All the Safeties

Sean Parent

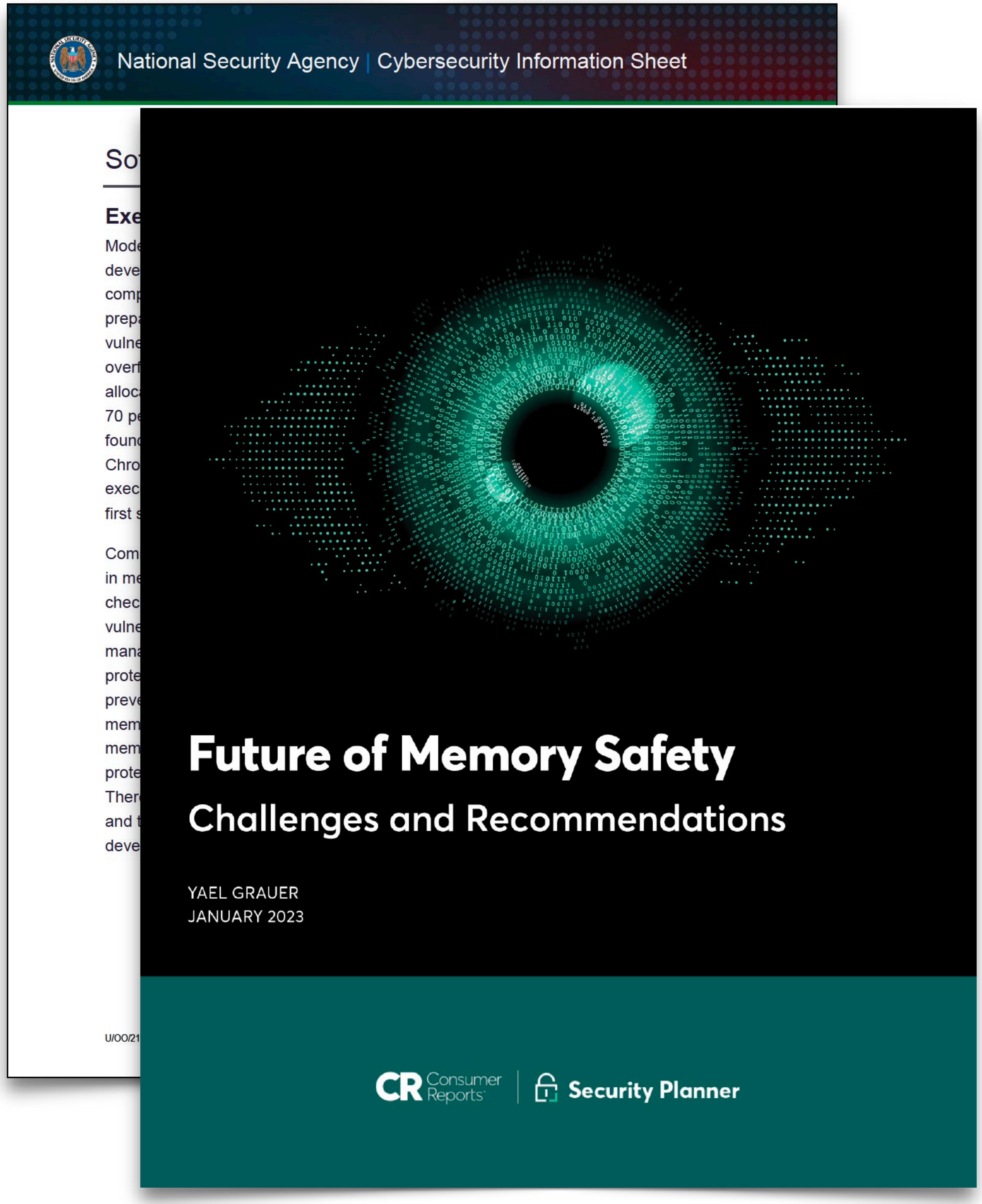
2023

C++ now

All the Safeties

Sean Parent | Sr. Principal Scientist
Adobe Software Technology Lab

Why Talk About Safety?

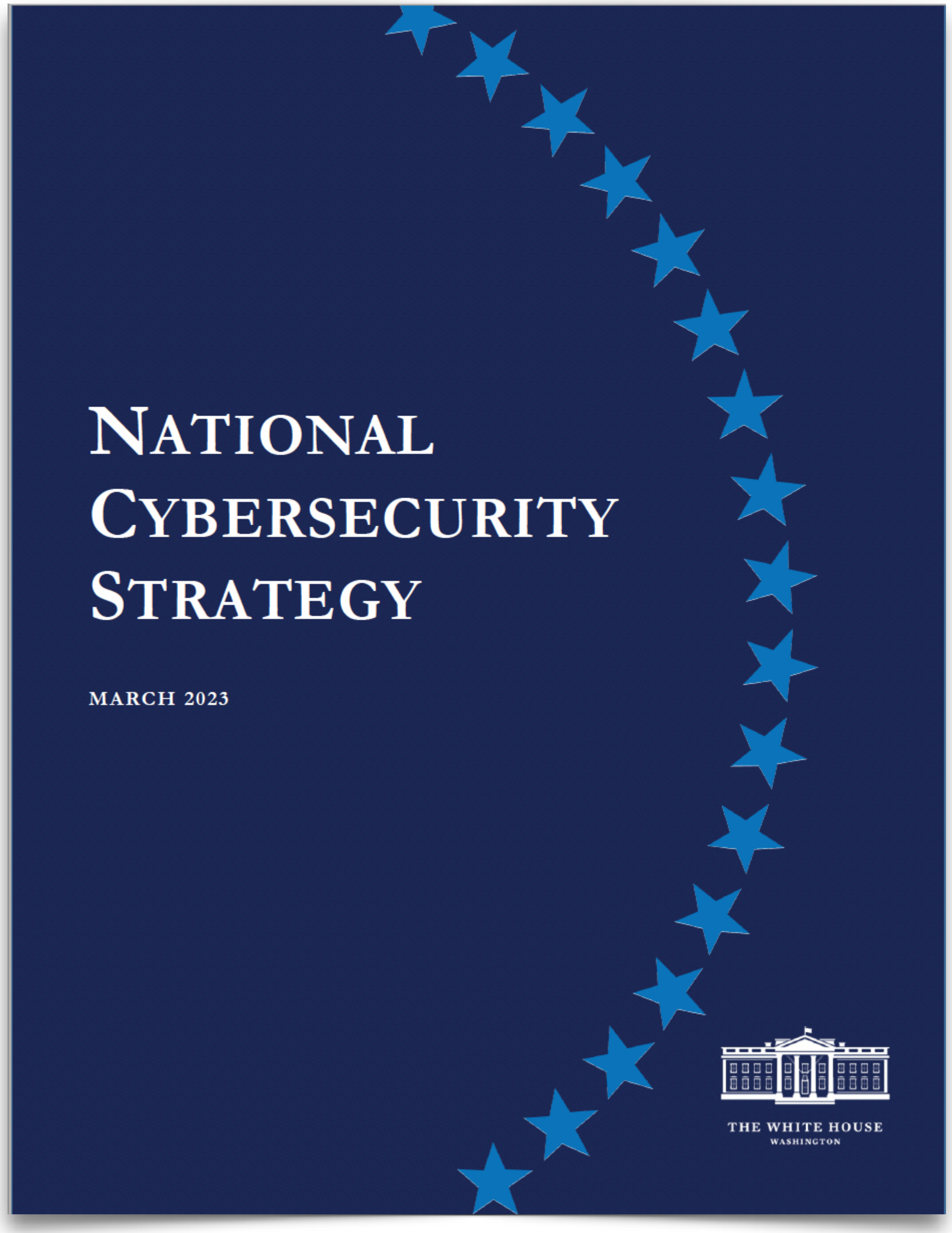


"70 percent of their vulnerabilities were due to ***memory safety issues****"*

"NSA recommends using a ***memory safe language*** *when possible."*

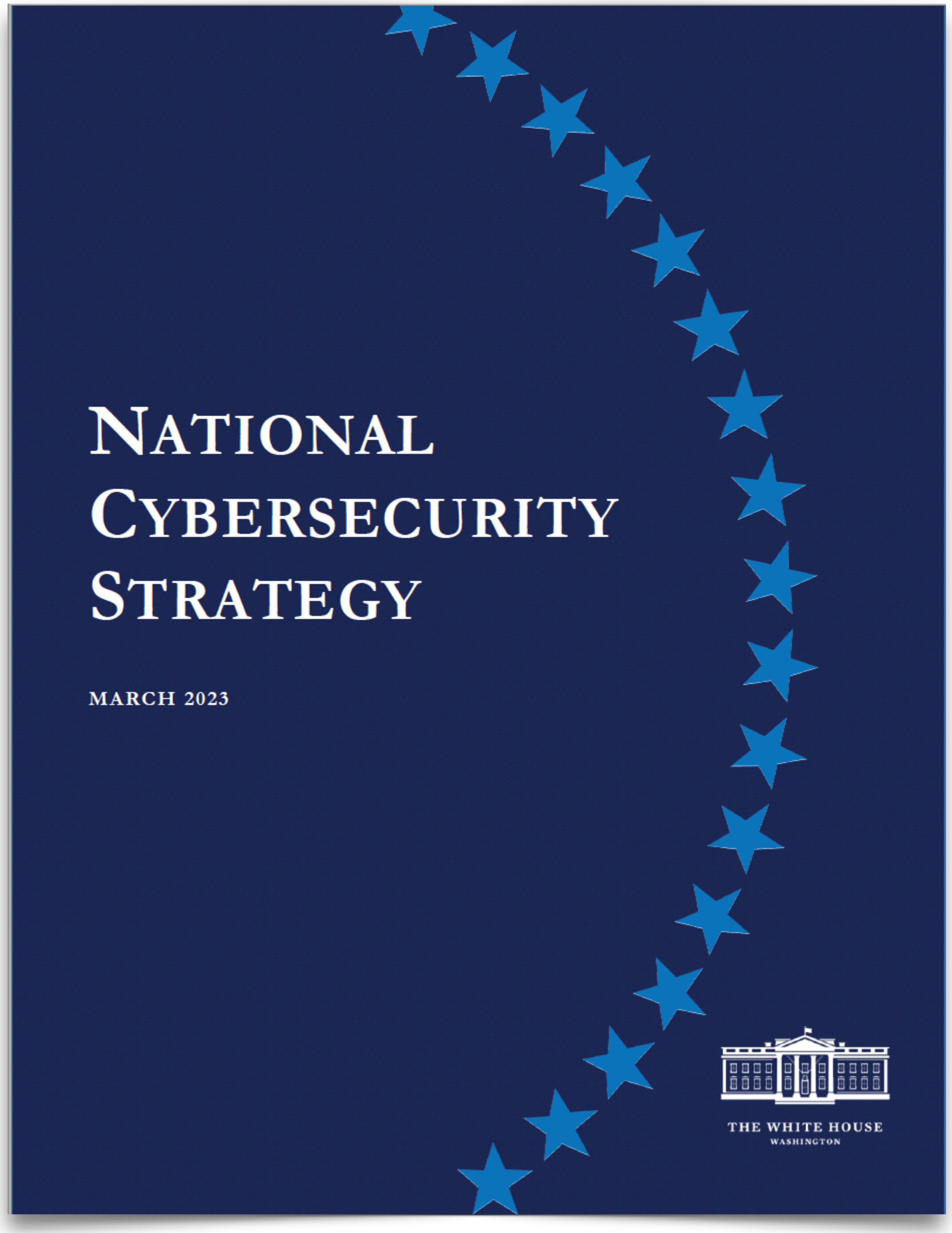
"Even when organizations put significant effort and resources into detecting, fixing, and mitigating this class of bugs, ***memory unsafety*** *continues to represent the majority of high-severity security vulnerabilities* ***and stability issues****."*

Why Talk About Safety?



"Cybersecurity is essential to the basic functioning of our economy, the operation of our critical infrastructure, the strength or our democracy and democratic institutions, the privacy of our data and communications, and our national defense."

Why Talk About Safety?



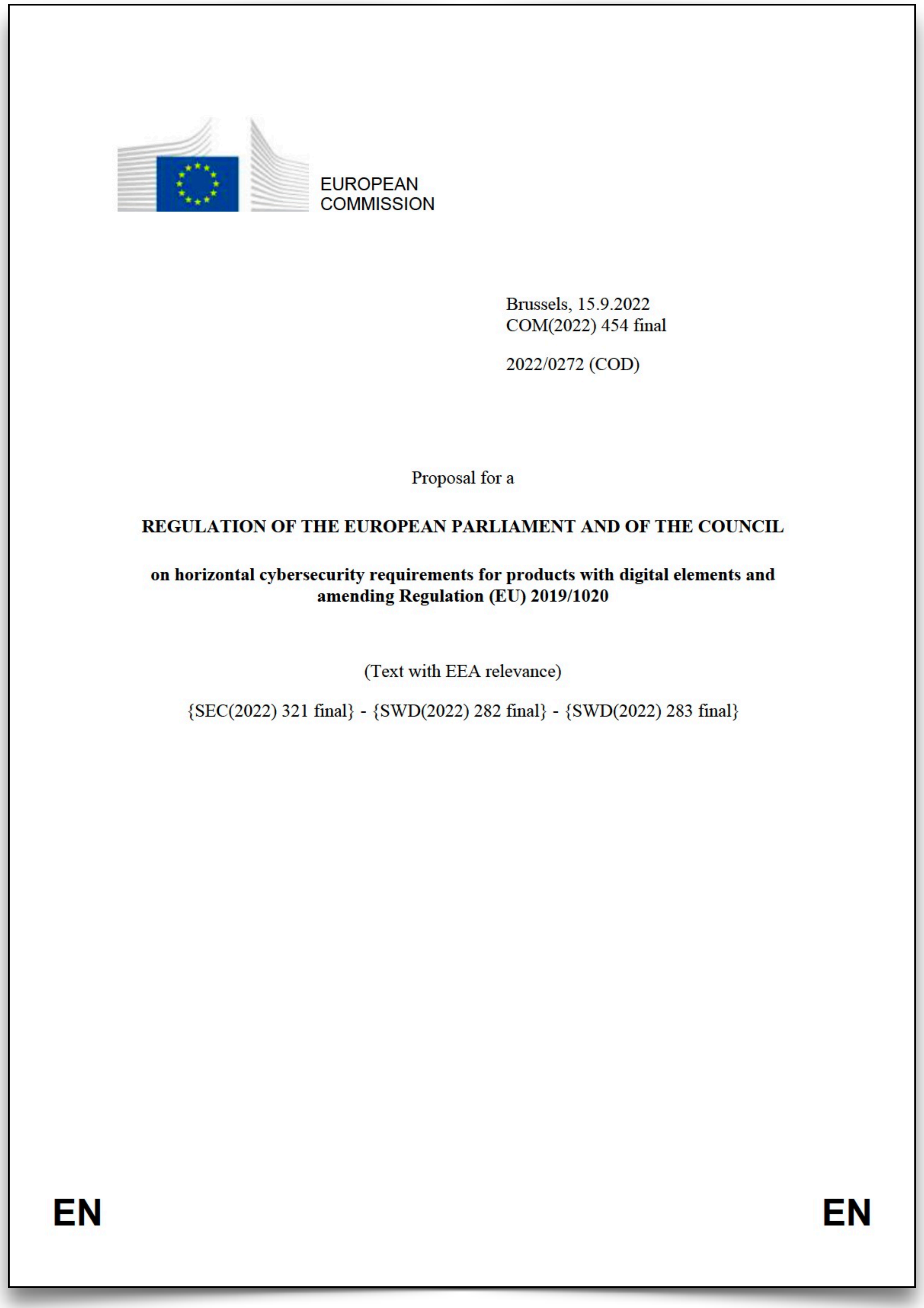
Pillar Three | Shape Market Forces to Drive Security and Resilience

Strategic Objective 3.1: Hold the Stewards of Our Data Accountable

Strategic Objective 3.3: Shift Liability for Insecure Software Products and Services

Strategic Objective 3.5: Leverage Federal Procurement to Improve Accountability

Why Talk About Safety?



EUROPEAN COMMISSION

Brussels, 15.9.2022
COM(2022) 454 final

2022/0272 (COD)

Proposal for a

REGULATION OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL

on horizontal cybersecurity requirements for products with digital elements and amending Regulation (EU) 2019/1020

(Text with EEA relevance)

{SEC(2022) 321 final} - {SWD(2022) 282 final} - {SWD(2022) 283 final}

EN EN

"It is necessary to improve the functioning of the internal market by laying down a uniform legal framework for essential cybersecurity requirements for placing products with digital elements on the Union market."

What is Memory Safety?

> *"**Memory safety** is a broad category of issues related to how a program manages memory."*
>
> *– NSA, Software Memory Safety*

What is Memory Safety?

*"**Memory safety** is the state of being protected from various software bugs and security vulnerabilities when dealing with memory access, such as buffer overflows and dangling pointers."*

"Race condition – concurrent reads/ writes to shared memory"

"Unwanted aliasing – when the same memory location is allocated and modified twice for unrelated purposes."

*"**Memory safety** is the property of a program where memory pointers used always point to valid memory"*

*"...we define a program as being **fully memory safe** if it satisfies the following criteria: it never reads uninitialized memory, performs no illegal operations on the heap (no invalid/double frees), and does not access freed memory (no dangling pointer errors)."*

*"**Memory safety** is a term used by software and security engineers to describe applications that access the operating system's memory in a way that doesn't cause errors."*

What is Safety?

safe·ty [ˈsāftē]

NOUN

1. the condition of being protected from or unlikely to cause danger, risk, or injury

– *Oxford Languages*

What are Safety Properties?

To prove the ***correctness*** of a program, one must prove two essentially different types of properties about it, which we call *safety* and *liveness* properties.

A *safety property* is one which states that something will not happen.

A *liveness property* is one which states that something must happen.

– Leslie Lamport, Proving the Correctness of Multiprocess Programs

What are Safety Properties?

- Safety properties compose
  - If every common execution prefix satisfies *P* then the program satisfies *P*

Example Safety Property for Self Driving Car

- The car cannot drive off the road
  - A safe operation does not allow the car to drive off the road*
  - If all operations are safe, the car cannot drive off the road*

*Assuming the preconditions are not violated

Example Liveness Property for Self Driving Car

- The car will eventually reach its destination
  - This may be achieved by a series of stepwise refinements

Safety in terms of Safety Properties

- The *safety* of a program is a set of properties that cannot happen given valid input.

Safety of a Programming Language

- The *safety* of a programming language is a set of safety properties guaranteed for any ***expressible*** program.
- Every program that can be written in the language (or a safe subset of the language) satisfies the safety properties of the language.

Achieving Safety in a Language

- *limiting expressibility* - the language cannot express code which would violate the safety property.
- *runtime validation* - the program prevents unsafe violation by resulting in an error or termination
- *defined results* - the program defines safe behavior of otherwise unsafe operations

What is a Memory Safe Language?

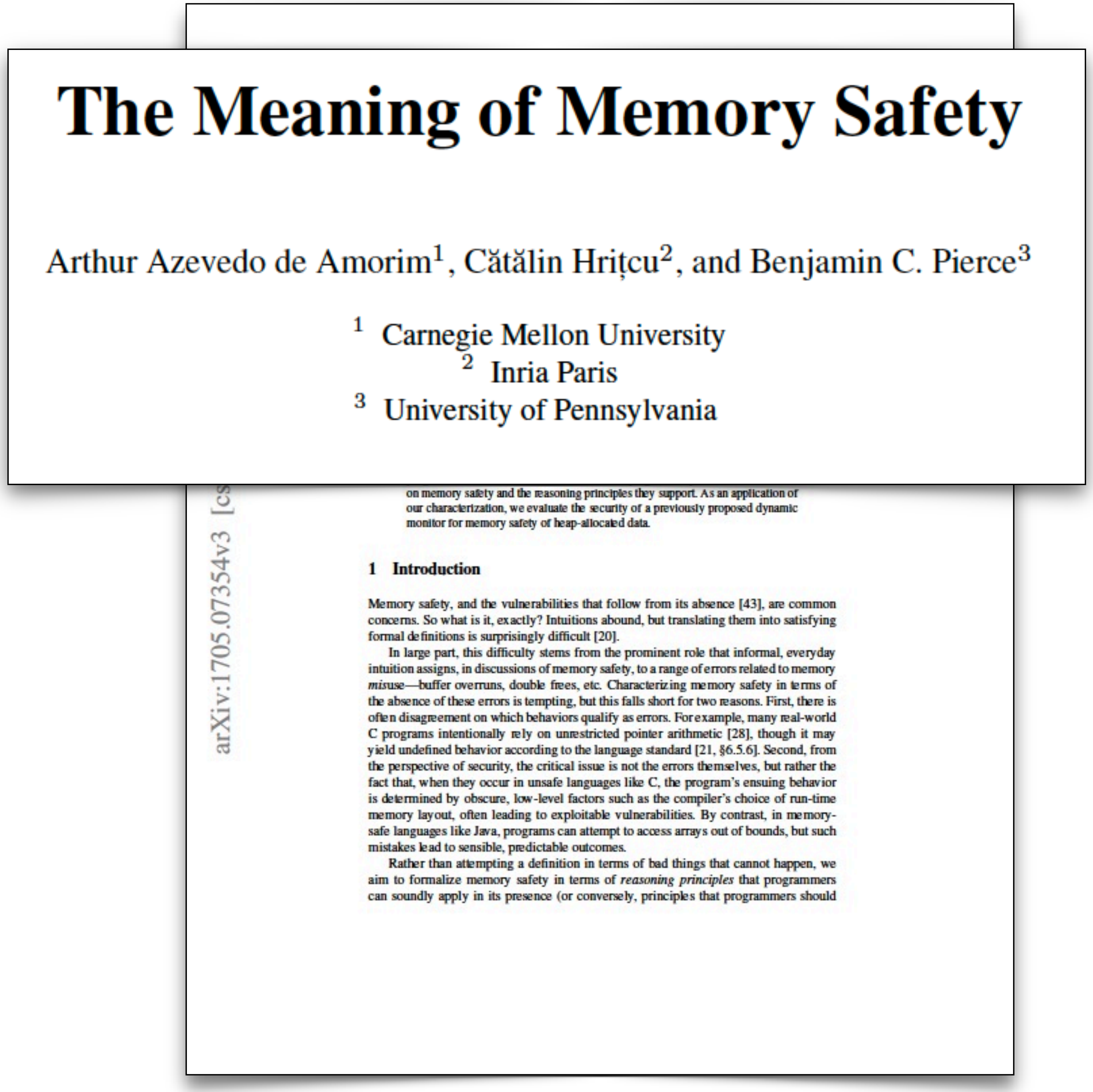
- The language has no operations that access memory
  - which is not allocated
  - has not been initialized
  - has been released

What's the Problem?

- A *secure system* is a system where the resources are used and ***accessed*** as intended under all circumstances
  - Any software defect may violate the security of a system
- Language safety can aid security by:
  - Assisting in the construction of correct software and finding defects sooner
  - Limiting the effect a defect may exhibit including limiting access to other parts of the system
- A key property of a secure system is *noninterference*
  - The *noninterference property* holds if and only if any sequence of low inputs will produce the same low outputs, regardless of what the high level inputs are.

What is a Memory Safe Language?

- *frame rule:* A verified program can only effect a well-defined portion of the state, with all other memory regions left untouched.
  - Inspired by *separation logic* and *local reasoning*
  - Shows that this frame rule satisfies the *noninterference* property
- The frame rule relates to John McCall's *Law of Exclusivity* from Swift



The Meaning of Memory Safety

Arthur Azevedo de Amorim[1], Cătălin Hriţcu[2], and Benjamin C. Pierce[3]

[1] Carnegie Mellon University
[2] Inria Paris
[3] University of Pennsylvania

on memory safety and the reasoning principles they support. As an application of our characterization, we evaluate the security of a previously proposed dynamic monitor for memory safety of heap-allocated data.

arXiv:1705.07354v3 [cs

1 Introduction

Memory safety, and the vulnerabilities that follow from its absence [43], are common concerns. So what is it, exactly? Intuitions abound, but translating them into satisfying formal definitions is surprisingly difficult [20].

In large part, this difficulty stems from the prominent role that informal, everyday intuition assigns, in discussions of memory safety, to a range of errors related to memory *misuse*—buffer overruns, double frees, etc. Characterizing memory safety in terms of the absence of these errors is tempting, but this falls short for two reasons. First, there is often disagreement on which behaviors qualify as errors. For example, many real-world C programs intentionally rely on unrestricted pointer arithmetic [28], though it may yield undefined behavior according to the language standard [21, §6.5.6]. Second, from the perspective of security, the critical issue is not the errors themselves, but rather the fact that, when they occur in unsafe languages like C, the program's ensuing behavior is determined by obscure, low-level factors such as the compiler's choice of run-time memory layout, often leading to exploitable vulnerabilities. By contrast, in memory-safe languages like Java, programs can attempt to access arrays out of bounds, but such mistakes lead to sensible, predictable outcomes.

Rather than attempting a definition in terms of bad things that cannot happen, we aim to formalize memory safety in terms of *reasoning principles* that programmers can soundly apply in its presence (or conversely, principles that programmers should

Memory Safety in C++

- Shared mutable references make memory safety nearly impossible
  - Sanitizers can help
    - But at a performance cost, which often rules them out for production code
    - If not used in production, sanitizers should be combined with fuzzers
  - No safety properties hold in the presence of UB

"Understanding why software fails is important, but the real challenge is understanding why software works."

– Alexander Stepanov

The Illusion of Safety

- If a language is Turing complete we can always construct a C machine and execute all the unsafe code
- Safety can always be circumvented, and often is for performance or expressibility
- Safety is tool for local reasoning

Can C++ be Memory Safe?

- Not without significant changes
  - Will require an explicit safe subset
  - There is a required shift in the abstraction level

Direction

C++Now What?

- Consider a higher level memory-safe(ish) language where performance isn't critical
- For performance critical components, Rust
  - But… no specification, single source tool chain, and poor C++ interoperability

Professors? Grad students?

- I'd like to see an undergrad course on the practical construction of *correct* software
  - Design by Contract
  - Safety Properties
  - Local and Equational Reasoning
  - The Whole/Part Relationship
  - Frame Rule (Law of Exclusivity)
  - Error Handling

Better Code

- Rubric: No raw synchronization primitives
  - Mutexes

Mutexes

[Note 21: It can be shown that programs that *correctly* use mutexes and memory_order :: seq_cst operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as “sequential consistency”. However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result has undefined behavior.
— end note]

– The C++ Standard

Mutexes

[Note 21: It can be shown that **programs that *correctly* use mutexes** and memory_order :: seq_cst operations to prevent all data races and use no other synchronization operations **behave as if the operations executed** by their constituent threads **were simply interleaved**, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since **any program that behaves differently** as a result **has undefined behavior**.
— end note]

– The C++ Standard

Safety Properties of Mutexes

- Assists
  - No Race Conditions

Safety Properties of Mutexes

- Assists
  - ~~No Race Conditions~~
- Hinders
  - No Deadlocks
    - Invoking a mutex, while satisfying all preconditions, may deadlock

Safe Alternative to Mutex

- Serial Queue
  - Faster - non-blocking, no forced context switch
  - Non-trivial transformation - requires restructuring code, often with continuations

Better Code

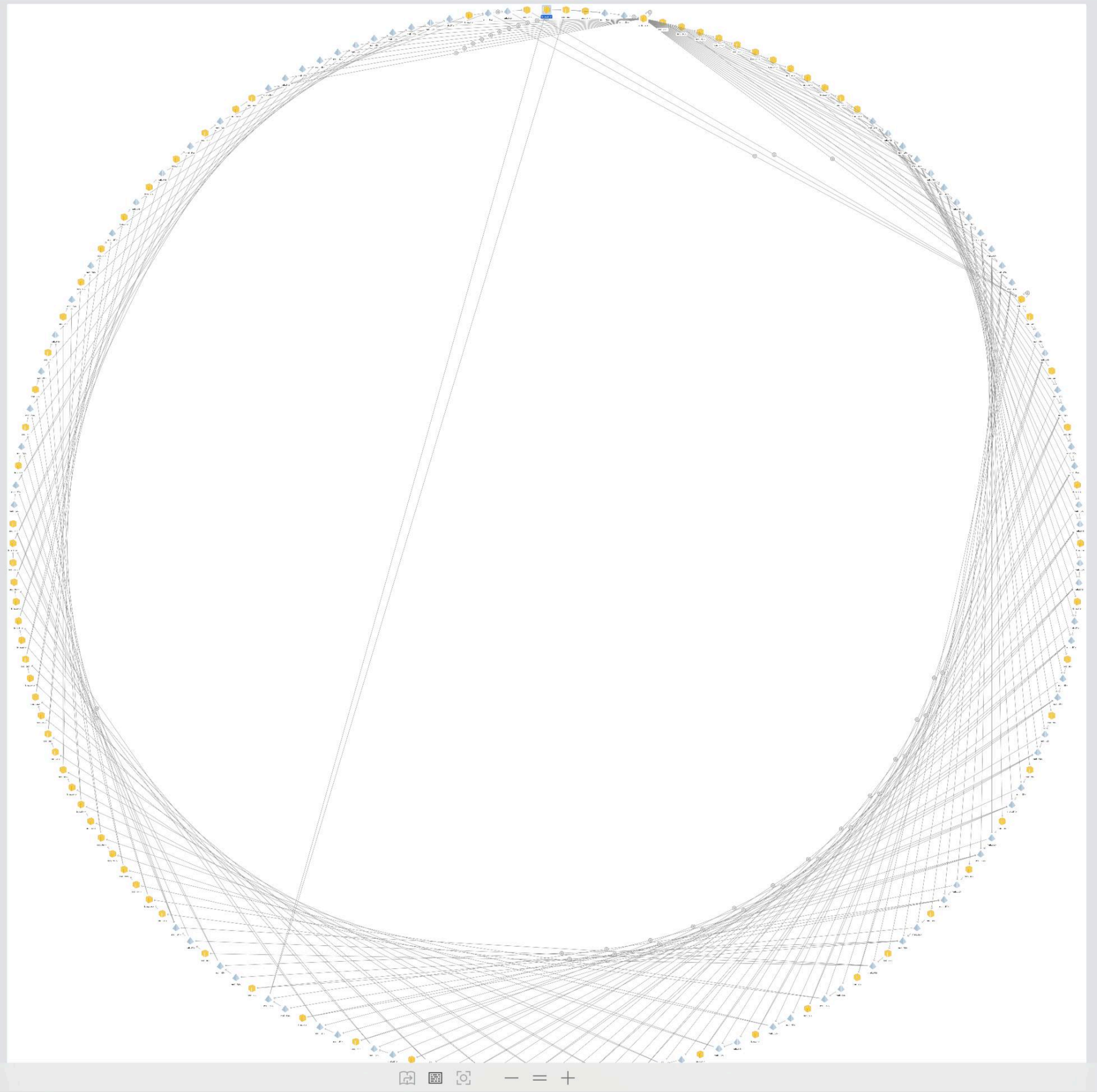
- Rubric: No raw pointers
  - Shared Pointers

Safety Properties of Shared Pointers

- Assists
  - Object Lifetime
  - No Memory Leaks

Safety Properties of Shared Pointers

- Assists
  - Object Lifetime
  - ~~No Memory Leaks~~
- Hinders
  - No Memory Leaks
  - No Race Conditions
  - No Observation of Broken Class Invariants
  - Noninterference Principal - Violation of Frame Rule



Safe Alternative to Shared Pointer

- (Mutable) Value Semantics and Whole/Part relationships
  - Easier to reason about
  - Requires difficult code transformations

Better Code

- Rubric: No raw loops
  - while, do, for

Safety Properties of Loops

- Assists
- Hinders
  - Termination
  - Time Constraint
  - Bounds
    - Noninterference Principal - Violation of Frame Rule

Safe Alternative to Raw Loops

- Algorithms
  - Easier to reason about, faster
  - No downside

Forward

A (New) Possible Future

- A large library of proven generic components
- A small number of non-Turing complete declarative *forms* for assembling the generic components

Dafny

```
method Reverse(a: seq<int>) returns (r: seq<int>)
  ensures |a| == |r|
  // ensures forall i :: 0 <= i < |a| ==> r[i] == a[|a| - i - 1]
{
  var remainder := a;
  r := [];
  while (|remainder| > 1)
    invariant |remainder| == |a| - |r|
    invariant forall i :: 0 <= i < |r| ==> r[i] == a[|r| - i - 1]
              ^^^^^^ This loop invariant might not be maintained by the loop.
 {
    r := [remainder[0]] + r;
    remainder := remainder[1..];
  }
  r := remainder + r;
}
```

Bing Chat

"How can I strengthen the loop invariant in the following Dafny code to convince Dafny that the loop invariant holds?"

"You could add an invariant that ensures that the elements of `remainder` are the same as the elements of `a` that have not yet been added to `r`. Here's an example:

```
invariant forall i :: 0 <= i < |remainder| ==> remainder[i] == a[|r| + i]
```

This invariant states that for all indices `i` in the range `[0, |remainder|)`, the `i`-th element of `remainder` is equal to the `(i + |r|)`-th element of `a`. This ensures that the elements of `remainder` are exactly the elements of `a` that have not yet been added to `r`."

Dafny

```
method Reverse(a: seq<int>) returns (r: seq<int>)
  ensures |a| == |r|
  ensures forall i :: 0 <= i < |a| ==> r[i] == a[|a| - i - 1]
{
  var remainder := a;
  r := [];
  while (|remainder| > 1)
    invariant |remainder| == |a| - |r|
    invariant forall i :: 0 <= i < |r| ==> r[i] == a[|r| - i - 1]
    invariant forall i :: 0 <= i < |remainder| ==> remainder[i] == a[|r| + i]
  {
    r := [remainder[0]] + r;
    remainder := remainder[1..];
  }
  r := remainder + r;
}
```

A (New) Possible Future

- A large library of proven generic components
- A small number of non-Turing complete declarative *forms* for assembling the generic components
- Built with AI assisted verification

Closing

- Memory safety is important
- Use safety properties so your code doesn't do bad things
- Don't lose sight that correctness is the goal

About the artist

MUE Studio

MUE Studio in New York City, a collaboration of Minjiin Kang and Mijoo Kim, creates visual experiences through 3D image design and photography. Drawing inspiration from the architecture and culture they see around them every day, the duo strive to blur the boundary between fantasy and reality in their work. For this piece, they used Adobe Photoshop and Cinema 4D to build a dreamlike space that connects emotionally with viewers and offers them an escape.

Made with

Adobe Photoshop


Adobe
Bē
Artwork by MUE Studio