A Dependent Nominal Physical Type System for Static Analysis of Memory in Low Level Code

---

Julien Simonnet Matthieu Lemerre Mihaela Sighireanu

OOPSLA 2024

Context

Google Security Engineering Technical Report
March 4, 2024

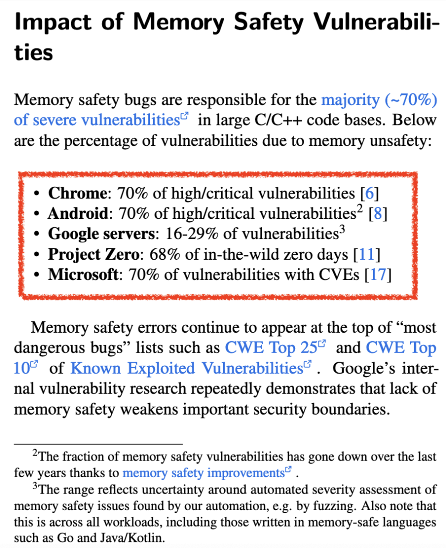
Secure by Design:
Google's Perspective on Memory Safety

Alex Rebert
`arebert@google.com`

Christoph Kern
`xtof@google.com`

Source: `https://research.google/pubs/secure-by-design-googles-perspective-on-memory-safety/`

Context



Impact of Memory Safety Vulnerabilities

Memory safety bugs are responsible for the majority (~70%) of severe vulnerabilities in large C/C++ code bases. Below are the percentage of vulnerabilities due to memory unsafety:

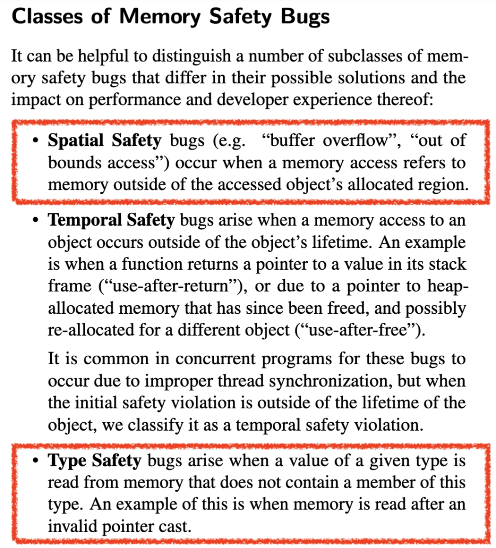
- **Chrome**: 70% of high/critical vulnerabilities [6]
- **Android**: 70% of high/critical vulnerabilities[2] [8]
- **Google servers**: 16-29% of vulnerabilities[3]
- **Project Zero**: 68% of in-the-wild zero days [11]
- **Microsoft**: 70% of vulnerabilities with CVEs [17]

Memory safety errors continue to appear at the top of "most dangerous bugs" lists such as CWE Top 25 and CWE Top 10 of Known Exploited Vulnerabilities. Google's internal vulnerability research repeatedly demonstrates that lack of memory safety weakens important security boundaries.

---

[2] The fraction of memory safety vulnerabilities has gone down over the last few years thanks to memory safety improvements.

[3] The range reflects uncertainty around automated severity assessment of memory safety issues found by our automation, e.g. by fuzzing. Also note that this is across all workloads, including those written in memory-safe languages such as Go and Java/Kotlin.



Classes of Memory Safety Bugs

It can be helpful to distinguish a number of subclasses of memory safety bugs that differ in their possible solutions and the impact on performance and developer experience thereof:

- **Spatial Safety** bugs (e.g. "buffer overflow", "out of bounds access") occur when a memory access refers to memory outside of the accessed object's allocated region.
- **Temporal Safety** bugs arise when a memory access to an object occurs outside of the object's lifetime. An example is when a function returns a pointer to a value in its stack frame ("use-after-return"), or due to a pointer to heap-allocated memory that has since been freed, and possibly re-allocated for a different object ("use-after-free").

  It is common in concurrent programs for these bugs to occur due to improper thread synchronization, but when the initial safety violation is outside of the lifetime of the object, we classify it as a temporal safety violation.
- **Type Safety** bugs arise when a value of a given type is read from memory that does not contain a member of this type. An example of this is when memory is read after an invalid pointer cast.

Source: `https://research.google/pubs/secure-by-design-googles-perspective-on-memory-safety/`

Objectives

Our goal is to develop a **sound static analysis** which:

- is **automatic**, practical and cost-effective
- targets **low-level programs**, e.g., C or binary code
- needs **no changes** to the source code or the compiler
- can prove **type safety** implying **spatial memory safety**

Contributions

We propose `TypedC`, a new **dependent type system**

- providing expressive invariants over memory
- on top of the C type system
- able to express low-level code patterns
  - e.g., bit-stealing, interior pointers, flexible array member ...

and an **automatic type-checking by abstract interpretation** featuring

- cheap analysis operations
- modular (per-function) analysis
- easy configuration of the analysis by the annotated header files

Example

```
struct buf {  // C header
  int   size;
  char* content;
};
void reset(struct buf* x);
```

```
∃ len:(int with self>0).struct buf { // TypedC spec
  (int with self=len) size;
  char[len]⋆ content;
};
void reset(struct buf* x);
```

Example

```
struct buf {  // C header
  int   size;
  char* content;
};
void reset(struct buf* x);
```

```
∃ len:(int with self>0).struct buf { // TypedC spec
  (int with self=len) size;
  char[len]⋆ content;
};
void reset(struct buf* x);
```

Incorrect program

```
void reset(struct buf* x) {
  int len = x->size; ← alarm 1
  for (i=0; i<=len; i++)
    x->content[i] = 0; ← alarm 2
  x->size++; ← alarm 3
}
```

The analysis outputs:

alarm 1: null pointer dereference

alarm 2: out of bound access

alarm 3: type error

Example

```
struct buf {  // C header
  int   size;
  char* content;
};
void reset(struct buf* x);
```

```
∃ len:(int with self>0).struct buf { // TypedC spec
  (int with self=len) size;
  char[len]⋆ content;
};
void reset(struct buf⋆ x);
```

Incorrect program

```
void reset(struct buf* x) {
  int len = x->size; ← alarm 1
  for (i=0; i<=len; i++)
    x->content[i] = 0; ← alarm 2
  x->size++; ← alarm 3
}
```

The analysis outputs:

alarm 1: null pointer dereference

alarm 2: out of bound access

alarm 3: type error

Correct program

```
void reset(struct buf* x) {
  int len = x->size;
  for (i=0; i<len; i++)
    x->content[i] = 0;

}
```

The analysis guarantees:

- spatial safety
- type safety

Comparison with other methods

Methods	Tools	Automation	Annotation	Expressivity	Runtime impact
Run-time & hybrid	EffectiveSan SoftBound	+++	none	NA	−−
Syntactic type-checking	CheckedC CCured TypedAssembly	+	some	+	−
Semantic type-checking	Codex (TypedC)	++	**light spec**	++	++
Shape analysis	MemCAD TVLA	+	full spec	+++	++

Overview

1. TypedC by example
2. Type-checking by abstract interpretation
3. Evaluation

Record types: $\texttt{struct}\{\tau_1\ f_1; \ldots\ \tau_n\ f_n;\}$ & array types: $\tau[e]$

Record types and **array types** concatenate types to represent memory layouts.

```
// C code


struct message {
  struct message* next;
  char* buffer; };

struct message_box {
  int length;
  struct message* first; };
```

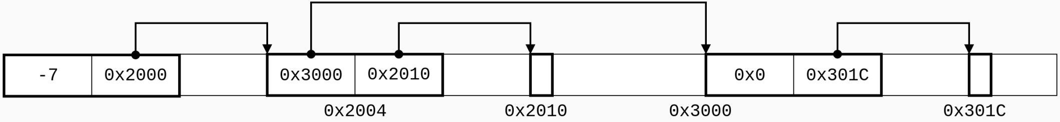
```
// TypedC spec


struct message {
  struct message* next;
  char* buffer; };

struct message_box {
  int length;
  struct message* first; };
```

Record types: $\texttt{struct}\{\tau_1\ f_1; \ldots\ \tau_n\ f_n; \}$ & array types: $\tau[e]$

Record types and **array types** concatenate types to represent memory layouts.

```
// C code


struct message {
  struct message* next;
  char* buffer; };

struct message_box {
  int length;
  struct message* first; };
```

```
// TypedC spec


struct message {
  struct message* next;
  char* buffer; };

struct message_box {
  int length;
  struct message* first; };
```

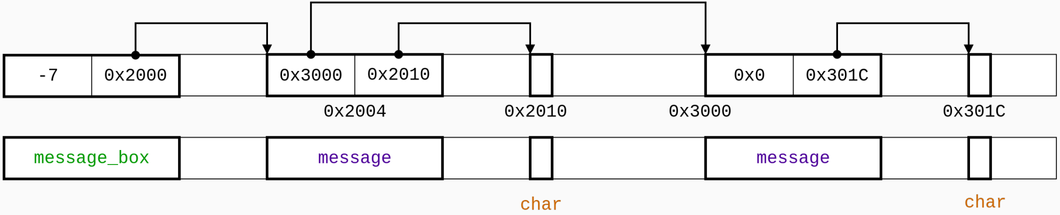


Record types: $\texttt{struct}\{\tau_1\ f_1; \ldots\ \tau_n\ f_n;\}$ & array types: $\tau[e]$

Record types and **array types** concatenate types to represent memory layouts.

```
// C code


struct message {
  struct message* next;
  char* buffer; };

struct message_box {
  int length;
  struct message* first; };
```

```
// TypedC spec


struct message {
  struct message* next;
  char* buffer; };

struct message_box {
  int length;
  struct message* first; };
```

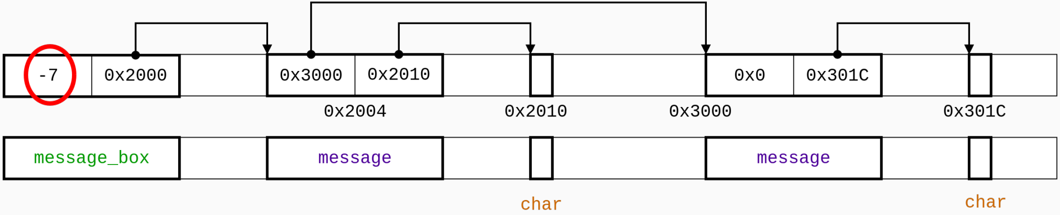


Record types: $\texttt{struct}\{\tau_1\ f_1; \ldots\ \tau_n\ f_n; \}$ & array types: $\tau[e]$

Record types and **array types** concatenate types to represent memory layouts.

```
// C code


struct message {
  struct message* next;
  char* buffer; };

struct message_box {
  int length;
  struct message* first; };
```

```
// TypedC spec


struct message {
  struct message* next;
  char* buffer; };

struct message_box {
  int length;
  struct message* first; };
```

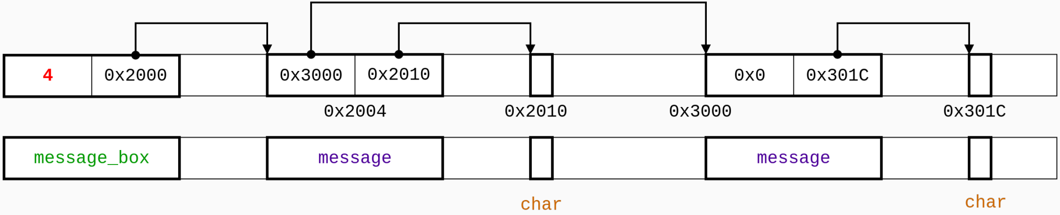


Refinement types: τ `with` p

Values in a **refinement type** fulfil a given predicate.

```c
// C code


struct message {
  struct message* next;
  char* buffer; };

struct message_box {
  int length;
  struct message* first; };
```

```c
// TypedC spec
#define posint (byte[4] with self>0)

struct message {
  struct message* next;
  char* buffer; };

struct message_box {
 posint length;
  struct message* first; };
```

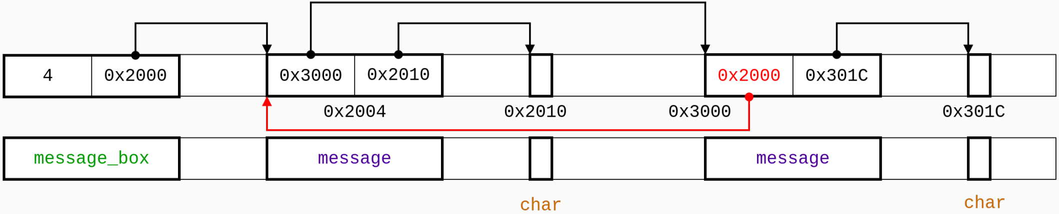


Non null pointer types: $\eta\star$

Pointer types denote **non null** addresses.

```
// C code


struct message {
  struct message* next;
  char* buffer; };

struct message_box {
  int length;
  struct message* first; };
```

```
// TypedC spec
#define posint (byte[4] with self>0)

struct message {
  struct message⋆ next;
  char⋆ buffer; };

struct message_box {
  posint length;
  struct message⋆ first; };
```

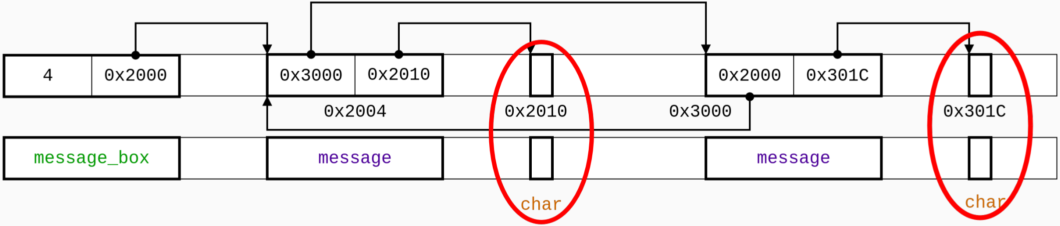


Non null pointer types: $\eta\star$

Pointer types denote **non null** addresses.

```
// C code


struct message {
  struct message* next;
  char* buffer; };

struct message_box {
  int length;
  struct message* first; };
```

```
// TypedC spec
#define posint (byte[4] with self>0)

struct message {
  struct message⋆ next;
  char⋆ buffer; };

struct message_box {
  posint length;
  struct message⋆ first; };
```

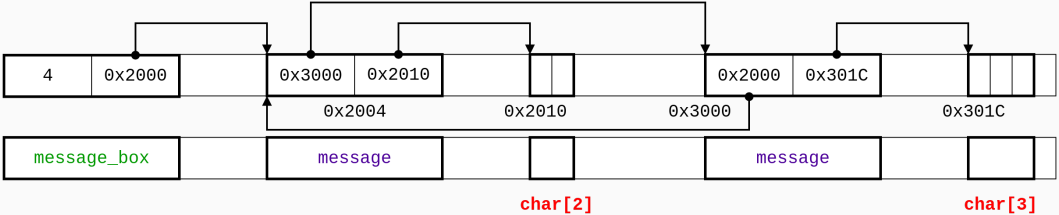


Existential types: $\exists\alpha : \tau_1.\ \tau_2$

Existential types introduce new **symbolic variables**.

```
// C code


struct message {
  struct message* next;
  char* buffer; };

struct message_box {
  int length;
  struct message* first; };
```

```
// TypedC spec
#define posint (byte[4] with self>0)

∃ len:posint. struct message {
  struct message⋆ next;
  char[len]⋆ buffer; };

struct message_box {
  posint length;
  struct message⋆ first; };
```

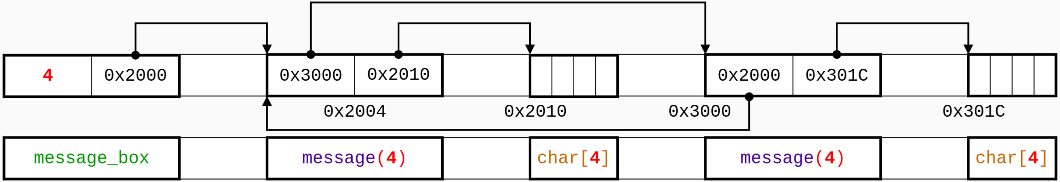


Parameterized types: $n(e_1, \ldots, e_\ell)$

Parameterized types share constraints between memory blocks.

```
// C code


struct message {
  struct message* next;
  char* buffer };

struct message_box {
  int length;
  struct message* first; };
```

```
// TypedC spec
#define posint (byte[4] with self>0)

struct message(len:posint) {
  struct message(len)⋆ next;
  char[len]⋆ buffer; };

∃ l:posint. struct message_box {
  (byte[4] with self=l) length;
  struct message(l)⋆ first; };
```

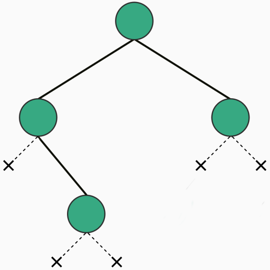


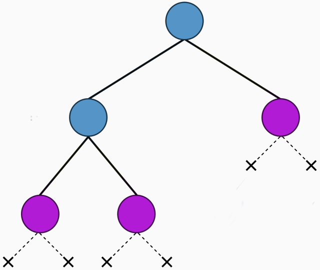
Union types: $\texttt{union}\{\tau_1\ c_1; \ldots\ \tau_n\ c_n;\}$

Union types specify disjunctions of invariants.

```
// Basic TypedC definition

struct node {
  struct node* left;
  struct node* right;
};
```

```
// Refined TypedC spec
#define nullptr (byte[4] with self=0)

struct leaf {nullptr l; nullptr r;};
struct interior {node⋆ l; node⋆ r;};
union node {struct interior inode; struct leaf lnode;};
```

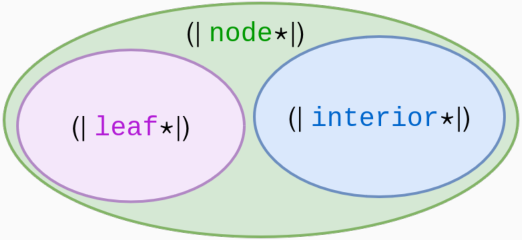


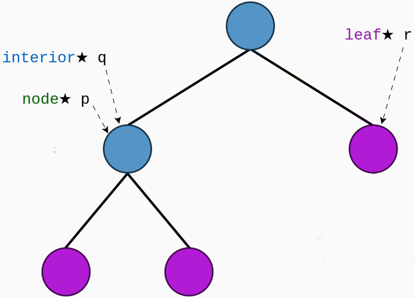


Nominal type system

Pointer types $\eta\star$ represent addresses to region named η.

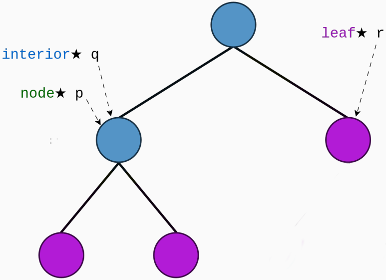
```
#define nullptr (byte[4] with self=0)

struct leaf {nullptr l; nullptr r;};
struct interior {node⋆ l; node⋆ r;};
union node {
  struct interior inode; struct leaf lnode;
};
```





Mild updates: playing with names

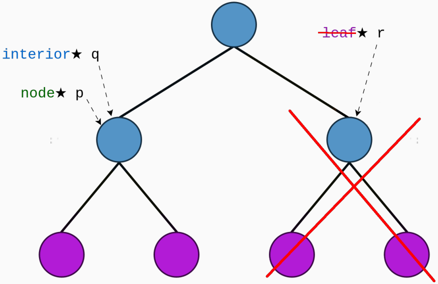
```
#define nullptr (byte[4] with self=0)

struct leaf {nullptr l; nullptr r;};
struct interior {node⋆ l; node⋆ r;};
union node {
  struct interior inode; struct leaf lnode;};
```



pointers to `leaf` may exist

Mild updates: playing with names

```
#define nullptr (byte[4] with self=0)

struct leaf {nullptr l; nullptr r;};
struct interior {node⋆ l; node⋆ r;};
union node {
  struct interior inode; struct leaf lnode;};
```

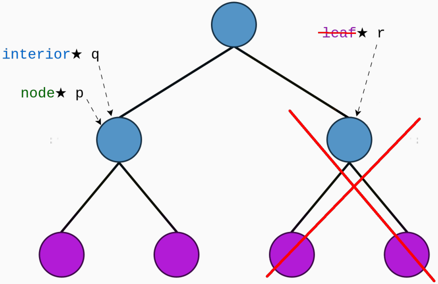


pointers to `leaf` may exist

nodes cannot change from `leaf` to `interior`

Mild updates: playing with names

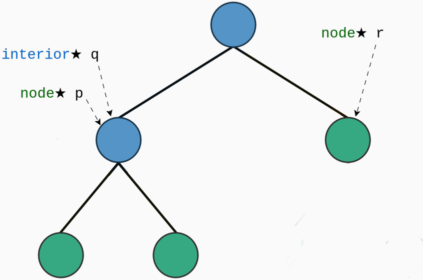
```
#define nullptr (byte[4] with self=0)

struct leaf {nullptr l; nullptr r;};
struct interior {node⋆ l; node⋆ r;};
union node {
  struct interior inode; struct leaf lnode;};
```



pointers to `leaf` may exist

nodes cannot change from `leaf` to `interior`

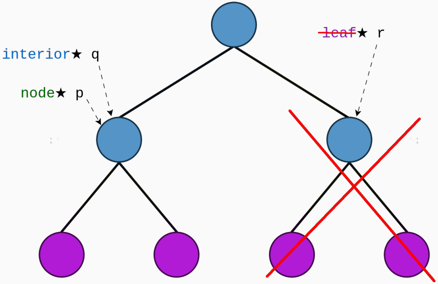
```
#define nullptr (byte[4] with self=0)

#define leaf struct {nullptr l; nullptr r;}
struct interior {node⋆ l; node⋆ r;};
union node {
  struct interior inode; leaf lnode;};
```



pointers to `leaf` do not exist

Mild updates: playing with names

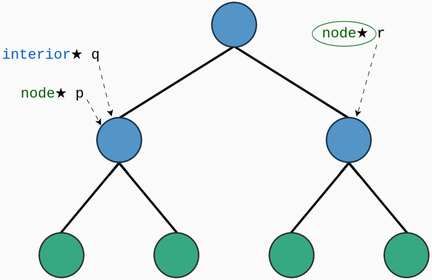
```
#define nullptr (byte[4] with self=0)

struct leaf {nullptr l; nullptr r;};
struct interior {node⋆ l; node⋆ r;};
union node {
  struct interior inode; struct leaf lnode;};
```



pointers to leaf may exist

nodes cannot change from leaf to interior

```
#define nullptr (byte[4] with self=0)

#define leaf struct {nullptr l; nullptr r;}
struct interior {node⋆ l; node⋆ r;};
union node {
  struct interior inode; leaf lnode;};
```



pointers to leaf do not exist

nodes can change from leaf to interior

Type-checking by abstract interpretation

```
// C code
void example(int* p) {  ⟵
  p++ ;
  int x = *p ;
  *p = x - 17;
}
```

```
// TypedC spec
type posint = (byte[4] with self>0)

void example(posint[3]⋆ p);
```

Abstract state

Initially $s^\sharp = (\sigma^\sharp,\ \Gamma^\sharp,\ \nu^\sharp)$ where

$$\sigma^\sharp = \{p \mapsto \alpha\}$$

$$\Gamma^\sharp = \{\alpha \mapsto \texttt{posint}[3]\star + 0\}$$

$$\nu^\sharp = \{\alpha \in [1; 2^{32} - 12)\}$$

Type-checking by abstract interpretation

```
// C code
void example(int* p) {
  p++ ;                  ⟵
  int x = *p ;
  *p = x - 17;
}
```

```
// TypedC spec
type posint = (byte[4] with self>0)

void example(posint[3]⋆ p);
```

Abstract state

Following the instruction p++,

$$\sigma^\sharp = \{p \mapsto \alpha + 4\}$$

$$\Gamma^\sharp = \{\alpha \mapsto \texttt{posint}[3]\star + 0;\ \alpha + 4 \mapsto \texttt{posint}[3]\star + 4\}$$

$$\nu^\sharp = \{\alpha \in [1; 2^{32} - 12);\ \alpha + 4 \in [5; 2^{32} - 8)\}$$

Type-checking by abstract interpretation

```
// C code
void example(int* p) {
  p++ ;
  int x = *p ;                ←
  *p = x - 17;
}
```

```
// TypedC spec
type posint = (byte[4] with self>0)

void example(posint[3]⋆ p);
```

Abstract state

Following the instruction `int x = *p`,

$$
\begin{aligned}
\sigma^\sharp &= \{p \mapsto \alpha + 4;\ x \mapsto \beta\} \\
\Gamma^\sharp &= \{\alpha \mapsto \texttt{posint}[3]\star + 0;\ \alpha + 4 \mapsto \texttt{posint}[3]\star + 4\} \\
\nu^\sharp &= \{\alpha \in [1; 2^{32} - 12);\ \alpha + 4 \in [5; 2^{32} - 8);\ \beta \in [1; 2^{31})\}
\end{aligned}
$$

Type-checking by abstract interpretation

```c
// C code
void example(int* p) {
  p++ ;
  int x = *p ;
  *p = x - 17;              ←—— type error
}
```

```
// TypedC spec
type posint = (byte[4] with self>0)

void example(posint[3]⋆ p);
```

Abstract state

Following the instruction $*\mathrm{p} = \mathrm{x} - 17$,

$$\sigma^\sharp = \{p \mapsto \alpha + 4;\ x \mapsto \beta\}$$

$$\Gamma^\sharp = \{\alpha \mapsto \texttt{posint}[3]\star + 0;\ \alpha + 4 \mapsto \texttt{posint}[3]\star + 4\}$$

$$\nu^\sharp = \{\alpha \in [1; 2^{32} - 12);\ \alpha + 4 \in [5; 2^{32} - 8);\ \beta \in [1; 2^{31});\ (\beta - 17) \in [-16; 2^{31} - 17)\}$$

Type error: $\beta - 17 \in [-16, \dots)$ is not of type `posint`.

Evaluation using the Codex tool

Code patterns

(BS) bit-stealing
(DU) discriminated variant types
(NLI) non-local invariants

(FAM) flexible array member
(IP) interior pointers
(P?) possibly null pointer

	Case studies	#LoC	#Fun	Code patterns						Spec#lines		#Alarms			Time
				BS	DU	NLI	FAM	IP	P?	orig	man	orig	final	true	(s)
OS	Contiki	329	12	–	–	–	–	–	✓	19	14	16	2	0	1.33
	QDS^{bin}	401	3	–	✓	✓	–	–	✓	–	83	18	0	0	1.28
	RBTree Linux	1 111	2	–	–	–	–	✓	✓	29	17	6	2	0	0.46
Emacs	list^{bin}	464	8	✓	✓	–	–	–	✓		73	-	0	0	3.03
	$\text{string}^{\text{bin}}$	109	5	✓	✓	✓	–	–	✓			-	4	0	3.20
	$\text{buffer}^{\text{bin}}$	42	3	✓	✓	–	✓	–	✓			-	0	0	3.12
Shapes	Graph	155	7	–	–	–	–	–	✓	26	14	0	0	0	0.79
	Javl	920	9	–	–	–	–	–	✓	37	34	10	1	1	0.70
	Kennedy	197	6	–	–	–	–	✓	✓	44	24	6	0	0	0.74
	RBtree	978	7	–	–	–	–	–	✓	32	18	56	16	0	0.42
	(6-)Other	5 742	19	–	–	–	–	–	✓	113	50	43	5	0	3.79

Legend: Specification lines originally in C and lines manually modified.

Evaluation using the Codex tool

Code patterns

(BS) bit-stealing
(DU) discriminated variant types
(NLI) non-local invariants

(FAM) flexible array member
(IP) interior pointers
(P?) possibly null pointer

	Case studies	**#LoC**	**#Fun**	**Code patterns**						**Spec#lines**		**#Alarms**			**Time**
				BS	DU	NLI	FAM	IP	P?	orig	man	orig	final	true	(s)
OS	Contiki	329	12	–	–	–	–	–	✓	19	14	16	2	0	1.33
	QDS^{bin}	401	3	–	✓	✓	–	–	✓	–	83	18	0	0	1.28
	RBTree Linux	1 111	2	–	–	–	–	✓	✓	29	17	6	2	0	0.46
Emacs	list^{bin}	464	8	✓	✓	–	–	–	✓			-	0	0	3.03
	string^{bin}	109	5	✓	✓	✓	–	–	✓		73	-	4	0	3.20
	buffer^{bin}	42	3	✓	✓	–	✓	–	✓			-	0	0	3.12
Shapes	Graph	155	7	–	–	–	–	–	✓	26	14	0	0	0	0.79
	Javl	920	9	–	–	–	–	–	✓	37	34	10	1	1	0.70
	Kennedy	197	6	–	–	–	–	✓	✓	44	24	6	0	0	0.74
	RBtree	978	7	–	–	–	–	–	✓	32	18	56	16	0	0.42
	(6-)Other	5 742	19	–	–	–	–	–	✓	113	50	43	5	0	3.79

Legend: Specification lines originally in C and lines manually modified.

Comparison with state-of-the-art tool CheckedC

Bench	**#LoC**	**#Fun**	**Code patterns**						**CC+3C**		**Spec#lines**		**#Alarms**			**Time**
Olden			BS	DU	NLI	FAM	IP	P?	man	gen	man	orig	orig	final	true	(s)
bh^{C}	2 107	30	–	✓	–	–	–	✓	181	48	27	144	39	3	1	26.04
bisort^{C}	356	11	–	✓	–	–	✓	✓	92	34	26	29	9	0	0	2.18
em3d^{C}	693	17	–	–	✓	–	–	✓	158	88	52	53	42	15	0	6.48
health^{C}	485	13	–	–	–	–	–	✓	99	57	39	57	16	4	0	5.96
mst^{C}	431	5	–	✓	–	–	–	✓	161	28	17	44	33	10	3	1.89
perimeter^{C}	486	12	–	✓	–	–	–	✓	44	10	69	41	13	1	0	1.64
power^{C}	618	17	–	–	–	–	–	✓	83	30	26	75	26	5	0	6.04
treeadd^{C}	249	2	–	–	–	–	–	✓	46	16	0	19	0	0	0	0.42
tsp^{C}	617	12	–	–	✓	–	–	✓	78	9	2	32	6	0	0	3.86
voronoi^{C}	1 151	40	✓	–	–	–	–	✓	✗	✗	38	101	57	44	0	21.35

Semantic type-checking with respect to Syntactic type-checking

Pros

+ more expressive invariants
+ works on unmodified programs
+ fully static, no execution overhead

Cons

- needs deeper code understanding
- elaborate analysis
- may report false alarms

Comparison with state-of-the-art tool CheckedC

Bench	#LoC	#Fun	Code patterns						CC+3C		Spec#lines		#Alarms			Time
Olden			BS	DU	NLI	FAM	IP	P?	man	gen	man	orig	orig	final	true	(s)
bh^{C}	2 107	30	–	✓	–	–	–	✓	181	48	27	144	39	3	1	26.04
bisort^{C}	356	11	–	✓	–	–	✓	✓	92	34	26	29	9	0	0	2.18
em3d^{C}	693	17	–	–	✓	–	–	✓	158	88	52	53	42	15	0	6.48
health^{C}	485	13	–	–	–	–	–	✓	99	57	39	57	16	4	0	5.96
mst^{C}	431	5	–	✓	–	–	–	✓	161	28	17	44	33	10	3	1.89
perimeter^{C}	486	12	–	✓	–	–	–	✓	44	10	69	41	13	1	0	1.64
power^{C}	618	17	–	–	–	–	–	✓	83	30	26	75	26	5	0	6.04
treeadd^{C}	249	2	–	–	–	–	–	✓	46	16	0	19	0	0	0	0.42
tsp^{C}	617	12	–	–	✓	–	–	✓	78	9	2	32	6	0	0	3.86
voronoi^{C}	1 151	40	✓	–	–	–	–	✓	✗	✗	38	101	57	44	0	21.35

Semantic type-checking with respect to Syntactic type-checking

Pros

+ more expressive invariants

+ works on unmodified programs

+ fully static, no execution overhead

Cons

- needs deeper code understanding

- elaborate analysis

- may report false alarms

Comparison with state-of-the-art tool CheckedC

Bench	**#LoC**	**#Fun**	**Code patterns**						**CC+3C**		**Spec#lines**		**#Alarms**			**Time**
Olden			BS	DU	NLI	FAM	IP	P?	man	gen	man	orig	orig	final	true	(s)
bh^{C}	2 107	30	–	✓	–	–	–	✓	181	48	27	144	39	3	1	26.04
bisort^{C}	356	11	–	✓	–	–	✓	✓	92	34	26	29	9	0	0	2.18
em3d^{C}	693	17	–	–	✓	–	–	✓	158	88	52	53	42	15	0	6.48
health^{C}	485	13	–	–	–	–	–	✓	99	57	39	57	16	4	0	5.96
mst^{C}	431	5	–	✓	–	–	–	✓	161	28	17	44	33	10	3	1.89
perimeter^{C}	486	12	–	✓	–	–	–	✓	44	10	69	41	13	1	0	1.64
power^{C}	618	17	–	–	–	–	–	✓	83	30	26	75	26	5	0	6.04
treeadd^{C}	249	2	–	–	–	–	–	✓	46	16	0	19	0	0	0	0.42
tsp^{C}	617	12	–	–	✓	–	–	✓	78	9	2	32	6	0	0	3.86
voronoi^{C}	1 151	40	✓	–	–	–	–	✓	✗	✗	38	101	57	44	0	21.35

Semantic type-checking with respect to Syntactic type-checking

Pros

+ more expressive invariants
+ works on unmodified programs
+ fully static, no execution overhead

Cons

- needs deeper code understanding
- elaborate analysis
- may report false alarms

Conclusion

Contributions

- **Novel type system** for **spatial memory safety** of low-level code
    - Types specify layout and content of memory
- **Automatic type checking** using abstract interpretation
- Evaluation on challenging **low-level code patterns**

`https://codex.top`

Future work

- Deal with temporal memory safety
- Improve precision for array and string types
- Partially infer type specifications