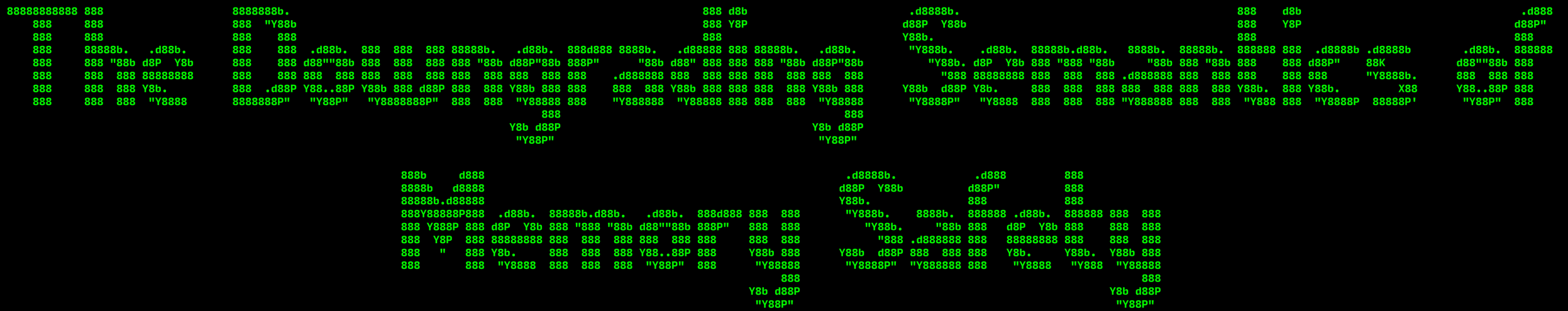


Aslan Askarov, Aarhus University

aslan@cs.au.dk

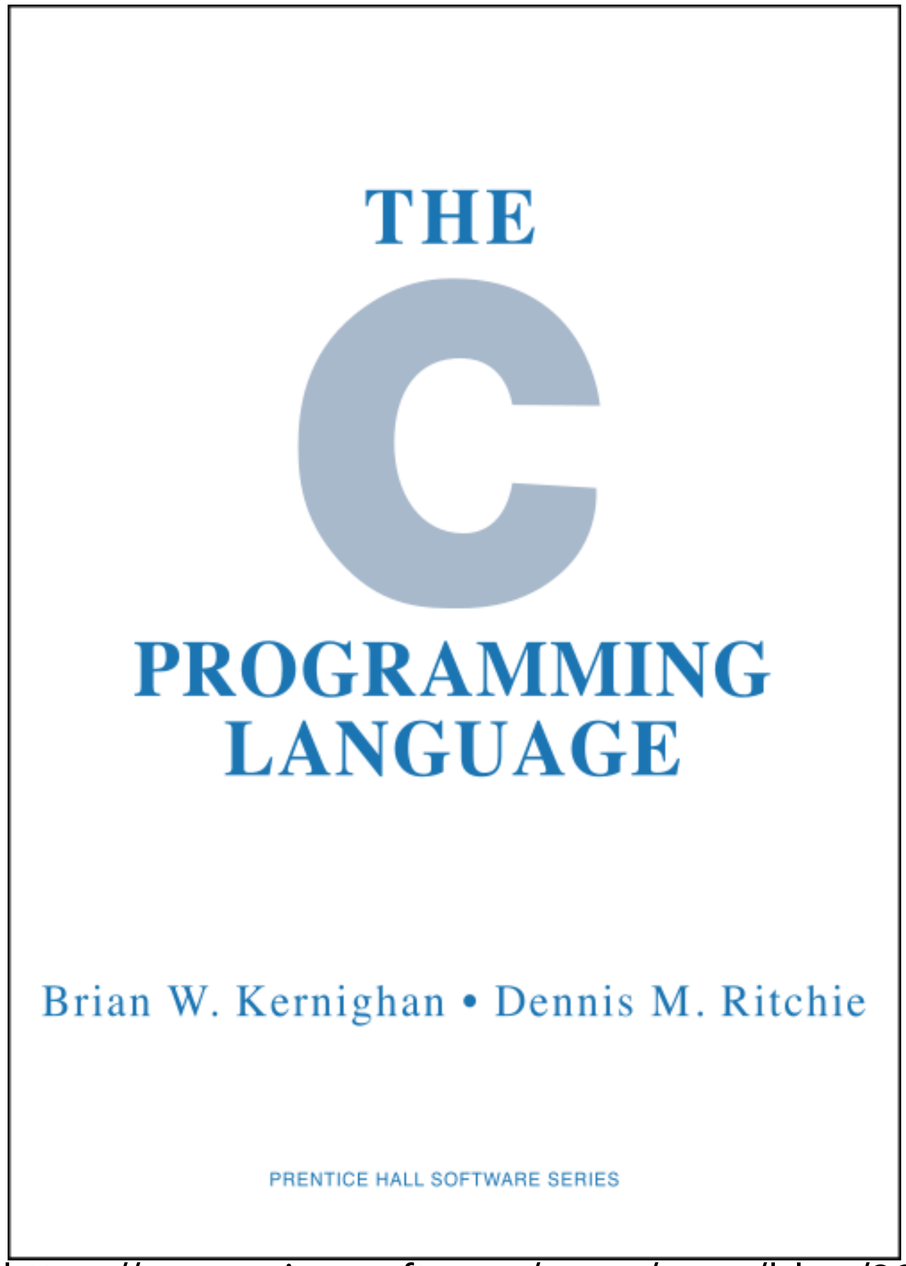
Joint work with **Rene Rydhof Hansen** (AAU) and **Andreas Stenbæk Larsen** (AU)

AU PLS summer school, 2026-08-13

THE C PROGRAMMING LANGUAGE

Brian W. Kernighan • Dennis M. Ritchie

PRENTICE HALL SOFTWARE SERIES



We know that C is not a memory-safe language

- Direct memory access with pointers
- Lack of bounds checking
- Manual memory management
- Uninitialized memory
- Undefined behavior

Can we write memory-safe C programs?

- Of course, if we are disciplined programmers!

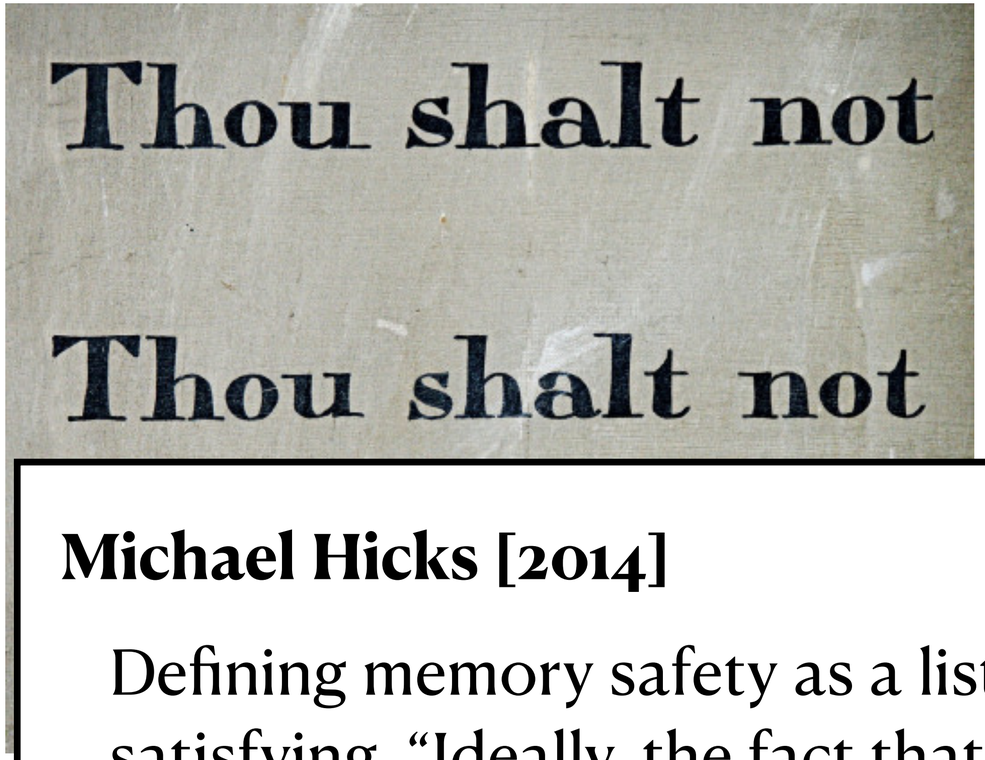
Should we care about memory safety?

- Yes. Microsoft/Google report that 70% of security vulnerabilities are due to memory safety

But what is memory safety, anyway?

https://www.microsoft.com/en-us/msrc/blog/2019/07/we-need-a-safer-systems-programming-language
https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html

The standard definitions of memory safety have the form of a **list of bad things that are not supposed to happen**



– Buffer overflows
– Use after free
– Double/illegal free
– Use of uninitialized memory
– Null pointer dereference

Michael Hicks [2014]

http://www.pl-enthusiast.net/2014/07/21/memory-safety/

> Defining memory safety as a list of what is *not-supposed-to-happen* is not very satisfying. “Ideally, the fact that these errors are ruled out by memory safety is a consequence of its definition, rather than the substance of it. What is the idea that unifies these errors?”

“A good definition matters for scientific progress”

Today's focus

Is **not** about memory-safe languages

Is **not** about tools or analysis for finding bugs in C

It is about a semantic definition that unifies many classes of memory errors

- concretely: heap-related memory errors
- in a flat memory model; pointers are just integers

It is also about discovering a connection between two research areas:

- downgrading in information flow control
- memory safety

Today's focus

Is **not** about memory-safe languages

Is **not** about tools or analysis for finding bugs in C

It is about a semantic definition that unifies many classes of memory errors

The rest of the talk has 4 parts

1. The high level intuition
2. The technical development
3. Examples
4. Discussion

POST'18

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

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.

POST'18

The Meaning of Memory Safety

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

Memory safe languages must support a few reasoning principles, one of which is *noninterference*: programs cannot affect or be affected by unreachable memory.

Noninterference – classical security concept from the 1980s; captures semantic independence

– typically for confidentiality
– but also for integrity

Our approach, for unsafe languages:
– We use the noninterference intuition
– We **cannot** use the memory model of POST'18
  – there pointers are symbolic pairs (b,o)
  – allocated memory is zero'ed
  – memory is infinite

New definition: Gradual Allocator Independence

for allocator-specific aspects of memory safety

Intuition: program **should not depend on** how the allocator is implemented

```
int *p = malloc (128);
printf ("%d\n", p[0]); // bad-1: reads uninitialized memory
                       // bad-2: segfault if malloc fails
```

Insight #1: think of *allocators as secrets*

Noninterference

(as we teach it)

Confidentiality: the program behavior must not depend on secrets

The secrets are just initial values of program variables

```
output (secret)
```

[bad]

```
if (secret) {
   output(1)
} else {
   output(0)
}
```

[bad]

```
if (secret) {
   output(1)
} else {
   output(1)
}
```

[ok]

Strategies

In complex interactive models, we use noninterference on strategies*

Strategy of a user is a **function from prior the user-visible I/O to the next input**

```
input (Alice, secret) (* Invokes Alice's strategy *)
output (Bob, secret)
```

Confidentiality noninterference : strategies should remain confidential

program's public behavior should not depend on the user's secret strategy

* [K. R. O'Neill, M. R. Clarkson, and S. Chong. "Information-flow security for interactive programs", CSFW'06]

Information Flow Control

```
input (Alice, secret)
output (Bob, secret)
```

[bad: Bob learns Alice's secret]

Allocators as secrets

(simplified syntax)

```
p = malloc (8)
print (p)
```

[bad: program behavior depends on the allocator implementation]

Information Flow Control

```
input (Alice, secret1)
input (Alice, secret2)
if (secret1 < secret2){
   output(1)
} else {
   output(0)
}
```

[bad: Bob learns Alice's secret]

Allocators as secrets

```
p = malloc (8)
q = malloc (8)
if (p < q) {
   print (1)
} else {
   print (2)
}
```

[bad: program behavior depends on the allocator implementation]

Information Flow Control

```
input (Alice, secret1)
input (Alice, secret2)
if (secret1 < secret2){
   output(1)
} else {
  output(1)
}
```

[ok]

Allocators as secrets

```
p = malloc (8)
q = malloc (8)
if (p < q) {
   print (1)
} else {
   print (1)
}
```

[ok]

According to the C standard, this pointer comparison is *undefined behavior*

This program satisfies our definition

Semantically, this is the same as `print(1)`

Our goals are not to decree what C standard should say. Let's focus on finding a good semantic definition

```
p = malloc (8)
q = malloc (8)
if (p < q) {
   print (1)
} else {
   print (1)
}
```

[ok]

Some allocator-dependencies are OK

Program **should not depend on** how the allocator is implemented, **except**

- Allocators running out of memory
- Casts of allocated pointers

```
p = malloc (LARGE)
if (p == NULL) {
    print ("Out-of-mem”)
} else {
    print (“OK”)
}
```

```
p = malloc (LARGE)
int i = cast (p)
print (i)
```

Downgrading

Controlled weakening of noninterference
– declassification (C), endorsement (I)

Allowed allocator-dependence

```
p = malloc (LARGE)
if (p == NULL) {
   print ("Out-of-mem”)
} else {
   print (“OK”)
}
```

```
p = malloc (LARGE)
int i = cast (p)
print (i)
```

Downgrading

Controlled weakening of noninterference
– declassification (C), endorsement (I)

```
input (Alice, secret)
int x = declassify (secret)
output (Bob, secret)
```

Allowed allocator-dependence

```
p = malloc (LARGE)
int i = cast (p)
print (i)
```

Operationally, a NO-OP

In IFC literature, downgrading is a no-op, but it is accepted to be important for security definitions.

Thinking of casts as “downgrades” unveils a **clean semantic device** for understanding casts

See [A Two-Phase Infinite/Finite Low-Level Memory Model, by Beck et al., ICFP’24] for the state of the art symbolic memory model where casts are not NO-OPs

Downgrading

Controlled weakening of noninterference
– declassification (C), endorsement (I)

Weakening should not be wrecking

Dimensions and principles of declassification
[Sabelfeld & Sands' 2006]

We adapt our 2020 downgrading definition from the work on Troupe information-flow language

[Bay, Askarov, CSF'2020]

Allowed allocator-dependence

```
p = malloc (LARGE)
if (p == NULL) {
   print ("Out-of-mem")
} else {
   print ("OK")
}
```

```
p = malloc (LARGE)
int i = cast (p)
print (i)
```

Downgrading

Controlled weakening of noninterference
– declassification (C), endorsement (I)

Weakening should not be wrecking

Dimensions and principles of declassification
[Sabelfeld & Sands' 2006]

We adapt our 2020 downgrading definition from the work on Troupe information-flow language

[Bay, Askarov, CSF'2020]

Allowed allocator-dependence

```
p = malloc (LARGE)
if (p == NULL) {
   print ("Out-of-mem")
} else {
   print ("OK")
}
```

```
p = malloc (LARGE)
int i = cast (p)
print (i)
```

Gradual Allocator Independence (informally)

Program **should not depend on** how the allocator is implemented, **except**

- Allocators running out of memory
- Casts of allocated pointers

```
p = malloc (LARGE)
if (p == NULL) {
   print ("Out-of-mem”)
} else {
   print (“OK”)
}
```

```
p = malloc (LARGE)
int i = cast (p)
print (i)
```

The technical development

Notac – a memory unsafe language

$$e ::= n \mid x \mid e \; op \; e \mid *e \mid \&x \mid \text{NULL}$$

$$lval ::= x \mid *e$$

$$\begin{aligned} c ::= \; & lval = e \\ \mid \; & lval = \text{malloc}(e) \\ \mid \; & lval = \text{free}\ (e) \\ \mid \; & \text{skip} \mid c; c \mid \text{if}\ (e)\ c\ \text{else}\ c \\ \mid \; & \text{while}\ (e)\ c \\ \mid \; & lval = \text{cast}\ (e) \\ \mid \; & \text{observe}(e) \end{aligned}$$

No-op

Externally observable

Defined by allocation strategy α

1. Abstract domain of allocator states State
2. NULL: Mem
3. INIT: $\text{Mem} \to \text{Mem} \times \text{State}$
4. MALLOC: $\text{State} \times \text{Mem} \times \mathbf{N} \to \text{State} \times \text{Mem} \times \text{Addr}$
5. FREE: $\text{State} \times \text{Mem} \times \text{Addr} \to \text{State} \times \text{Mem}$

Step relation $\alpha \vdash \langle c; H, A \rangle \to_\eta \langle c'; H', A' \rangle$

H – heap, A – allocator state

Events $\eta ::= \text{obs}(v) \mid \text{malloc}(v, a) \mid \text{mfail}(v) \mid \text{free}(a) \mid \text{cast}(a)$

Transition rules examples

(see the paper for details)

Exp-Null

$$\frac{}{E, H, \alpha \vdash \texttt{NULL} \Downarrow \alpha.\mathbf{null}}$$

C-Cast

$$\frac{E, H, \alpha \vdash lval \downarrow a \qquad E, H, \alpha \vdash e \Downarrow v \qquad a \in \mathrm{dom}(H)}{E, \alpha \vdash \langle lval = \texttt{cast}(e); H, A\rangle \rightarrow_{\mathrm{cast}(v)} \langle H[a \mapsto v], A\rangle}$$

C-Malloc

$$\frac{E, H, \alpha \vdash e \Downarrow n \qquad n \in \mathrm{Size} \qquad (H', A', a) = \alpha.\mathbf{malloc}(H, A, n) \qquad E, H, \alpha \vdash lval \downarrow a_{lval} \qquad a_{lval} \in \mathrm{dom}(H') \qquad \eta = \begin{cases} \mathrm{malloc}(n, a) & a \neq \alpha.\mathbf{null} \\ \mathrm{mfail}(n) & a = \alpha.\mathbf{null} \end{cases}}{E, \alpha \vdash \langle lval = \texttt{malloc}(e); H, A\rangle \rightarrow_{\eta} \langle H'[a_{lval} \mapsto a], A'\rangle}$$

Epistemic approach

The key ingredient is the concept of allocator impact and its variations

Def: **Allocator impact**

Given $\alpha \vdash \langle c, H, A \rangle \rightarrow_t^* \ldots$ then

$$\mathbf{A}(c, t) \triangleq \{\alpha' \mid \alpha' \vdash \langle c, H, A \rangle \rightarrow_{t'}^* \ldots \wedge t \sim t'\}$$

is the set of allocators α' that produce **similar events** as in trace t

Based on [Askarov, Sabelfeld, SP 2007]

Def: **Allocator impact**

Given $\alpha \vdash \langle c, H, A \rangle \rightarrow_t^* \ldots$ then

$$\mathbf{A}(c,t) \triangleq \{\alpha' \mid \alpha' \vdash \langle c, H, A \rangle \rightarrow_{t'}^* \ldots \wedge\, t \sim t'\}$$

is the set of allocators α' that produce similar events as in trace t

Def: **Progess impact for malloc**

Given $\alpha \vdash \langle c, H, A \rangle \rightarrow_t^* \ldots$

$$\mathbf{A}_{\rightarrow}^{+n}(c,t) \triangleq \{\alpha' \mid \alpha' \vdash \langle c, H, A \rangle \rightarrow_{t'}^* \langle \ldots \rangle \rightarrow_\eta \wedge\, t \sim t' \wedge (\eta = \mathsf{malloc}(n,a) \vee \eta = \mathsf{mfail}(n))\}$$

is the set of allocators α' that produce similar events as in trace t

followed by an attempt to allocate n bytes

Def: **Allocator impact**

Given $\alpha \vdash \langle c, H, A \rangle \rightarrow_t^* \ldots$ then

$$\mathbf{A}(c, t) \triangleq \{\alpha' \mid \alpha' \vdash \langle c, H, A \rangle \rightarrow_{t'}^* \ldots \wedge t \sim t'\}$$

is the set of allocators α' that produce similar events as in trace t

Def: **Progess impact for cast**

Given $\alpha \vdash \langle c, H, A \rangle \rightarrow_t^* \ldots$

$$\mathbf{A}_{\rightarrow}^{\mathsf{cast}}(c, t) \triangleq \{\alpha' \mid \alpha' \vdash \langle c, H, A \rangle \rightarrow_{t'}^* \langle \ldots \rangle \rightarrow_\eta \wedge\, t \sim t' \wedge \eta = \mathsf{cast}(a)\}$$

is the set of allocators α' that produce similar events as in trace t **followed by** a cast

Def: **Gradual Allocator Independence**

Suppose $\alpha \vdash \langle c, H, A \rangle \rightarrow_t^* \langle \ldots \rangle \rightarrow_\eta^*$ then

if $\eta = \mathsf{malloc}(n, a) \vee \eta = \mathsf{mfail}(n)$ then $\mathbf{A}_{\rightarrow}^{+n}(c, t) \supseteq \mathbf{A}(c, t)$

if $\eta = \mathsf{cast}(v)$ then $\mathbf{A}_{\rightarrow}^{\mathsf{cast}}(c, t) \supseteq \mathbf{A}(c, t)$

otherwise: $\mathbf{A}(c, t \cdot \eta) \supseteq \mathbf{A}(c, t)$

Program **should not depend on** how the allocator is implemented, **except**

– Allocators running out of memory

– Casts of allocated pointers

Examples

Published March 17, 2026 | Version v1

Software Open

Notac interpreter and examples

Hansen, René Rydhof[1]; Stenbaek Larsen, Andreas[2]; Askarov, Aslan (Contact person)[2]

Show affiliations

This is in the artifact for the paper "The downgrading semantics of memory safety". See included README.md

https://zenodo.org/records/19076610

Files

README.md

Notac Interpreter - PLDI 2026 Artifact

This is the artifact for the Notac interpreter, accompanying the PLDI 2026 paper. The interpreter executes Notac programs under different memory allocator strategies, demonstrating how allocator choice affects program behaviour.

Getting Started (< 5 minutes)

Prerequisites: Docker (Linux, macOS, or Windows).

Step 1. Load the Docker image:

```
docker load -i notac-interpreter.tar.gz
```

Step 2. Run the demo:

```
docker run --rm notac-interpreter sh demo.sh
```

Expected output:

```
------------------------[ NOTAC INTERPRETER DEMO ]------------------------

--------[ Executing null-deref (allocator: null; null-addr: 1) ]--------
Segmentation fault

--------[ Executing null-deref (allocator: bump) ]--------
```

Demo

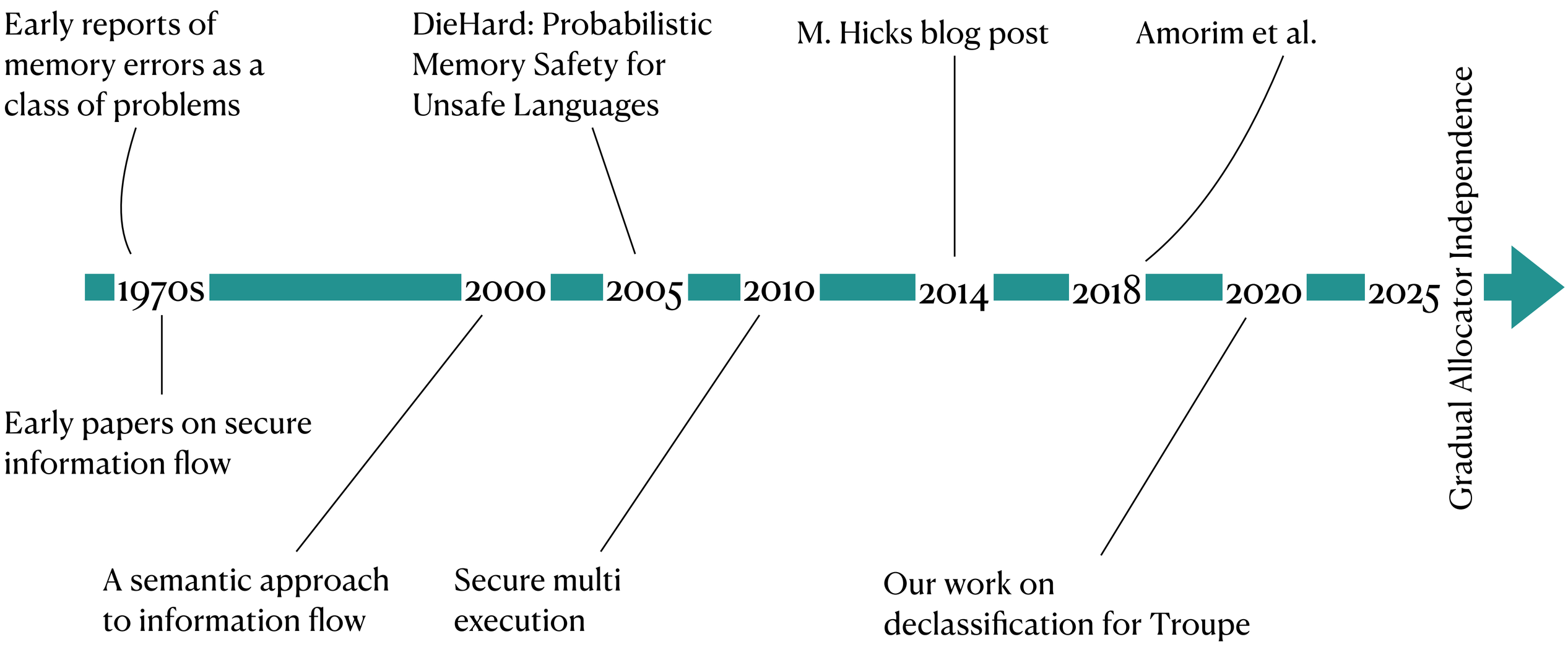
https://askarov.net/notac/

Discussion

Translation from a memory safe language to Notac

- We define a Memsafe (the language of [POST'18]) to Notac translation [[c]]
  - Tracks OOM-status of each malloc in a special variable
  - Initializes all allocations to zero
- Transparency Theorem
  - The translated programs satisfy GAI
  - If the Memsafe program c is not stuck, the translated Notac program [[c]] is either OOM or succeeds with the same memory

Perspective



PLDI'06

DieHard: Probabilistic Memory Safety for Unsafe Languages

Emery D. Berger
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
emery@cs.umass.edu

Benjamin G. Zorn
Microsoft Research
One Microsoft Way
Redmond, WA 98052
zorn@microsoft.com

Abstract

Applications written in unsafe languages like C and C++ are vulnerable to memory errors such as buffer overflows, dangling pointers, and reads of uninitialized data. Such errors can lead to program crashes, security vulnerabilities, and unpredictable behavior. We present DieHard, a runtime system that tolerates these errors while probabilistically maintaining soundness. DieHard uses randomization and replication to achieve *probabilistic memory safety* by approximating an infinite-sized heap. DieHard's memory manager randomizes the location of objects in a heap that is at least twice as large as required. This algorithm prevents heap corruption and provides a probabilistic guarantee of avoiding memory errors. For additional safety, DieHard can operate in a replicated mode where multiple replicas of the same application are run simultaneously. By initializing each replica with a different random seed and requiring agreement on output, the replicated version of DieHard increases the likelihood of correct execution because errors are unlikely to have the same effect across all replicas. We present analytical and experimental results that show DieHard's resilience to a wide range of memory errors, including a heap-based buffer overflow in an actual application.

Dangling pointers: If the program mistakenly frees a live object, the allocator may overwrite its contents with a new object or heap metadata.

Buffer overflows: Out-of-bound writes can corrupt the contents of live objects on the heap.

Heap metadata overwrites: If heap metadata is stored near heap objects, an out-of-bound write can corrupt it.

Uninitialized reads: Reading values from newly-allocated or unallocated memory leads to undefined behavior.

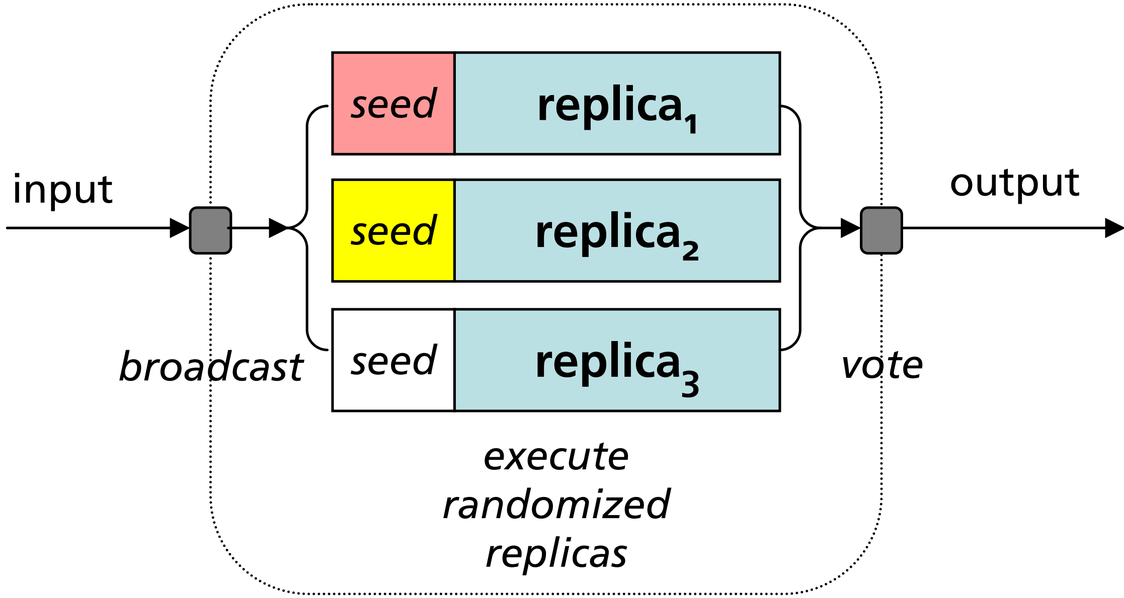
Invalid frees: Passing illegal addresses to `free` can corrupt the heap or lead to undefined behavior.

Double frees: Repeated calls to `free` of objects that have already been freed cause freelist-based allocators to fail.

Tools like Purify [18] and Valgrind [28, 35] allow programmers to pinpoint the exact location of these memory errors (at the cost of a 2-25X performance penalty), but only reveal those bugs found during testing. Deployed programs thus remain vulnerable to crashes or attack. Conservative garbage collectors can, at the cost of increased runtime and additional memory [12, 20], disable calls to

DieHard's replicated allocator [PLDI'06]

While replacing an application's allocator with DieHard reduces the likelihood of memory errors, this stand-alone approach cannot detect uninitialized reads. To catch these errors, and to further increase the likelihood of correct execution, we have built a version of DieHard (currently for UNIX platforms only) that executes several replicas simultaneously. Figure 3 depicts the architecture, instantiated with three replicas.



cf. the idea of self-composition in IFC and later secure-multi-execution

Future work

- Information disclosure vulnerabilities
  - Adversarial allocators that retain copy of information on memset_s
- Sub-object access for supporting structs and stack safety
- Reasoning about optimizations via GAI
  - (as opposed to via defined/undefined behavior as it is done today)
  - Can we show soundness of intra-procedural optimizations, assuming external functions satisfy GAI?
  - Can we connect Memory safety <-> Downgrading <-> Quantitative IFC to ask questions such as “How much memory safety is broken by optimization X”?
- Deduplication safety based on the allocator model (not in the talk)
- What does it all mean for existing languages: C, Rust, etc
- Enforcing GAI

JOEL ON SOFTWARE

YOUR HOST

NOVEMBER 11, 2002 *by* JOEL SPOLSKY

The Law of Leaky Abstractions

“All non-trivial abstractions, to some degree, are leaky”

TCP is a way to transmit data that is *reliable*. By this I mean: if you

Is there a general theory of the downgrading semantics of leaky abstractions?

The Downgrading Semantics of Memory Safety

RENÉ RYDHOF HANSEN, Aalborg University, Denmark
ANDREAS STENBÆK LARSEN, Aarhus University, Denmark
ASLAN ASKAROV, Aarhus University, Denmark

Memory safety is traditionally characterized in terms of bad things that cannot happen. This approach is currently embraced in the literature on formal methods for memory safety. However, a general semantic principle for memory safety, that implies the negative items, remains elusive.

This paper focuses on the allocator-specific aspects of memory safety, such as null-pointer dereference, use after free, double free, and heap overflow. To that extent, we propose a notion of *gradual allocator independence* that accurately captures the allocator-dependent aspects of memory safety. Our approach is inspired by the previously suggested connection between memory safety and noninterference, but extends that connection in a fundamentally important direction towards downgrading.

We consider a low-level language with access to an allocator that provides malloc and free primitives in a flat memory model. Pointers are just integers, and as such it is trivial to write memory-unsafe programs. The basic intuition of gradual allocator independence is that of noninterference, namely that allocators must not influence program execution. This intuition is refined in two important ways that account for the allocators running out-of-memory and for programs to have pointer-to-integer casts. The key insight of the definition is to treat these extensions as forms of downgrading and give them satisfactory technical treatment using the state-of-the-art information flow machinery.

CCS Concepts: • **Security and privacy** → **Information flow control**; • **Theory of computation** → **Operational semantics**; • **Software and its engineering** → *Allocation / deallocation strategies.*

Additional Key Words and Phrases: memory safety, allocators, declassification, cast

ACM Reference Format:
René Rydhof Hansen, Andreas Stenbæk Larsen, and Aslan Askarov. 2026. The Downgrading Semantics of Memory Safety. *Proc. ACM Program. Lang.* 10, PLDI, Article 182 (June 2026), 28 pages. https://doi.org/10.1145/3808260

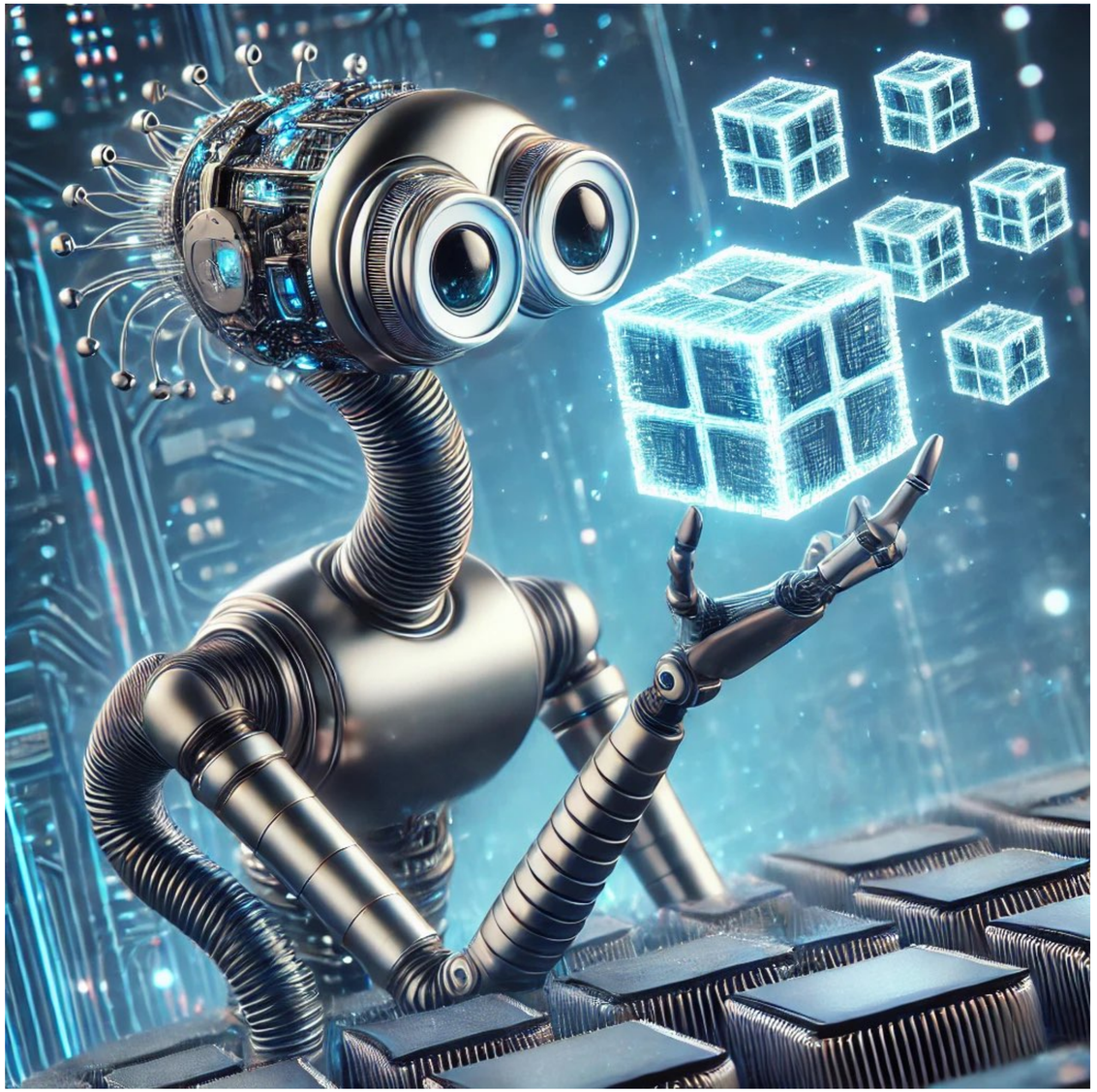
1 Introduction

Memory safety is traditionally characterized in terms of bad things that cannot happen [37, 12]. This characterization is used in the literature on memory safety, typically for reasoning about optimizations in C compilers [45, 17, 42], or basic security properties such as access control [69].

The approach to defining memory safety in terms of negative behavior has recognized limitations [52, 53]. For example, when programs exhibit complex behavior, such as casting pointers to integers and back, the associated reasoning principles become semantically fragile [17]. It is unclear how to build upon approaches based on negative lists for applications where semantic consistency is necessary, such as end-to-end security. These limitations align with the 2014 observation by Hicks [37] that a satisfactory semantic definition of memory safety – the one that would imply the traditionally accepted negative behavior – remains elusive.

Authors' Contact Information: René Rydhof Hansen, rrh@cs.aau.dk, Aalborg University, Aalborg, Denmark; Andreas Stenbæk Larsen, astenbaek@cs.au.dk, Aarhus University, Aarhus, Denmark; Aslan Askarov, aslan@cs.au.dk, Aarhus University, Aarhus, Denmark.

ACM 2475-1421/2026/6-ART182
https://doi.org/10.1145/3808260



https://dl.acm.org/doi/10.1145/3808260
https://arxiv.org/abs/2507.11282