Precise and efficient memory analysis for low level languages

Julien Simonnet
CEA LIST

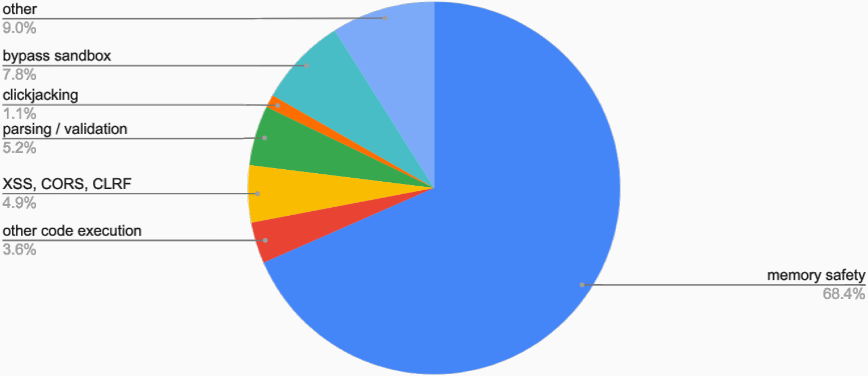
Matthieu Lemerre
CEA LIST

Mihaela Sighireanu
LMF

April 3, 2024

Working Group: "Formal Methods for Security" (GT MFS)

Memory safety



CVE vulnerabilities in Flash Player: `https://ultrasaurus.com/2019/12/memory-safety-necessary-not-sufficient`

~~null pointer dereferencing~~ ~~array out of bounds~~ ~~use after free~~

Low-level languages

Common idioms

- Hand-coded variant types
- Separated array and size
- Bitwise operations on addresses

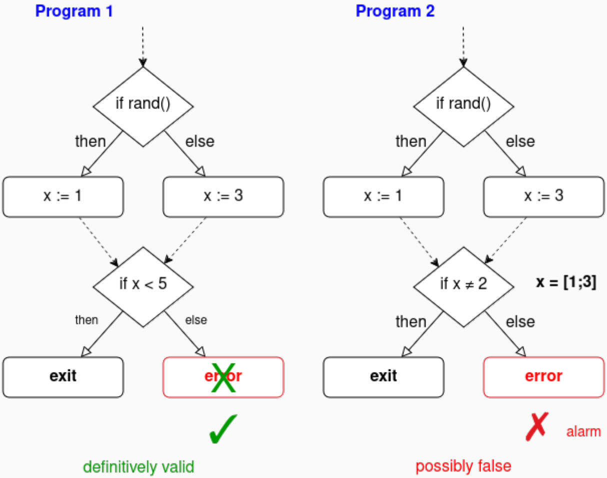
Objective

Develop an automated static analysis method:

- Efficient and precise
- For the verification of low-level programs without modification
- Ensuring spatial memory safety

Abstract interpretation

Abstract interpretation is an automatic verification method based on abstract domains that over-approximate sets of concrete states.



Our approach

Static analysis using abstract interpretation with a domain based on a new **dependent** type system, which provides:

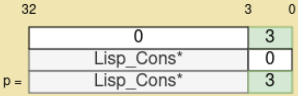
- Expressive invariants over memory
- Cheap analysis operations
- Easy configuration
- Modular (per-function) analysis

Checking type safety (by abstract interpretation) garantees spatial memory safety

Examples

Example 1 : Bit-stealing

```c
struct Lisp_Cons {
  int car ;
  int cdr ;
};
...
if ((p & 7) == 3)
  (Lisp_Cons*)(p-3)->cdr = p;
```



Valid program that uses bit-stealing

Example 2 : Buffer

```c
struct {
  int size ;
  char* buffer;
};
...
x->size = 3;
x->buffer = malloc(3);
for (i=0;i<5;i++)
  x->buffer[i] = 0; ← alarm 1
x->size++; ← alarm 2
```

The program contains:

1. A store outside the array
2. A type error

Record & array types

Types represent a memory layout.

Record types $\tau_1 \times \tau_2$ and **array types** $\tau[e]$ concatenate types.

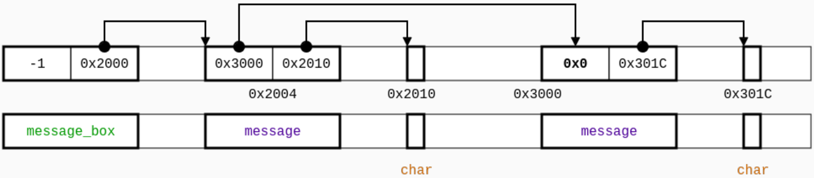
```
def int := byte[4]
def char := byte

def message :=
   message* ×
   char*

def message_box :=
  byte[4] ×
  message*
```

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

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



Refinement types

Types also represent values.

Values in a **refinement type** τ `with` p fulfill predicate p.

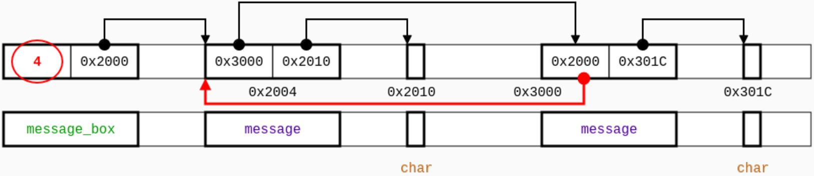
```
def int := byte[4]
def char := byte

def message :=
   message⋆ ×
   char⋆

def message_box :=
  byte[4] with self >= 0 ×
  message⋆
```

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

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



Existential types

Existential types $\exists\alpha : \tau_1.\tau_2$ introduce a **symbolic variable** α of type τ_2.

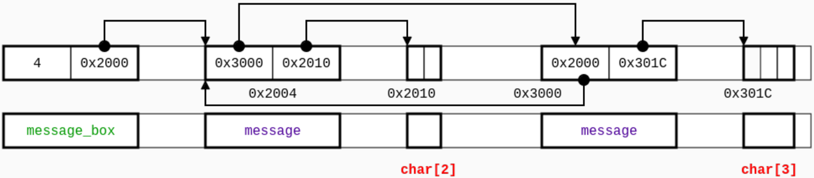
```
def int := byte[4]
def char := byte

def message :=
∃ len:byte[4] with self >= 0.
   message⋆ ×
   char[len]⋆

def message_box :=
  byte[4] with self >= 0 ×
  message⋆
```

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

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



Parameterized types

Parameterized types $\tau_a(e_1, \dots, e_n)$ use symbolic variables as parameters.

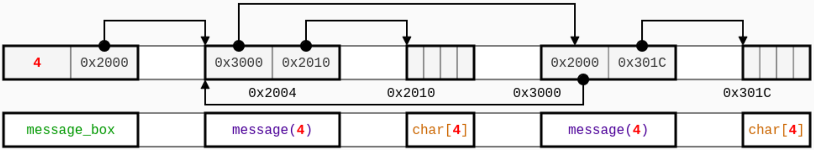
```
def int := byte[4]
def char := byte

def message(len:int) :=
   message(len)⋆ ×
   char[len]⋆

def message_box :=
∃ mlen:byte[4] with self >= 0.
  byte[4] self = mlen ×
  message(mlen)⋆
```

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

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



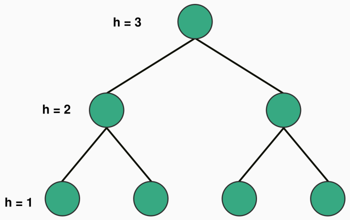
Union types $\tau_1 \cup \tau_2$ specify that a value belong to one type or the other (or both).

```
def node(h) :=
  byte[4] ×
  ((node(h-1)⋆ × node(h-1)⋆) with h > 0
  ∪ (nullptr × nullptr) with h <= 0)

def nodeptr :=
  ∃ h:byte[4] with self > 0. node(h)⋆
```

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

This specifies a perfect binary tree



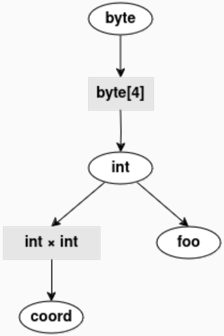
Nominal type system

Pointer types $\eta\star$ point to a **name** η

```
def int := byte[4]
def coord := int × int
def foo := int
```

Derivation rules

- $(|\texttt{coord}\star|) \subseteq (|\texttt{int}\star|)$
- $(|\texttt{foo}\star|) \subseteq (|\texttt{int}\star|)$
- $(|\texttt{coord}\star|) \cap (|\texttt{foo}\star|) = \emptyset$



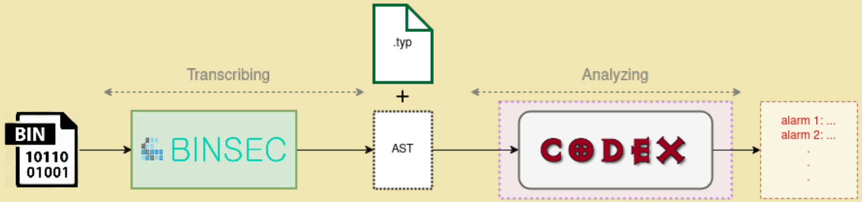
Complete type system

Our complete system is the following:

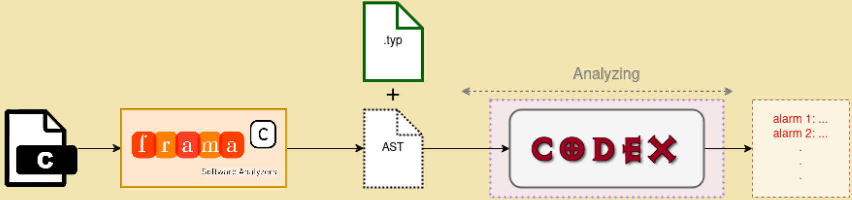
τ	$::=$	`byte`	(basic type of a single byte)
	$\mid$	η	(type name)
	$\mid$	$\eta\star$	(non null pointer type)
	$\mid$	$\tau_1 \times \tau_2$	(product type)
	$\mid$	τ **`with`** e	(refinement type)
	$\mid$	$\tau[e]$	(array type)
	$\mid$	$\tau_a(e_1, ..., e_n)$	(application of a constructor)
	$\mid$	$\exists\alpha : \tau_1.\tau_2$	(existential type)
	$\mid$	$\tau_1 \cup \tau_2$	(union type)
	$\mid$	$\tau_1 \bigwedge \tau_2$	(intersection type)
τ_a	$:=$	$\eta(\alpha : \tau_1).\tau_2$	(type constructor)

Codex, an automatic interprocedural analysis engine

Binary code analysis



C code analysis



Evaluation

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	#Entry	Code patterns						#Spec		#Alarms			Time
				BS	DU	NLI	FAM	IP	P?	gen	man	gen	final	true	(s)
OS	Contiki	329	12	–	–	–	–	–	✓	19	14	16	2	0	1.33
OS	QDS^{bin}	401	3	–	✓	✓	–	–	✓	83	83	18	0	0	1.28
OS	RBTree Linux	1 111	2	–	–	–	–	✓	✓	29	17	5	2	0	0.46
Emacs	list^{bin}	464	8	✓	✓	–	–	–	✓			-	0	0	3.03
Emacs	string^{bin}	109	5	✓	✓	✓	–	–	✓		73	-	5	0	3.20
Emacs	buffer^{bin}	42	3	✓	✓	–	✓	–	✓			-	0	0	3.12
Shapes	Graph	155	7	–	–	–	–	–	✓	26	14	0	0	0	0.79
Shapes	Javl	920	9	–	–	–	–	–	✓	37	34	10	1	1	0.70
Shapes	Kennedy	197	6	–	–	–	–	✓	✓	44	24	6	0	0	0.74
Shapes	RBtree	978	7	–	–	–	–	–	✓	32	18	56	16	0	0.42
Shapes	(6-)Other	5 742	19	–	–	–	–	–	✓	113	50	27	5	0	3.79

Comparison with CheckedC

CheckedC is the state-of-the-art verification tool by Microsoft.

Case studies (Olden)	#LoC	#Entry	Code patterns						CC+3C		#Spec		#Alarms			Time
			BS	DU	NLI	FAM	IP	P?	man	gen	man	gen	gen	final	true	(s)
bh^{C}	2 107	30	–	✓	–	–	–	✓	181	18	27	144	9	3	1	26.04
bisort^{C}	356	11	–	✓	–	–	✓	✓	92	34	29	29	9	0	0	2.18
em3d^{C}	693	17	–	–	✓	–	–	✓	158	88	50	53	42	15	0	6.48
health^{C}	486	12	–	–	–	–	–	✓	99	57	37	57	14	4	0	5.96
mst^{C}	431	6	–	✓	–	–	–	✓	161	28	16	44	33	10	3	1.89
perimeter^{C}	486	7	–	✓	–	–	–	✓	44	10	26	41	13	1	0	1.64
power^{C}	618	17	–	–	–	–	–	✓	83	20	26	75	29	4	0	6.04
treeadd^{C}	249	2	–	–	–	–	–	✓	46	16	0	19	0	0	0	0.42
tsp^{C}	617	11	–	–	✓	–	–	✓	78	10	0	32	0	0	0	3.86
voronoi^{C}	1 151	40	✓	–	–	–	–	✓	✗	✗	37	101	57	49	0	21.35

Codex vs. CheckedC

Pros

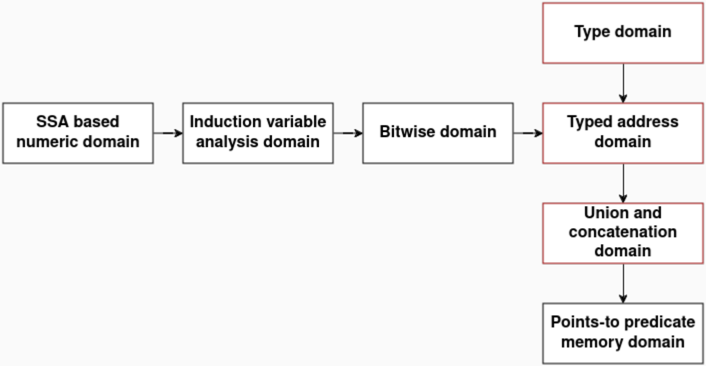
+ could still crash in CheckedC
+ overhead in CheckedC
+ detects null pointer deferences
+ works on unmodifed programs
+ supports union types

Cons

- specification need an understanding of programs
- still some imprecision to solve (with the tool)

Implementation overview

This work is combined with additionnal work on abstract interpretation:



As well as support for interprocedural and higher order analysis.

Conclusion

Summary

- Type checking by abstract interpretation
  - Automated inference from assembly to typed assembly
  - Automated proof of spatial memory safety for C and machine code
- Novel type system with many interesting properties
- Evaluation on challenging low-level code patterns

Future work

- Improve automation (infer specification)
- Improve precision (domains for strings and arrays)
- Verify temporal memory safety (use-after-free errors)
- Resubmit paper 😥

Questions?