								◕	●
RTCC [45]	'92					✔										✔								✔			◕									◑	n/a
Safe-C [46]	'94					✔		✔								✔								✔	✔		●	●								◑	●
P&F [47]	'97					✔		✔								✔									✔		●	●								◑	●
MSCC [52]	'04					✔		✔								✔									✔		◕	◕								◕	●
SoftBound+CETS [48], [56]	'10					✔		✔									✔								✔		●	●								◕	●
Intel Pointer Checker [49]	'12	✔				✔			✔								✔								✔		●	◔								◕	●
SGXBounds [54]	'17					✔											✔			✔				✔			◕									◕	●
CUP [55]	'18				✔	✔											✔							✔	✔		◕	◑								◕	n/a
J&K [34]	'97						✔									✔							✔				◕									◑	n/a
CRED [37]	'04						✔									✔							✔				◕									◕	n/a
D&A Bounds [38]	'06						✔										✔						✔				◕									◕	n/a
BBC [39]	'09						✔										✔							✔			◑									◕	●
PAriCheck [41]	'10						✔									✔							✔				◑									◕	n/a
Low-fat Pointer [42], [43]	'17	✔					✔										✔							✔			◑									◕*	●
Undangle [57]	'12								✔									✔							✔			◕								○	○
FreeSentry [59]	'15								✔							✔						✔	✔					◕								◕	n/a
DangNull [58]	'15								✔								✔						✔					◑								◕	●
DangSan [60]	'17								✔								✔					✔	✔					◑								◕*	●
MSan [66]	'15	✔									✔						✔				✔								◕							◕*	●
CaVer [69]	'15											✔				✔	✔						✔			✔				◑						◕	●
TypeSan [70]	'16											✔				✔	✔					✔				✔				◑						◕*	●
HexType [71]	'17											✔				✔	✔						✔			✔				◕						●*	●
Loginov et al. [72]	'01												✔			✔					✔										◑					○	n/a
LLVM TySan [73]	'17	✔											✔			✔	✔				✔										◕					n/a	◑
EffectiveSan [74]	'18												✔			✔	✔			✔				✔		✔	●	◑			◕					◑	●
Clang CFI [68]	'15	✔										✔	✔			✔	✔			✔						✔				◔		●				●*	●
HexVASAN [79]	'17													✔			✔						✔			✔							●			●*	●
UBSan [67]	'12	✔										✔	✔		✔	✔				✔						✔				◔		●		●	◑	◕*	●

Precision: False Positives and False Negatives

Programs conforming to the ISO standard

Precision: False Positives and False Negatives

Precision: False Positives and False Negatives



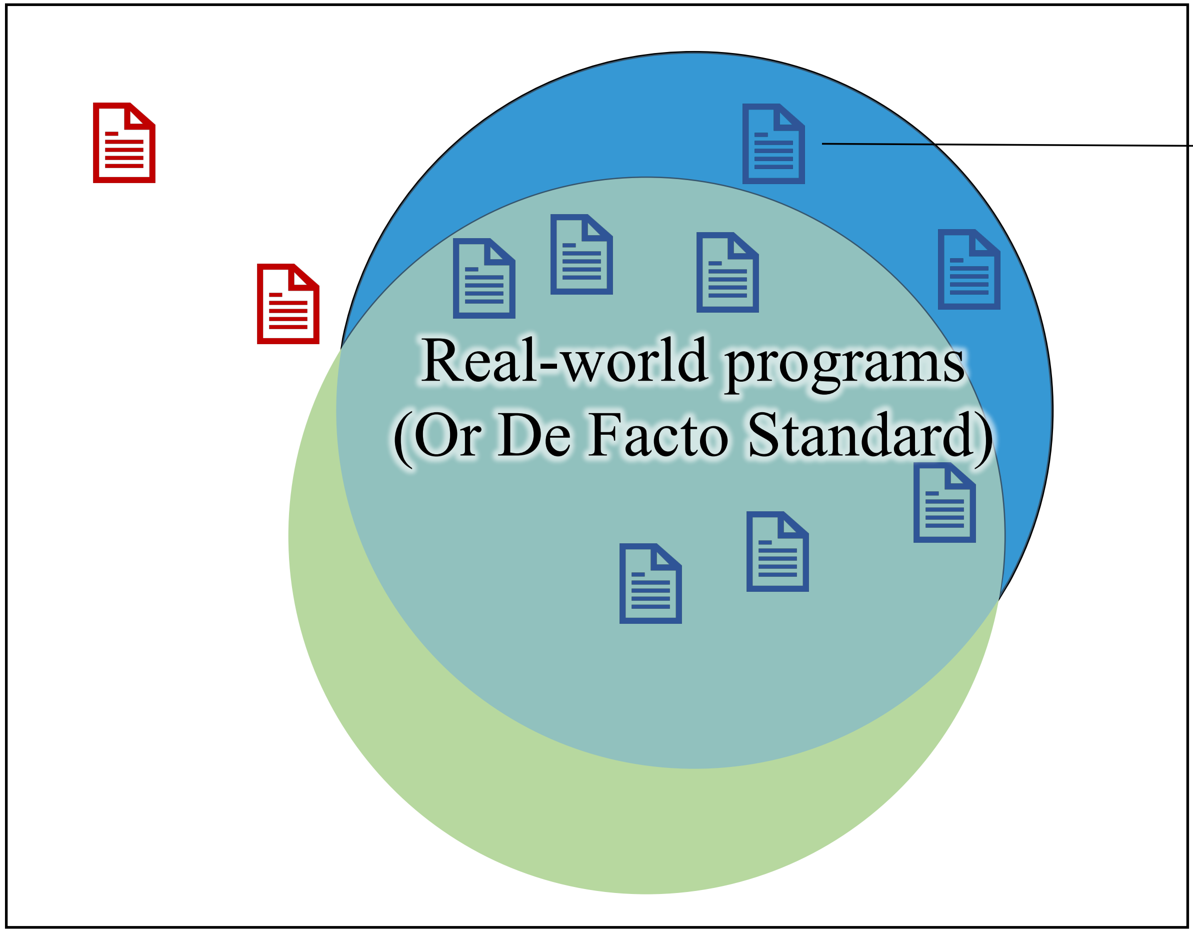
e.g., programs creating OOB pointers

“*Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object.** ”

Programs conforming to the ISO standard

* ISO/IEC JTC1/SC22/WG14. ISO/IEC 9899:2011, Programming Languages — C

Precision: False Positives and False Negatives



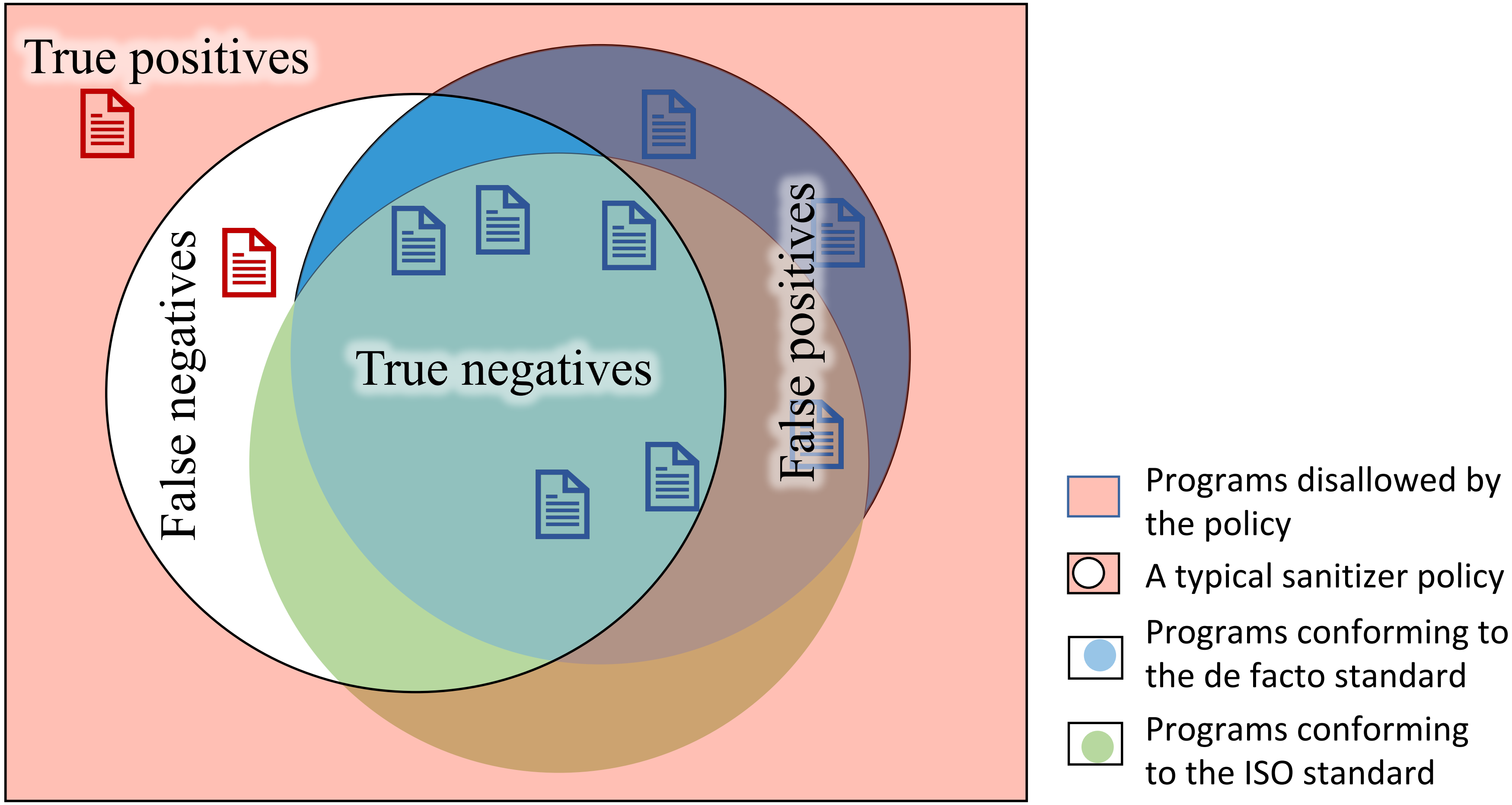
e.g., programs creating OOB pointers

> *But in practice it seems to be common to transiently construct out-of-bounds pointers.* *

- Programs conforming to the de facto standard
- Programs conforming to the ISO standard

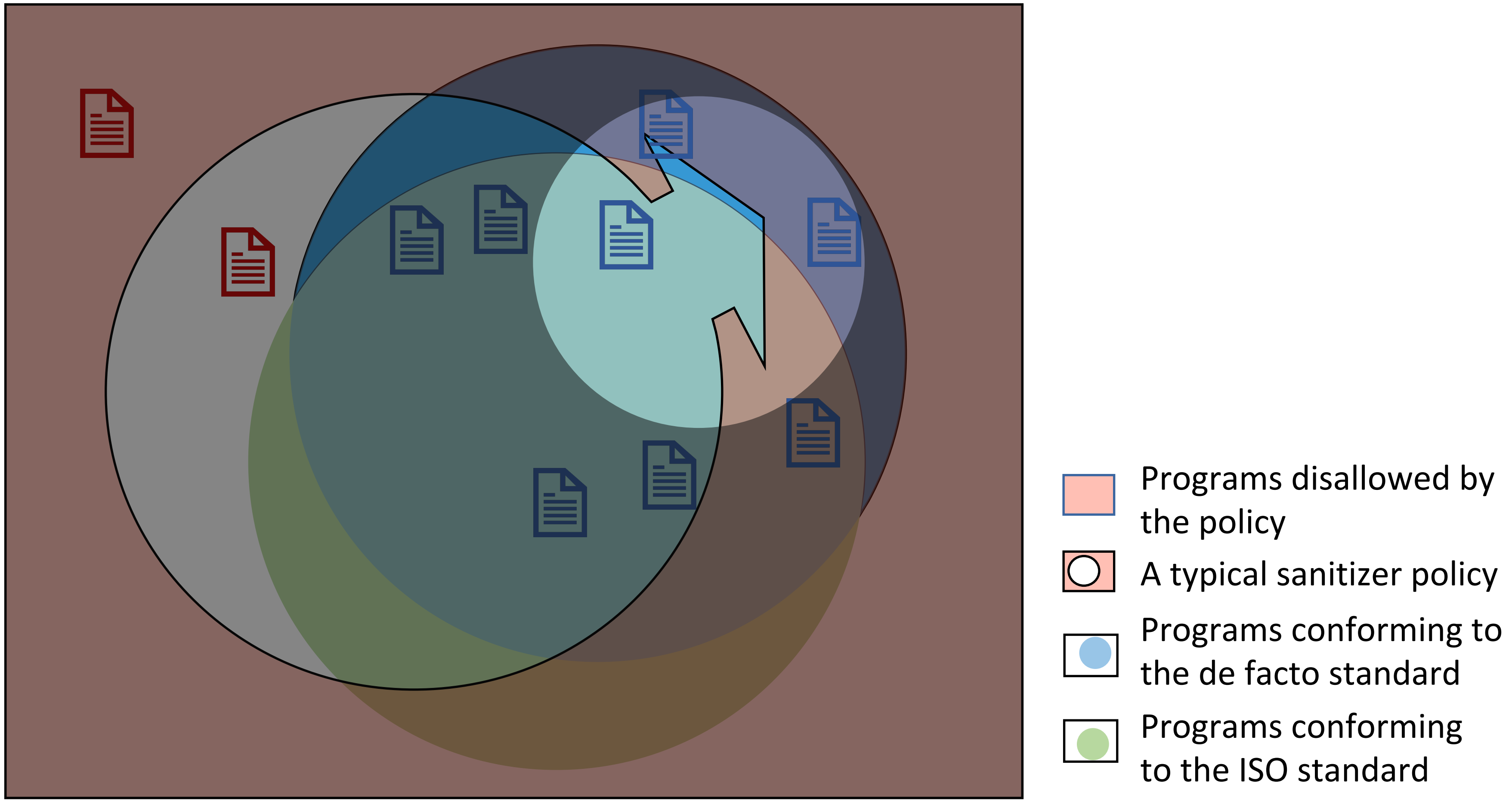
* K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell. Into the Depths of C: Elaborating the De Facto Standards. In *PLDI'16*

Precision: False Positives and False Negatives



Reducing Precision Gaps (1/2): Standard Compatibility

Compatibility with the ISO *and* de facto standards



Reducing Precision Gaps (1/2): Standard Compatibility

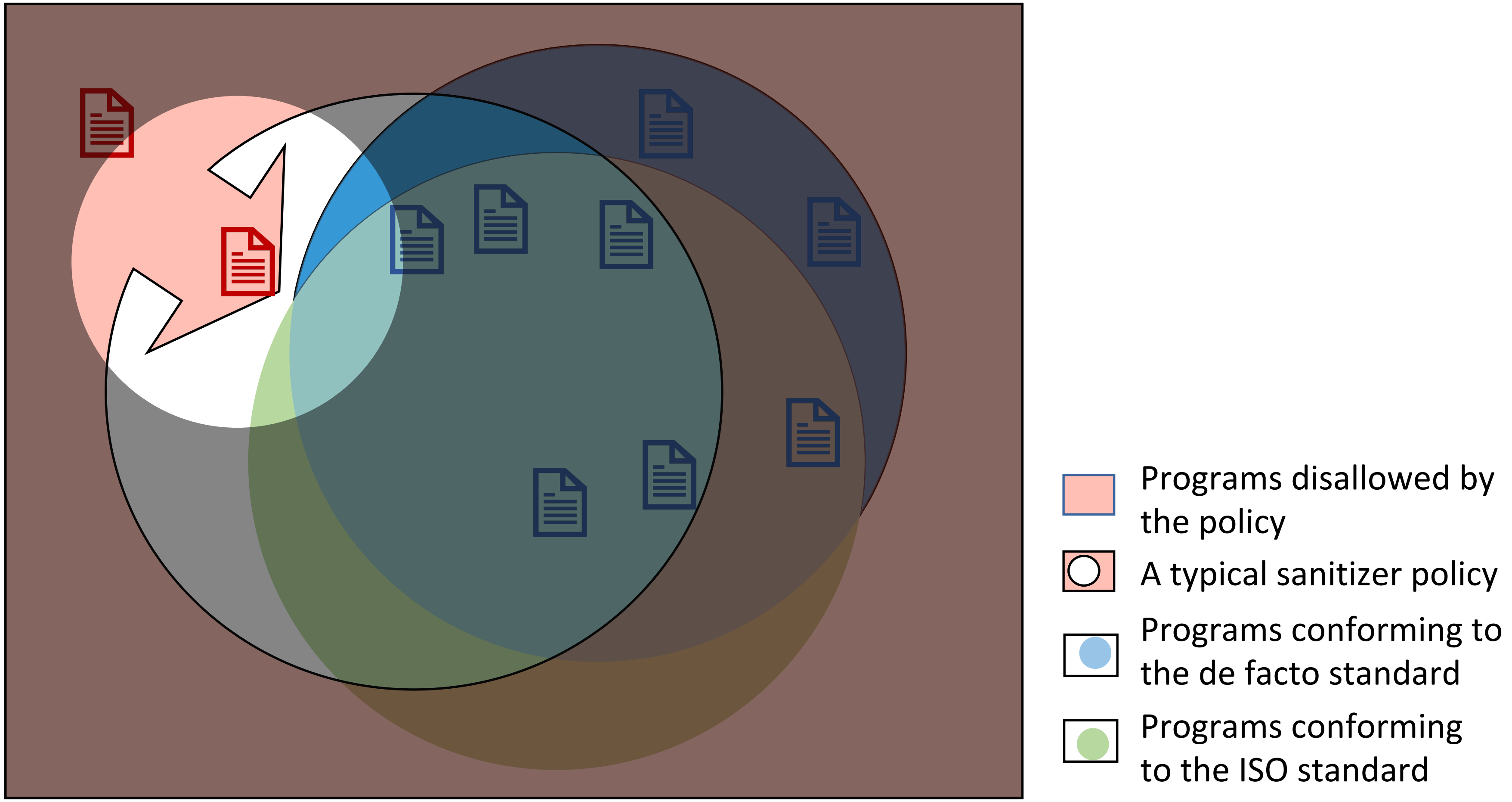
- Many programs transiently construct OOB pointers (de facto standards)
  - Thus, supporting this code idiom increases a tool's applicability

- Many tools, however, do not permit transient construction of OOB pointers
  - Some bounds checking tools invalidate pointers as soon as they go OOB
  - Dangling pointer tagging tools may incorrectly invalidate pointers, if OOB pointers are transiently stored in memory

- Many tools only support ISO standard compatibility by adding one byte between objects – this is not enough in practice

Reducing Precision Gaps (2/2): Finding Elusive Bugs

Finding bugs that elude existing or widely-deployed sanitizers



Reducing Precision Gaps (2/2): Finding Elusive Bugs

- Subclasses of memory safety violations that elude AddressSanitizer:
  - Intra-object buffer overflow
  - Buffer overflow into a valid but unintended object
  - Uses of freed memory that are being reused

- Type errors beyond bad casting (`static_casts`)
  - C programs or C++ programs using C-style casts and `reinterpret_casts`
  - Type errors are UBs that may silently break programs if does not instruct the compiler using flags like `-fno-strict-aliasing`

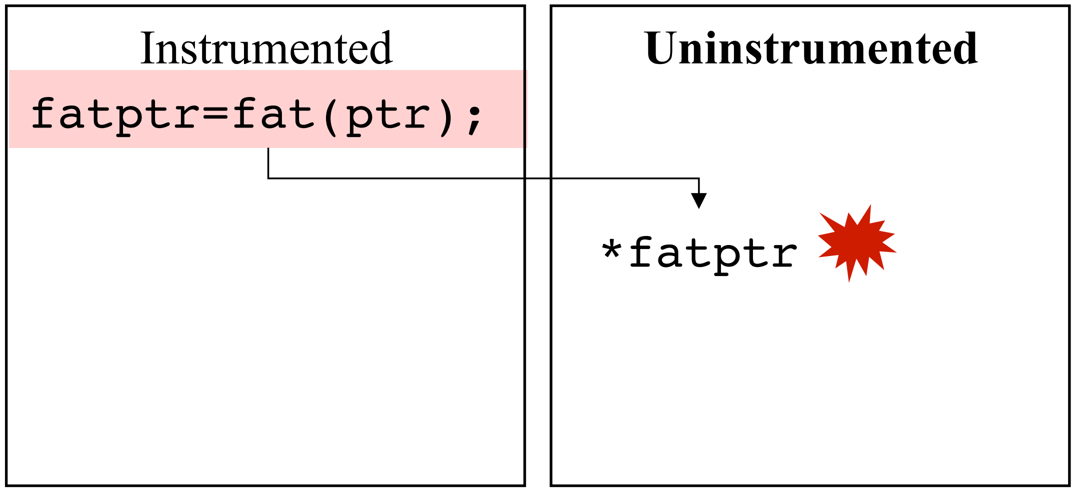
Reducing Precision Gaps (2/2): Finding Elusive Bugs

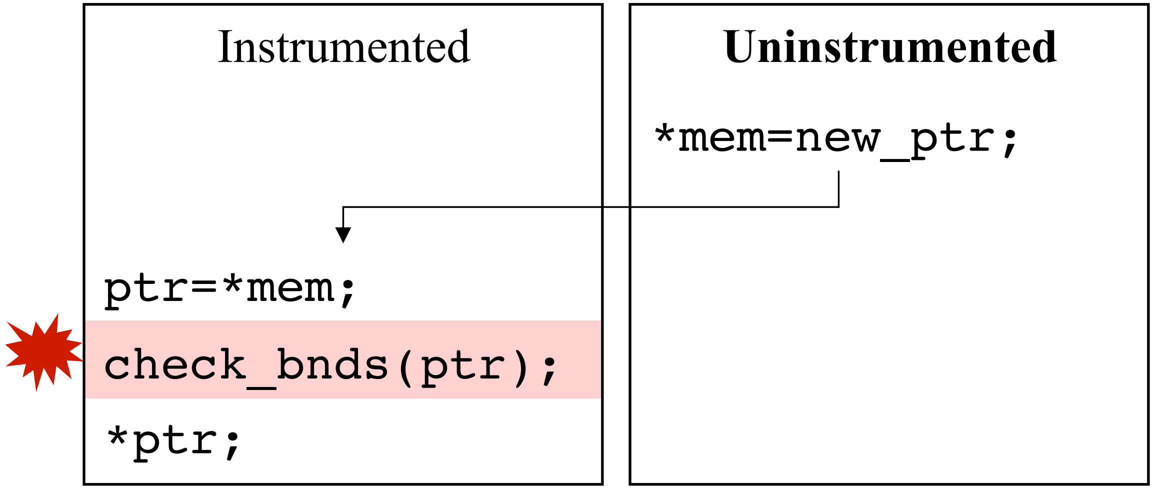
- Finding these elusive bugs, in general, requires **more precise dynamic metadata tracking**
  - Tracking **per-pointer metadata** such as pointer bounds
  - Tracking **effective types** of object storage in memory

- However, such metadata tracking poses precision and performance challenges
  - C’s weak type safety (e.g., pointer to integer casts, uses of void pointers) makes pointer metadata tracking difficult
  - The C standard has complex effective type and aliasing rules

- More research is needed in developing sanitizers that can find these elusive bugs while maintaining good compatibility

Pointer Metadata Tracking Challenges: Uninstrumented Code





Fat pointer is not compatible with uninstrumented code

Disjoint pointer metadata can get outdated, when uninstrumented code updates a pointer without updating corresponding metadata

Pointer Metadata Tracking Challenges: Pointer to Integer Casts

- Even with full instrumentation (via, e.g., dynamic binary translation), sanitizers can break programs having **pointer to integer casts**

```
some_object_type * → uint64_t
```

- Incompatible with fat/tagged pointers

- Disjoint pointer metadata can be a choice, but full pointer flow tracking across casts between pointers and integers can be expensive

- Existing tools stop tracking pointer metadata once they are cast to integers

Pointer Metadata Tracking Challenges: Multi-threaded Programs

- *Race-free* programs that use atomic operations can be problematic
  - Concurrent atomic operations from different threads without putting corresponding metadata operations into the same atomic unit can make metadata go out-of-sync

- Example of **naïve instrumentation**:

Thread A

❶ `atomic_store(addr_of_ptr, ptrA);`

❹ `*metadata_of_ptr = metadata_of_ptrA;`

metadata for `ptr` out-of-sync!

Thread B

❷ `atomic_store(addr_of_ptr, ptrB);`

❸ `*metadata_of_ptr = metadata_of_ptrB;`

Instrumented code

Type Error Detection Challenges

- Rules about determining an effective type of a stored value (i.e., **effective type rules**) are complex, due to weakly-typed nature of C
  - Prevalent uses of void pointer type and (omnipotent) character pointer type
    - `malloc` returns `void *`
    - `memcpy`-family of functions take and return `void *`
  - Type punning through C-style casts and union
- There are some tools that implement over-approximations of effective type rules, but precision and performance trade-offs are yet to be explored.
- Also, type error checking itself can be costly, because C's **aliasing rules** permit a stored value to be accessed by using pointers of many different types

Other Future Research Directions

Composing sanitizers

- Can find bugs closer to their source without generating duplicated bug reports for the bug's side-effects

Using hardware features to improve performance and compatibility

- e.g., Pointer tagging/memory tagging support in HW

Kernel and bare metal support

- Sanitizers for OS kernels, or non-user-space programs in general

Q & A

Thank you!

Dokyung Song
Ph.D. Student at UC Irvine
dokyungs@uci.edu