Keynote: All the Safeties

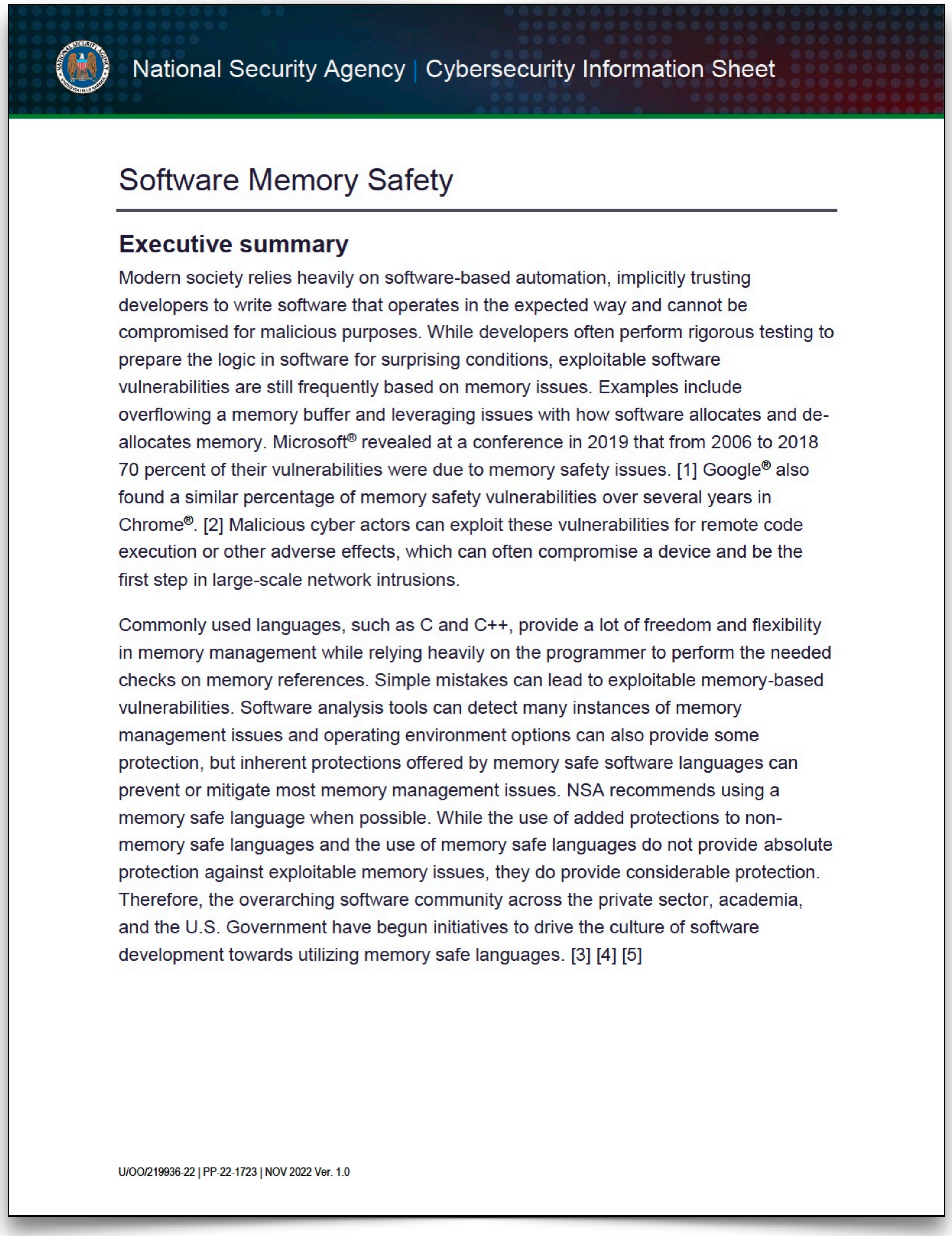
Sean Parent

2023

All the Safeties

Sean Parent | Sr. Principal Scientist
Adobe Software Technology Lab

Why Talk About Safety?



National Security Agency | Cybersecurity Information Sheet

Software Memory Safety

Executive summary

Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

Commonly used languages, such as C and C++, provide a lot of freedom and flexibility in memory management while relying heavily on the programmer to perform the needed checks on memory references. Simple mistakes can lead to exploitable memory-based vulnerabilities. Software analysis tools can detect many instances of memory management issues and operating environment options can also provide some protection, but inherent protections offered by memory safe software languages can prevent or mitigate most memory management issues. NSA recommends using a memory safe language when possible. While the use of added protections to non-memory safe languages and the use of memory safe languages do not provide absolute protection against exploitable memory issues, they do provide considerable protection. Therefore, the overarching software community across the private sector, academia, and the U.S. Government have begun initiatives to drive the culture of software development towards utilizing memory safe languages. [3] [4] [5]

U/OO/219936-22 | PP-22-1723 | NOV 2022 Ver. 1.0

Why Talk About Safety?

National Security Agency | Cybersecurity Information Sheet

Software Memory Safety

Executive summary

Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

Commonly used languages, such as C and C++, provide a lot of freedom and flexibility in memory management while relying heavily on the programmer to perform the needed checks on memory references. Simple mistakes can lead to exploitable memory-based vulnerabilities. Software analysis tools can detect many instances of memory management issues and operating environment options can also provide some protection, but inherent protections offered by memory safe software languages can prevent or mitigate most memory management issues. NSA recommends using a memory safe language when possible. While the use of added protections to non-memory safe languages and the use of memory safe languages do not provide absolute protection against exploitable memory issues, they do provide considerable protection. Therefore, the overarching software community across the private sector, academia, and the U.S. Government have begun initiatives to drive the culture of software development towards utilizing memory safe languages. [3] [4] [5]

U/OO/219936-22 | PP-22-1723 | NOV 2022 Ver. 1.0

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

Why Talk About Safety?

National Security Agency | Cybersecurity Information Sheet

Software Memory Safety

Executive summary

Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

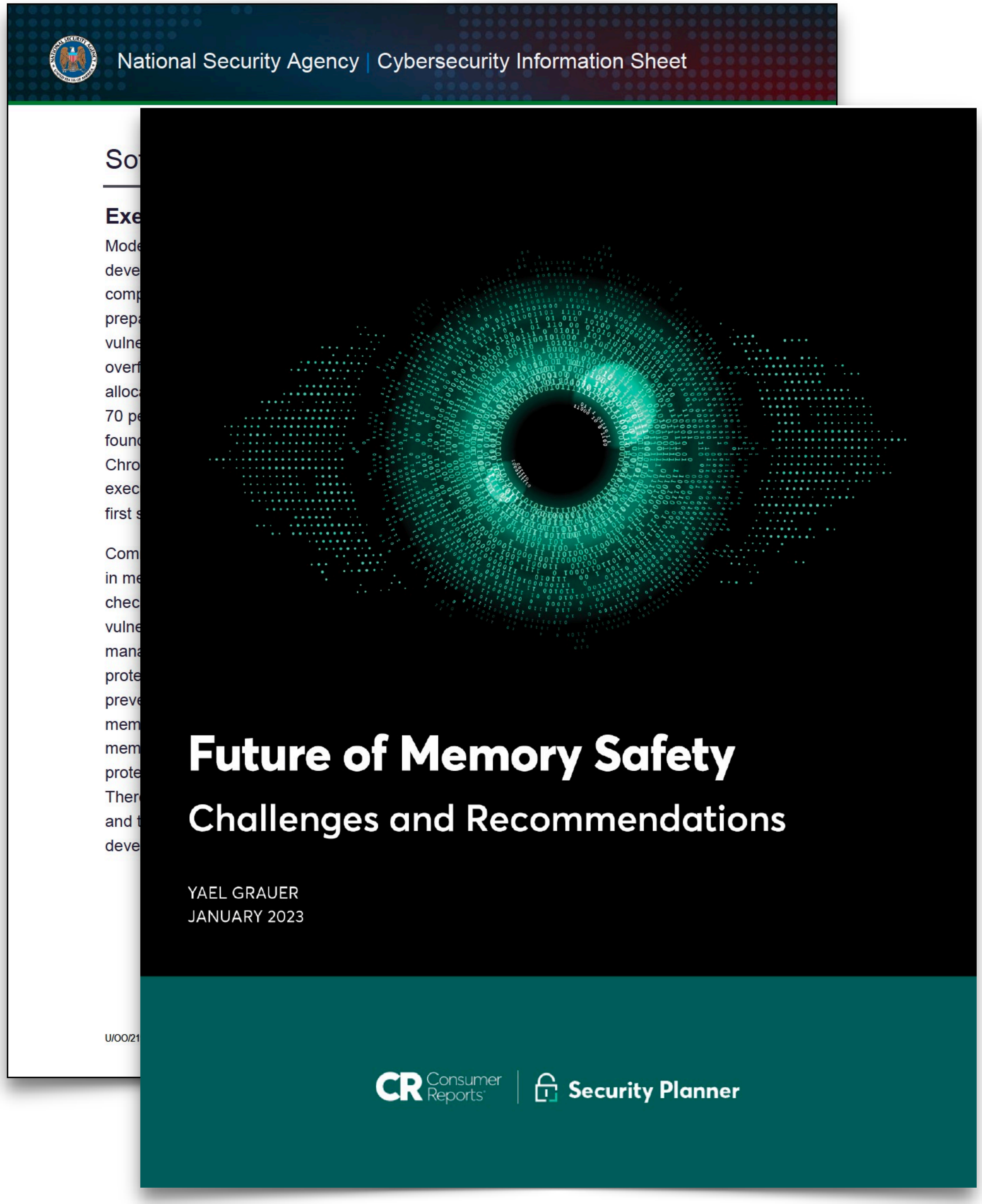
Commonly used languages, such as C and C++, provide a lot of freedom and flexibility in memory management while relying heavily on the programmer to perform the needed checks on memory references. Simple mistakes can lead to exploitable memory-based vulnerabilities. Software analysis tools can detect many instances of memory management issues and operating environment options can also provide some protection, but inherent protections offered by memory safe software languages can prevent or mitigate most memory management issues. NSA recommends using a memory safe language when possible. While the use of added protections to non-memory safe languages and the use of memory safe languages do not provide absolute protection against exploitable memory issues, they do provide considerable protection. Therefore, the overarching software community across the private sector, academia, and the U.S. Government have begun initiatives to drive the culture of software development towards utilizing memory safe languages. [3] [4] [5]

U/OO/219936-22 | PP-22-1723 | NOV 2022 Ver. 1.0

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

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

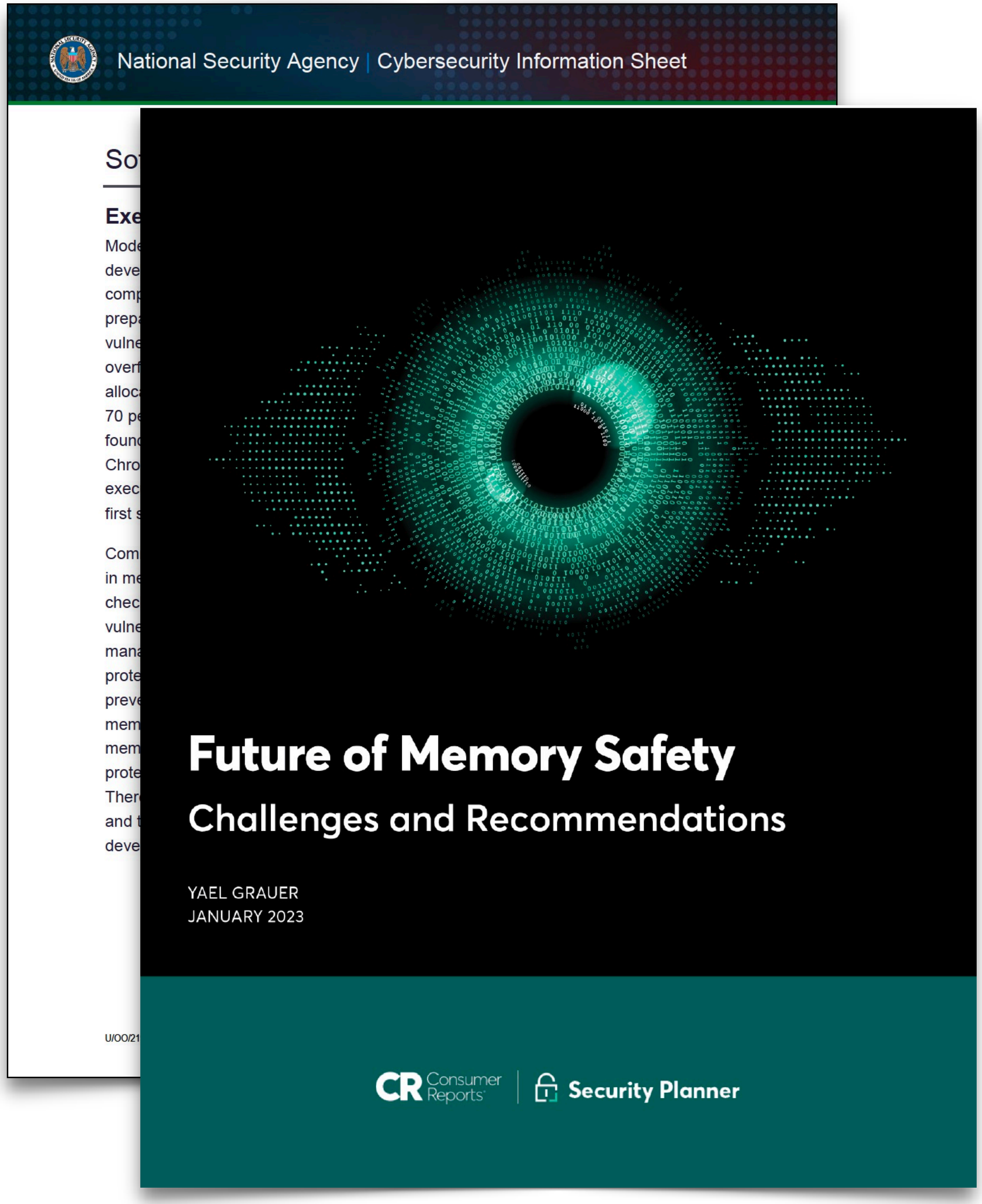
Why Talk About Safety?



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

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

Why Talk About Safety?

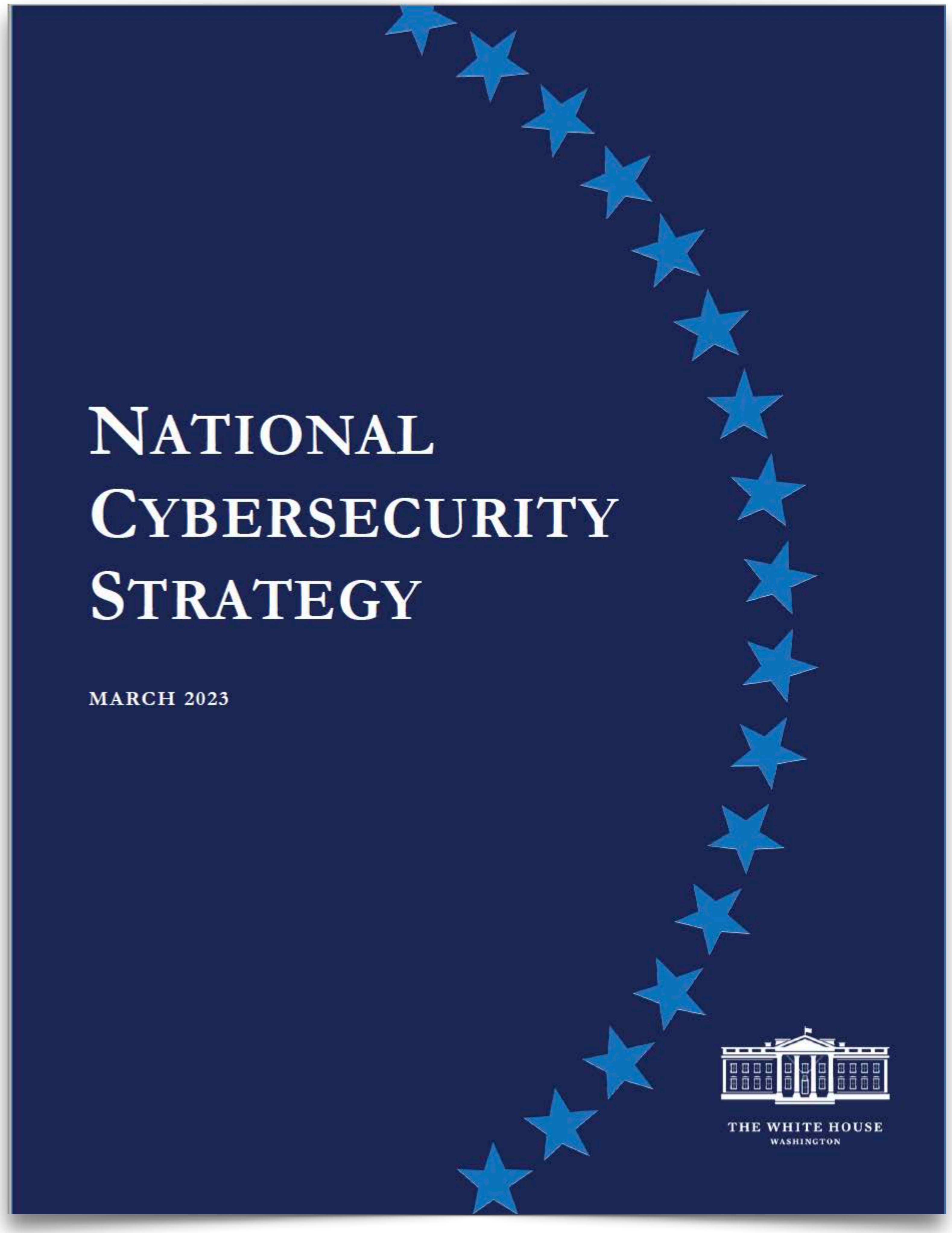


"70 percent of their vulnerabilities were due to ***memory safety issues****"*

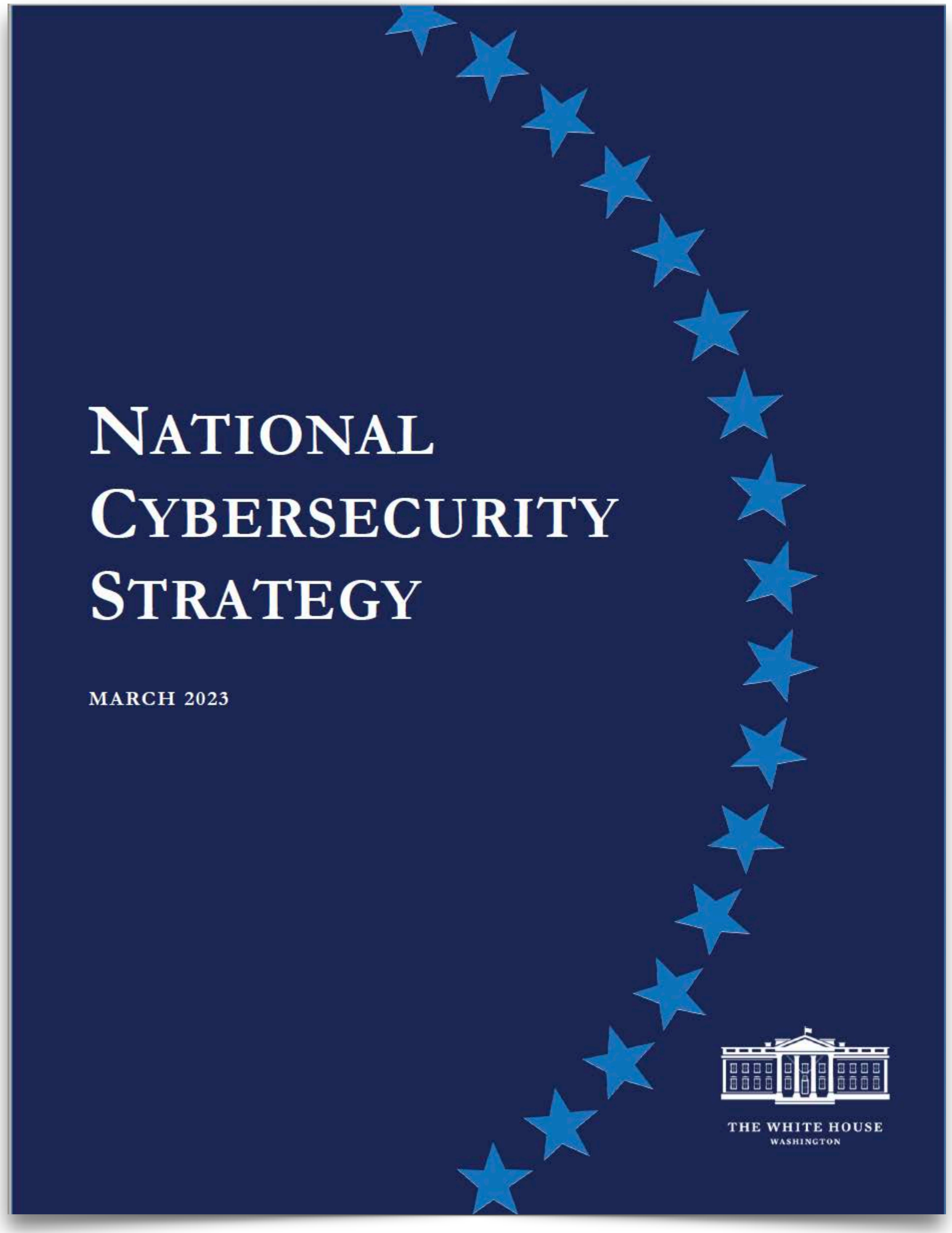
"NSA recommends using a ***memory safe language*** *when possible."*

"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?

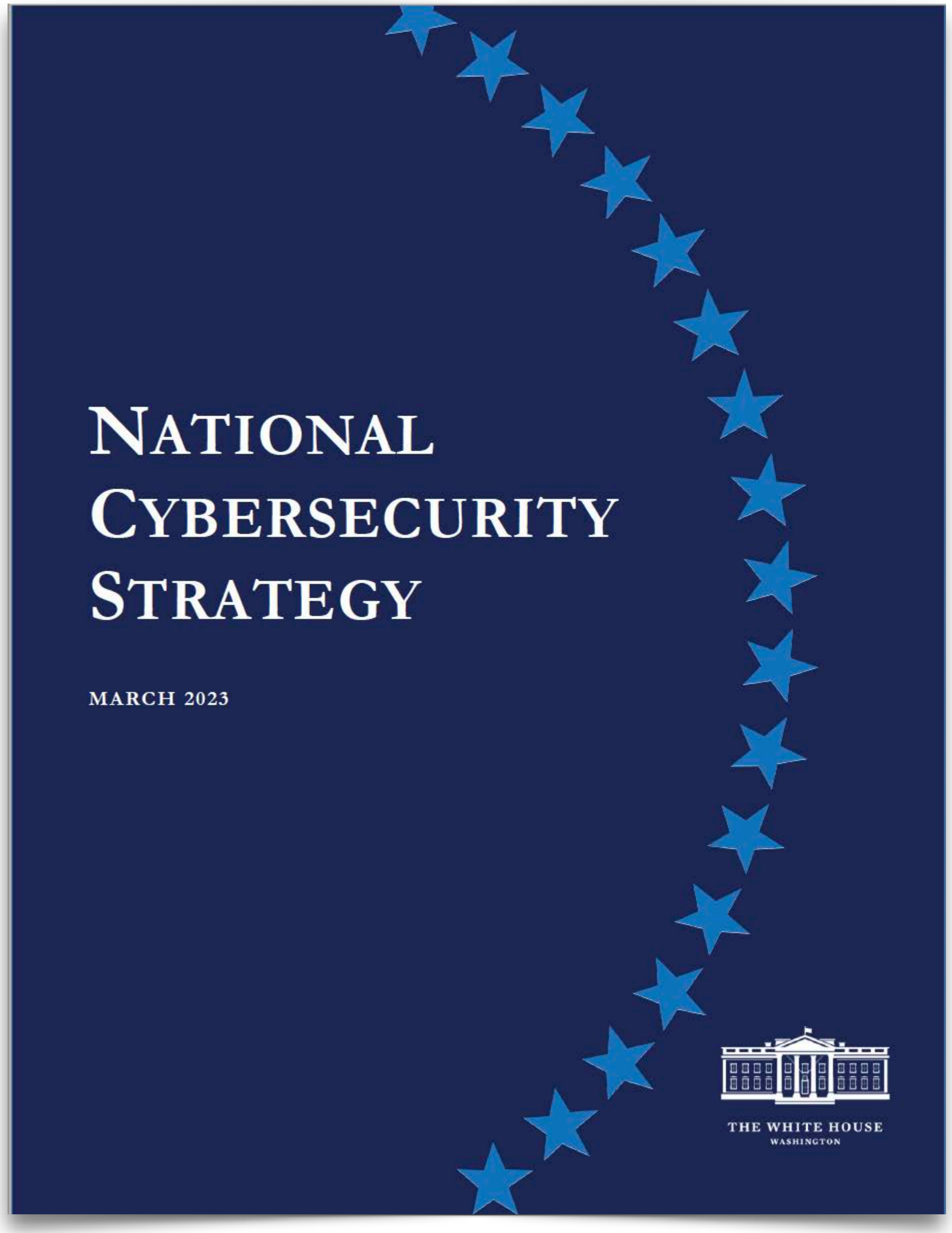


Why Talk About Safety?



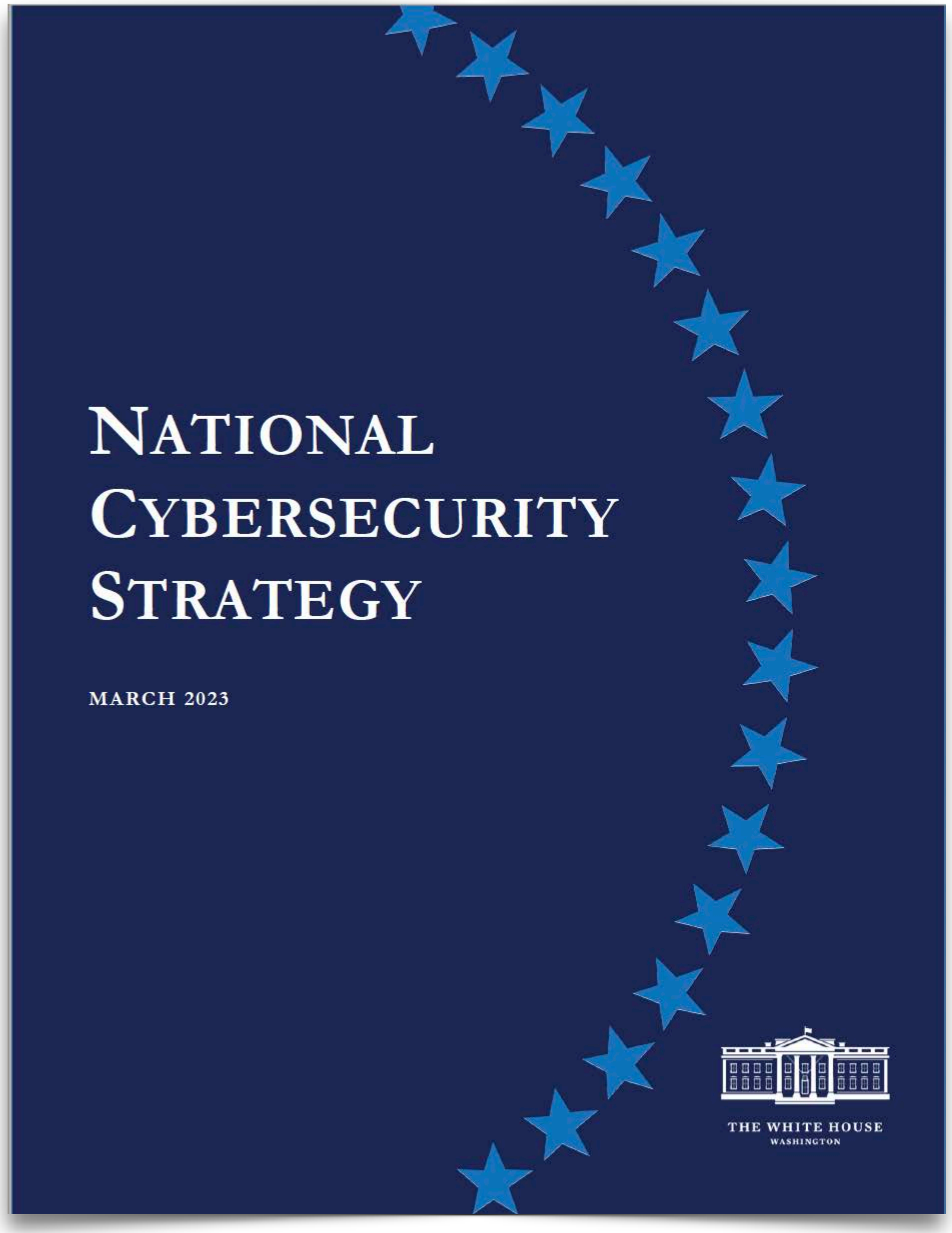
"Cybersecurity is essential to the basic functioning of our economy, the operation of our critical infrastructure, the strength or our democracy and democratic institutions, the privacy of our data and communications, and our national defense."

Why Talk About Safety?



Pillar Three | Shape Market Forces to Drive Security and Resilience

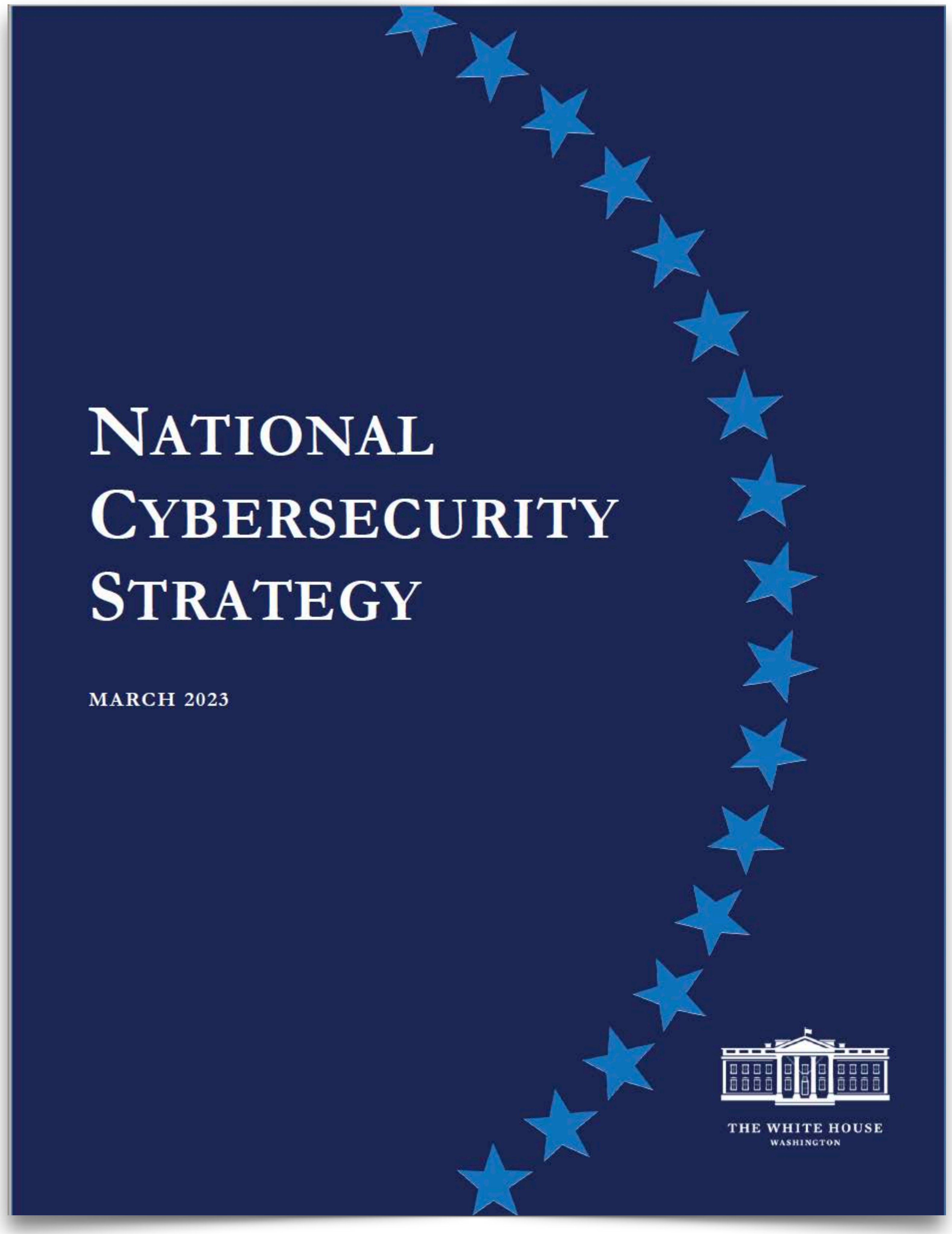
Why Talk About Safety?



Pillar Three | Shape Market Forces to Drive Security and Resilience

Strategic Objective 3.1: Hold the Stewards of Our Data Accountable

Why Talk About Safety?

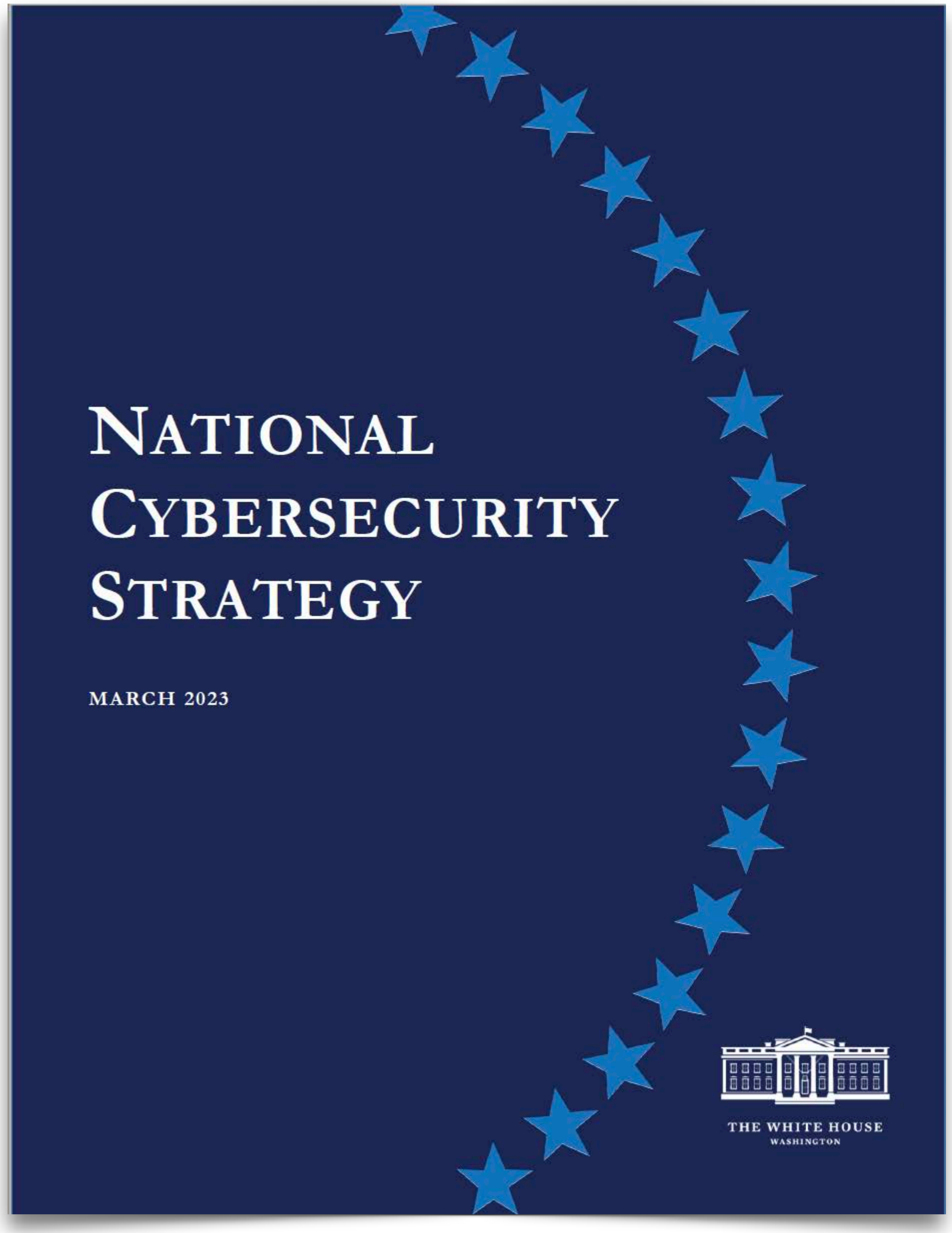


Pillar Three | Shape Market Forces to Drive Security and Resilience

Strategic Objective 3.1: Hold the Stewards of Our Data Accountable

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

Why Talk About Safety?



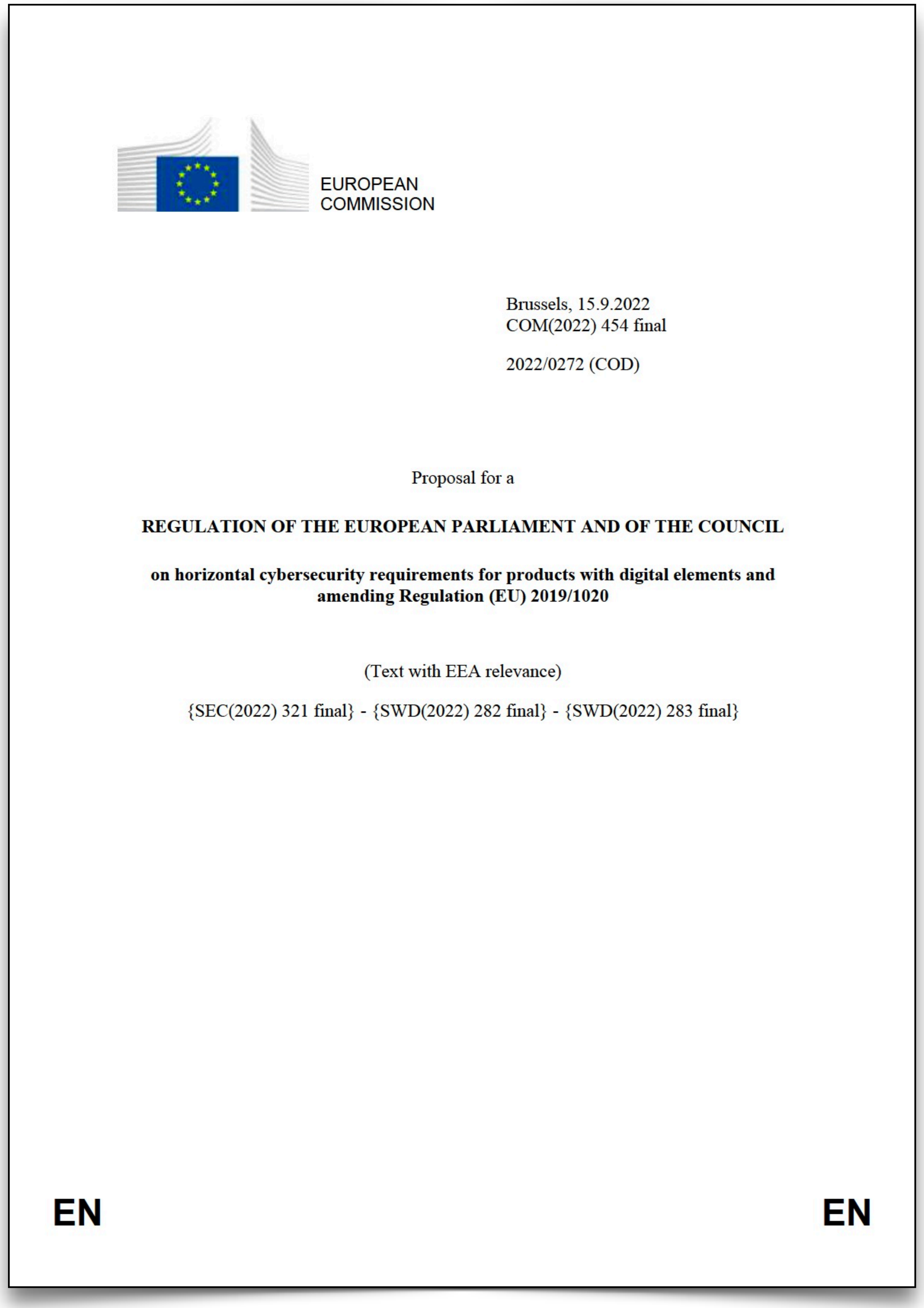
Pillar Three | Shape Market Forces to Drive Security and Resilience

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

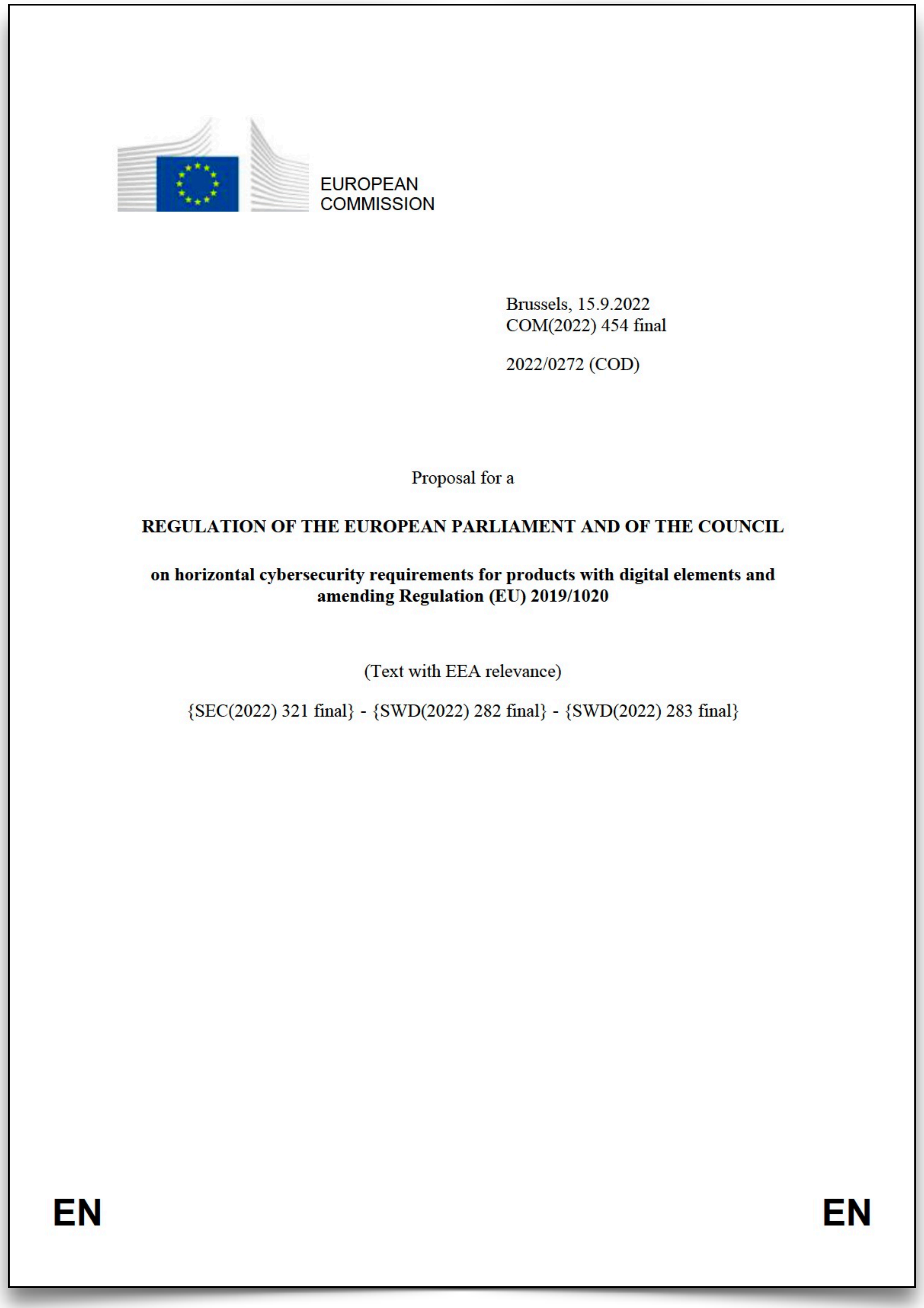
on horizontal cybersecurity requirements for products with digital elements and amending Regulation (EU) 2019/1020

(Text with EEA relevance)

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

EN EN

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?

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?

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."*

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."*

*"...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)."*

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."*

*"**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)."*

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."*

*"**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 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"

*"**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 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."*

Safety in terms of Safety Properties

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

What are Safety Properties?

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?

If some execution of an operation does not satisfy a safety property, then the defining *bad thing* occurs at some point in the operation

What are Safety Properties?

If some execution of an operation does not satisfy a safety property, then the defining *bad thing* occurs at some point in the operation

Safety properties are *irremediable*

What are Safety Properties?

If some execution of an operation does not satisfy a safety property, then the defining *bad thing* occurs at some point in the operation

Safety properties are *irremediable*

- If the execution continues, with subsequent operations, the bad thing happened within the composition

What are Safety Properties?

If some execution of an operation does not satisfy a safety property, then the defining *bad thing* occurs at some point in the operation

Safety properties are *irremediable*

- If the execution continues, with subsequent operations, the bad thing happened within the composition

Safety properties compose

What are Safety Properties?

If some execution of an operation does not satisfy a safety property, then the defining *bad thing* occurs at some point in the operation

Safety properties are *irremediable*

- If the execution continues, with subsequent operations, the bad thing happened within the composition

Safety properties compose

- If every common operation satisfies a safety property, every composition of those operations also satisfies the safety property

Example Safety Property for Self Driving Car

The car cannot drive off the road

Example Safety Property for Self Driving Car

The car cannot drive off the road

- A safe operation does not allow the car to go off the road*

Example Safety Property for Self Driving Car

The car cannot drive off the road

- A safe operation does not allow the car to go off the road*
- If all operations can be shown to satisfy the safety property, then the condition prevented by the property cannot occur*

Example Safety Property for Self Driving Car

The car cannot drive off the road

- A safe operation does not allow the car to go off the road*
- If all operations can be shown to satisfy the safety property, then the condition prevented by the property cannot occur*

*Assuming the preconditions are not violated

Example Liveness Property for Self Driving Car

Example Liveness Property for Self Driving Car

The car will eventually reach its destination

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 of a Programming Language

The *safety* of a programming language is a set of safety properties guaranteed for any ***expressible*** program.

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.

Achieving Safety in a Language

limiting expressibility - the language cannot express code which would violate the safety property.

runtime validation - the program prevents a violation by resulting in an error or termination

Achieving Safety in a Language

limiting expressibility - the language cannot express code which would violate the safety property.

runtime validation - the program prevents a 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 is a Memory Safe Language?

The language has no operations that access memory

- which is not allocated
- has not been initialized
- has been released

Memory Safe?

```
void set_element(int array[], int index) {
    array[index] = 42;
}

int main() {
    int array[]{1, 2, 3};
    int what = 0;

    set_element(array, 3);

    cout << what << "\n";
}
```

Memory Safe?

```
void set_element(int array[], int index) {
    array[index] = 42;
}

int main() {
    int array[]{1, 2, 3};
    int what = 0;

    set_element(array, 3);

    cout << what << "\n";
}
```

42

What's the Problem?

A secure system is a system where the resources are used and accessed as intended under all circumstances

What's the Problem?

A secure system is a system where the resources are used and accessed as intended under all circumstances

- Any defect can cause resources to be accessed in unintended ways

What's the Problem?

A secure system is a system where the resources are used and accessed as intended under all circumstances

- Any defect can cause resources to be accessed in unintended ways

Language safety can aid security by:

What's the Problem?

A secure system is a system where the resources are used and accessed as intended under all circumstances

- Any defect can cause resources to be accessed in unintended ways

Language safety can aid security by:

- Making defects harder to write and easier to see

What's the Problem?

A secure system is a system where the resources are used and accessed as intended under all circumstances

- Any defect can cause resources to be accessed in unintended ways

Language safety can aid security by:

- Making defects harder to write and easier to see
- Containing the damage due to a defect

What's the Problem?

A secure system is a system where the resources are used and accessed as intended under all circumstances

- Any defect can cause resources to be accessed in unintended ways

Language safety can aid security by:

- Making defects harder to write and easier to see
- Containing the damage due to a defect

A key property of a secure system is noninterference

What's the Problem?

A secure system is a system where the resources are used and accessed as intended under all circumstances

- Any defect can cause resources to be accessed in unintended ways

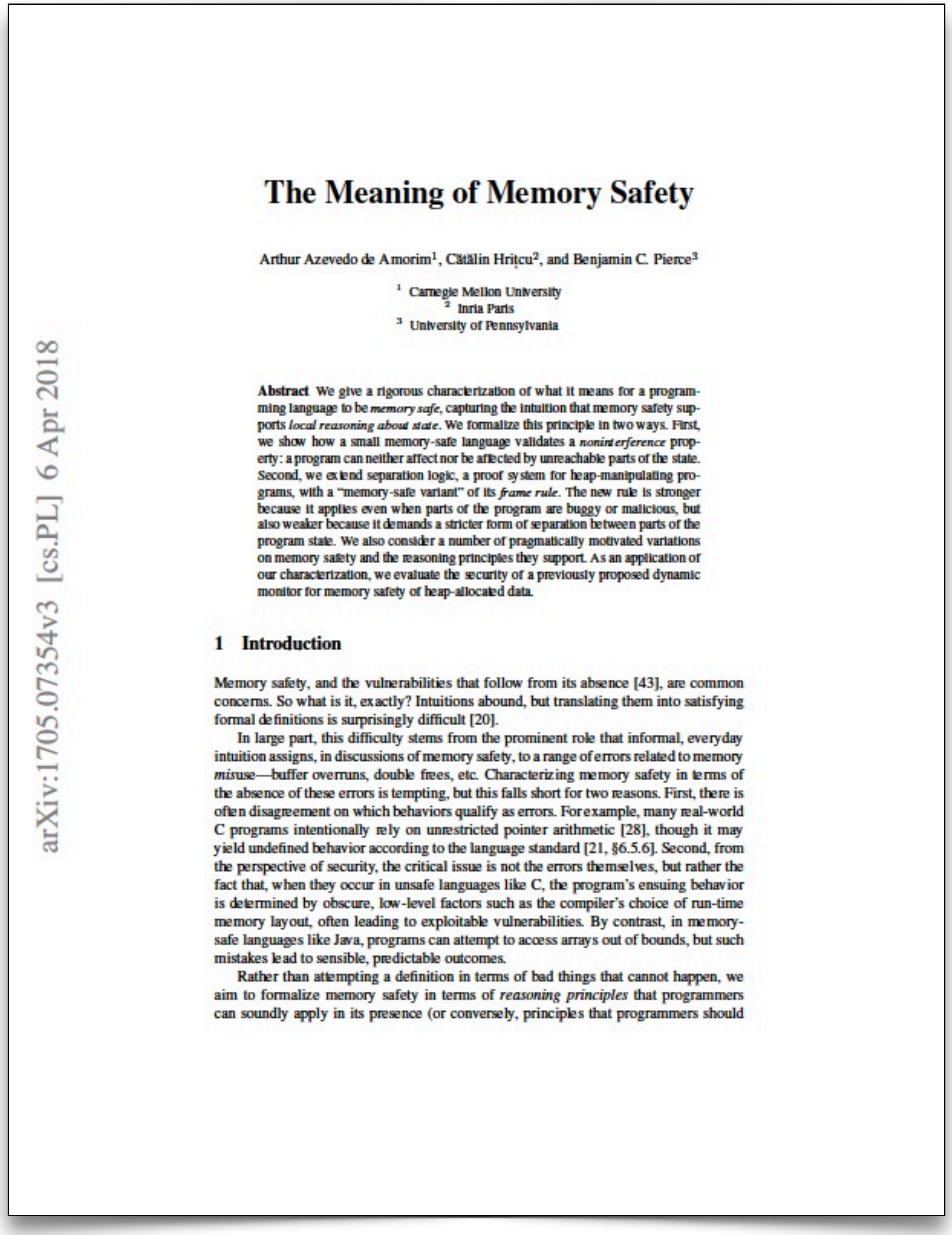
Language safety can aid security by:

- Making defects harder to write and easier to see
- Containing the damage due to a defect

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 inputs are

What is a Memory Safe Language?



arXiv:1705.07354v3 [cs.PL] 6 Apr 2018

The Meaning of Memory Safety

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

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

Abstract We give a rigorous characterization of what it means for a programming language to be *memory safe*, capturing the intuition that memory safety supports *local reasoning about state*. We formalize this principle in two ways. First, we show how a small memory-safe language validates a *noninterference* property: a program can neither affect nor be affected by unreachable parts of the state. Second, we extend separation logic, a proof system for heap-manipulating programs, with a "memory-safe variant" of its *frame rule*. The new rule is stronger because it applies even when parts of the program are buggy or malicious, but also weaker because it demands a stricter form of separation between parts of the program state. We also consider a number of pragmatically motivated variations 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.

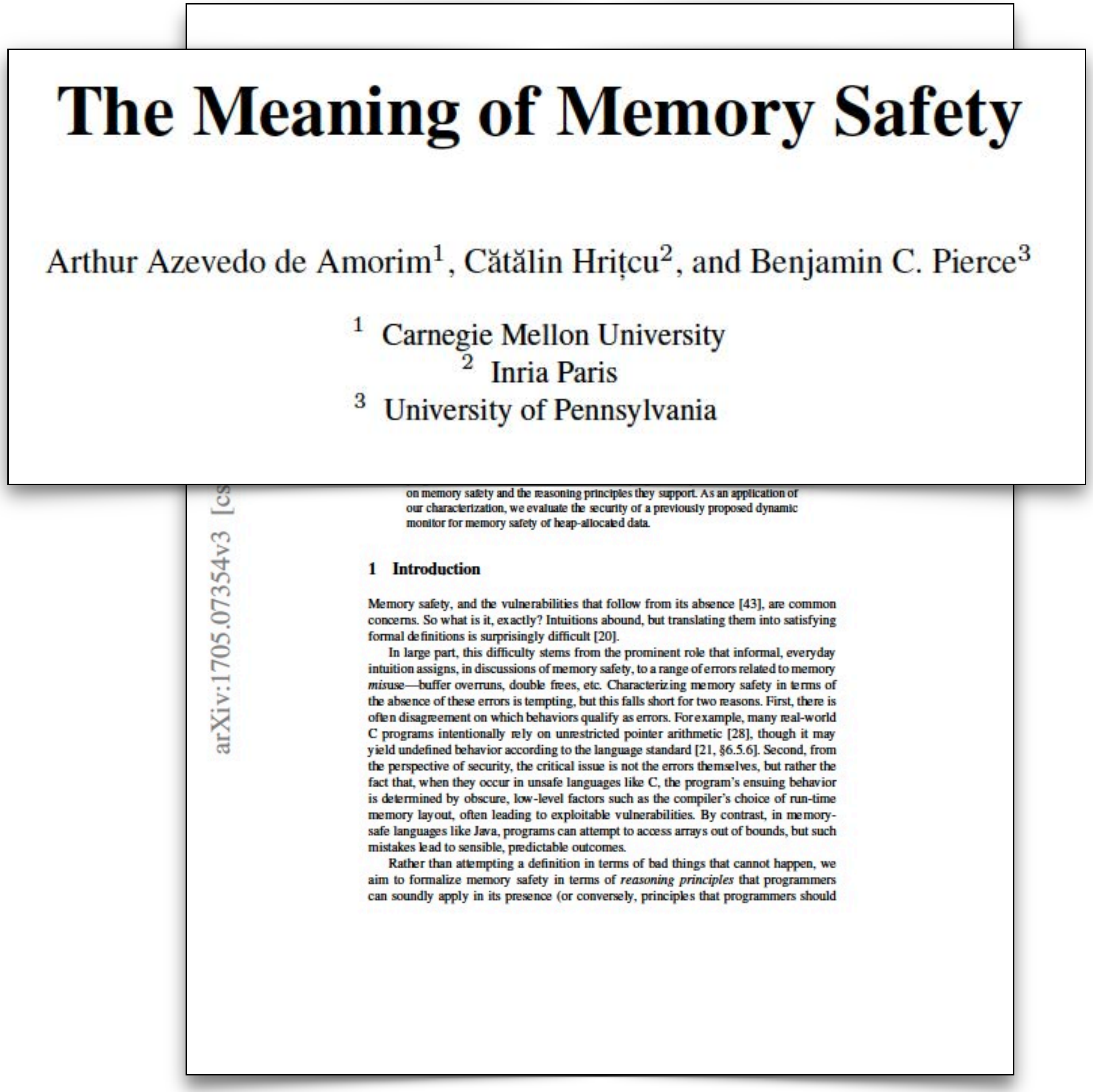
1 Introduction

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

What is a Memory Safe Language?



The Meaning of Memory Safety

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

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

arXiv:1705.07354v3 [cs

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.

1 Introduction

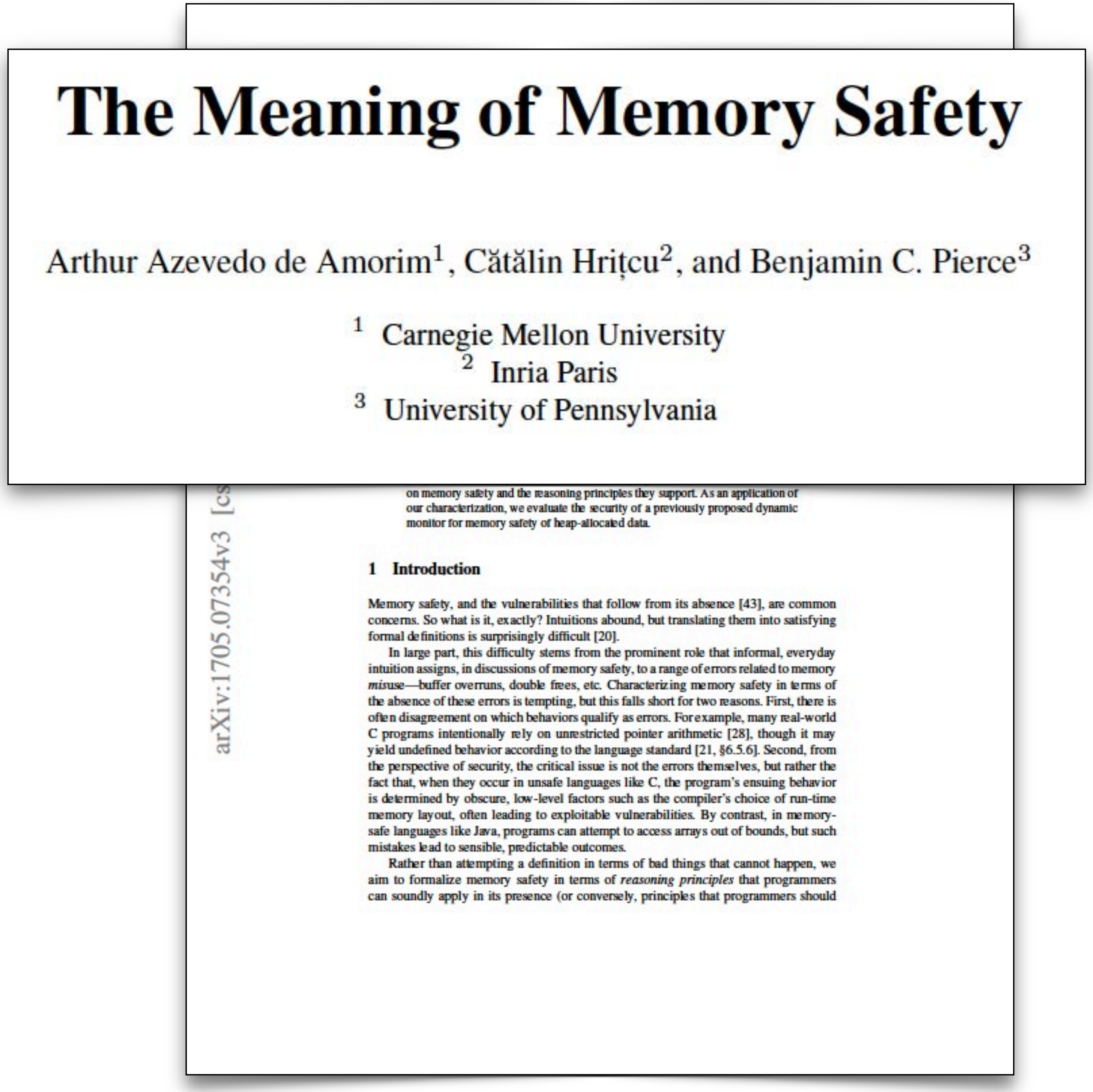
Memory safety, and the vulnerabilities that follow from its absence [43], are common concerns. So what is it, exactly? Intuitions abound, but translating them into satisfying formal definitions is surprisingly difficult [20].

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

What is a Memory Safe Language?

frame rule: A verified program can only affect a well-defined portion of the state, leaving all other memory regions untouched



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.

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].

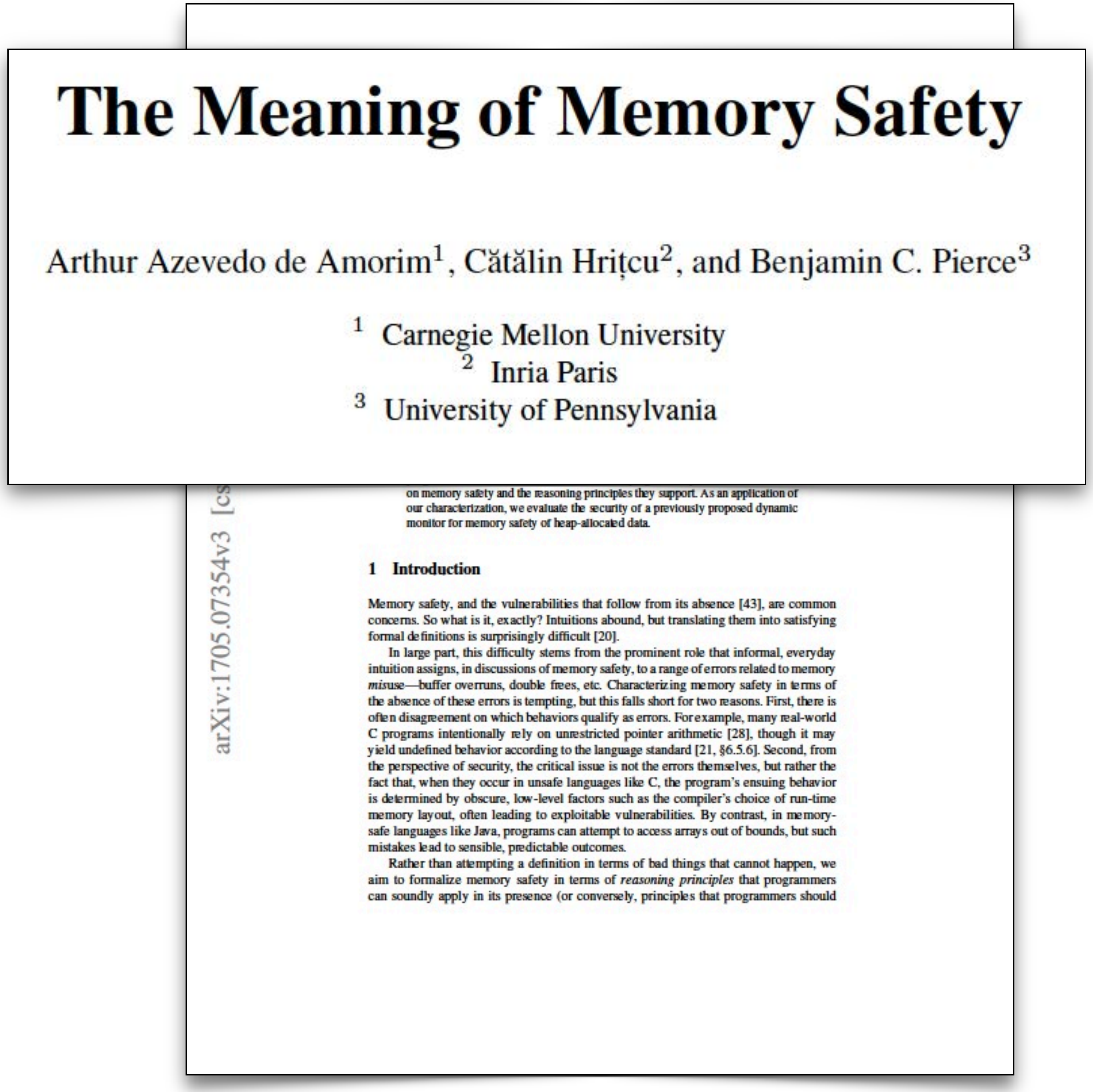
In large part, this difficulty stems from the prominent role that informal, everyday intuition assigns, in discussions of memory safety, to a range of errors related to memory *misuse*—buffer overruns, double frees, etc. Characterizing memory safety in terms of the absence of these errors is tempting, 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

What is a Memory Safe Language?

frame rule: A verified program can only affect a well-defined portion of the state, leaving all other memory regions untouched

- Inspired by separation logic and local reasoning



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

arXiv:1705.07354v3 [cs

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.

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].

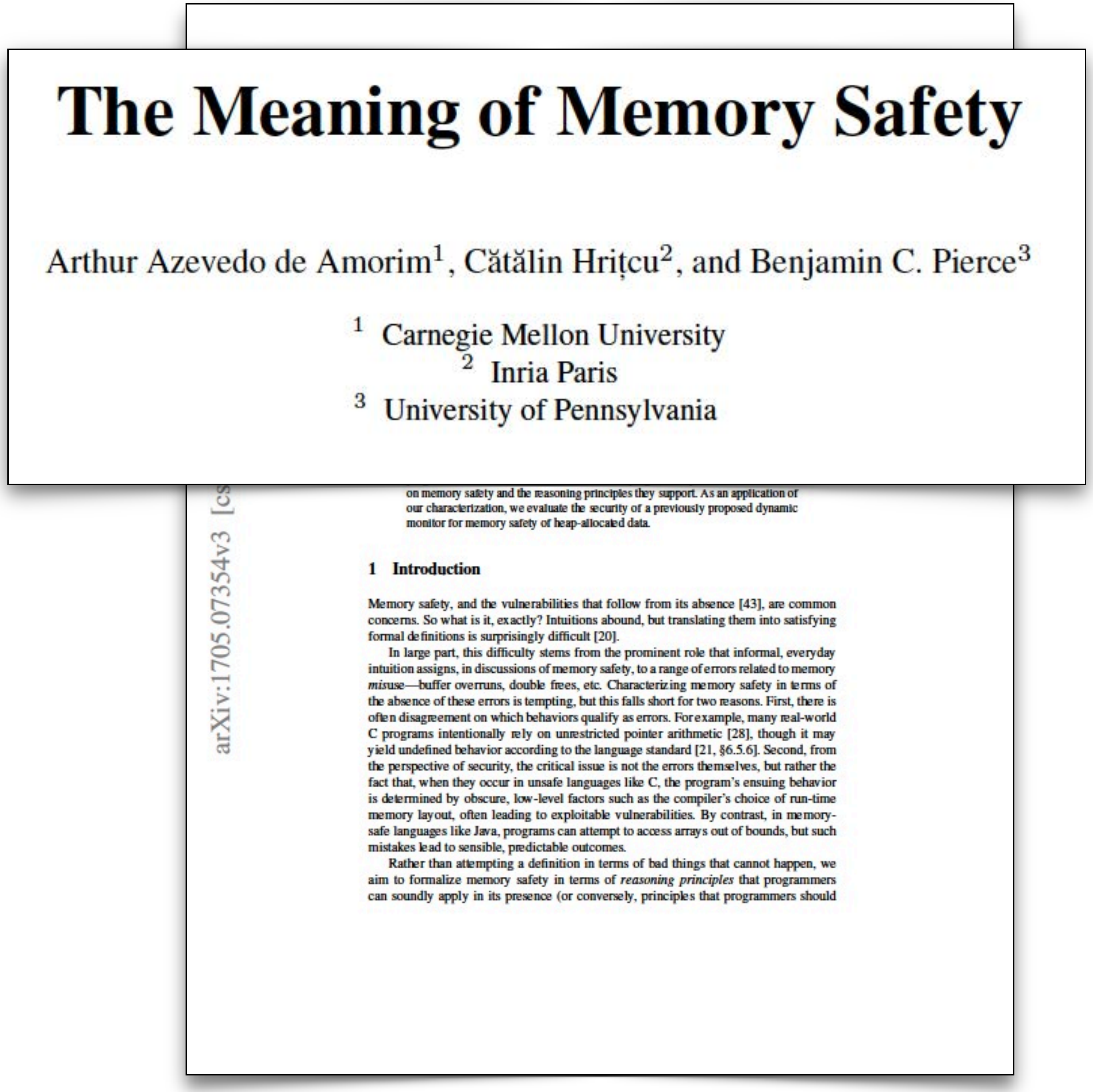
In large part, this difficulty stems from the prominent role that informal, everyday intuition assigns, in discussions of memory safety, to a range of errors related to memory *misuse*—buffer overruns, double frees, etc. Characterizing memory safety in terms of the absence of these errors is tempting, 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

What is a Memory Safe Language?

frame rule: A verified program can only affect a well-defined portion of the state, leaving all other memory regions untouched

- Inspired by separation logic and local reasoning
- The frame rule satisfies the noninterference property



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

arXiv:1705.07354v3 [cs

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.

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 *mis*use—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.

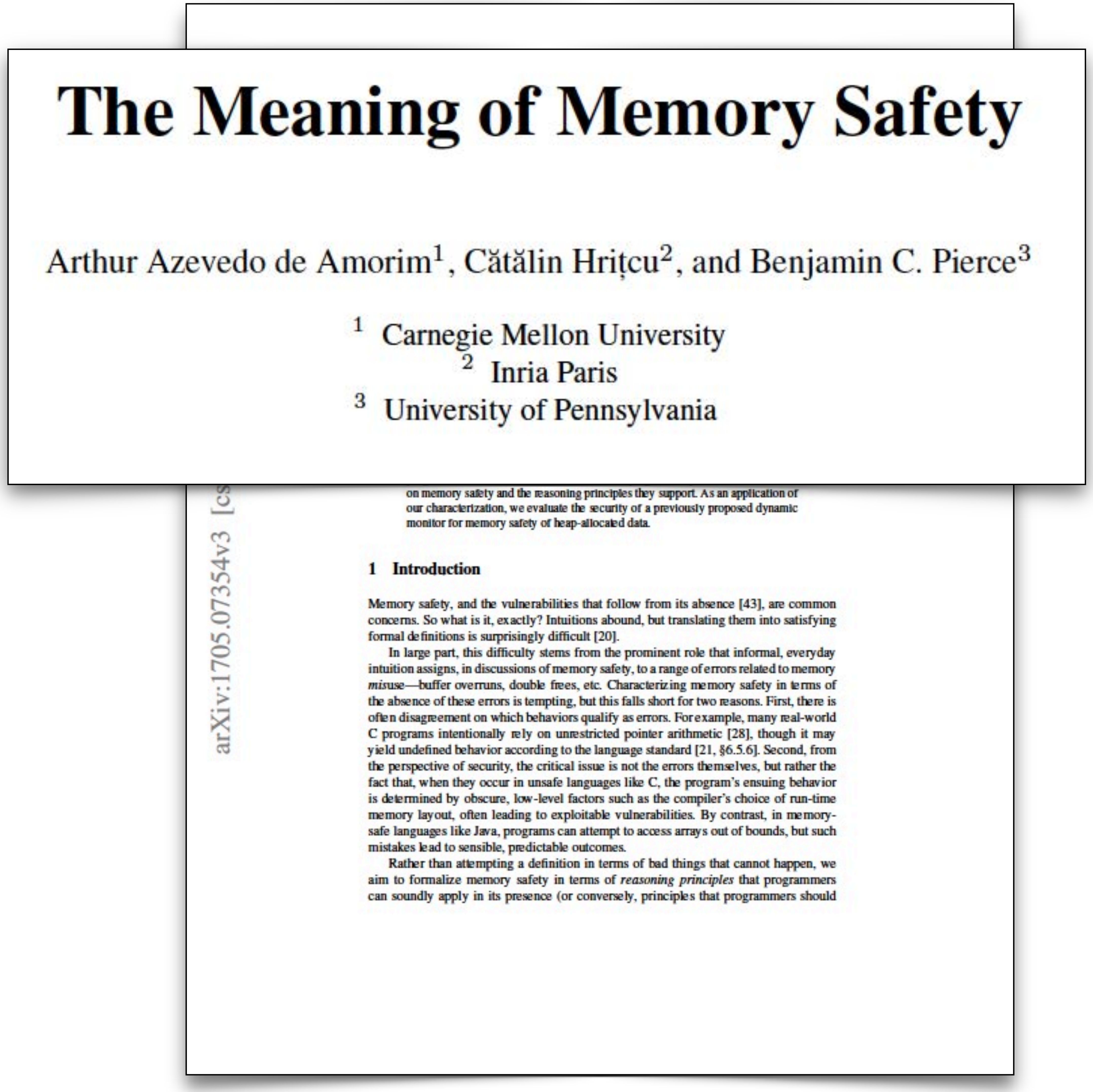
Rather than attempting a definition in terms of bad things that cannot happen, we aim to formalize memory safety in terms of *reasoning principles* that programmers can soundly apply in its presence (or conversely, principles that programmers should

What is a Memory Safe Language?

frame rule: A verified program can only affect a well-defined portion of the state, leaving all other memory regions untouched

- Inspired by separation logic and local reasoning
- The 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 Hrițcu[2], and Benjamin C. Pierce[3]

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

arXiv:1705.07354v3 [cs

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.

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

Frame Rule for Programming Languages

"A verified program can only affect a well-defined portion of the state, leaving all other memory regions untouched."

Frame Rule for Programming Languages

An operation can only access and affect objects defined by the operation's interface, with all other objects unaffected.

- The Law of Exclusivity follows

Memory Safety in C++

Shared mutable references make memory safety possible only with great discipline

Memory Safety in C++

Shared mutable references make memory safety possible only with great discipline

- Sanitizers can help

Memory Safety in C++

Shared mutable references make memory safety possible only with great discipline

- Sanitizers can help
  - But at a performance cost, which often rules them out for production code

Memory Safety in C++

Shared mutable references make memory safety possible only with great discipline

- 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

Memory Safety in C++

Shared mutable references make memory safety possible only with great discipline

- 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

Memory Safety in C++

Shared mutable references make memory safety possible only with great discipline

- 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 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

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

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 defined at a specific level of abstraction; it depends on how we define "operation"

Safety is a tool for local reasoning

Can C++ be Memory Safe?

Can C++ be Memory Safe?

Not without significant changes

Can C++ be Memory Safe?

Not without significant changes

- It will require an explicit safe subset

Can C++ be Memory Safe?

Not without significant changes

- It will require an explicit safe subset
- There is a necessary shift in the abstraction level

Can C++ be Memory Safe?

Not without significant changes

- It will require an explicit safe subset
- There is a necessary shift in the abstraction level

Forward

A (New) Possible Future

A (New) Possible Future

An extensive library of proven generic components

A (New) Possible Future

An extensive 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?"

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:

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]
```

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

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

Closing

Memory safety is important

Closing

Memory safety is important

Use safety properties so your code doesn't do bad things

Closing

Memory safety is important

Use safety properties so your code doesn't do bad things

Use the frame rule to promote security through local reasoning

Closing

Memory safety is important

Use safety properties so your code doesn't do bad things

Use the frame rule to promote security through local reasoning

Don't lose sight that correctness is the goal

Closing

Memory safety is important

Use safety properties so your code doesn't do bad things

Use the frame rule to promote security through local reasoning

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
Artwork by MUE Studio