Spatial Memory Safety with Dependent Types in CODEX

Julien Simonnet Matthieu Lemerre Mihaela Sighireanu

September 20, 2024

1 Getting started

This document provides a user guide for the static analysis based on the dependent nominal type system presented in [2] and implemented in the CODEX tool [1].

The analysis detects memory vulnerabilities (for example, null pointer dereferencing, out of bounds memory access, unsatisfiability of a memory invariant) in C or binary programs. For this, the analysis requires a specification of the correct memory layout of the program. This specification is given as a set of type definitions. The types used extend C types with specific constructs including type refinement by a predicate, unbounded unions or parameterized type definitions.

The analysis is inter-procedural, i.e., it could analyse functions separately. For this, the specification file may include function declarations (profile) using the type defined.

In the following, we introduce the specification language for types, the options and outputs of the analysis, as well as a usage methodology through several analysis examples. The full concrete syntax for type specification is given in Section A. The formal presentation of the analysis is given in [2].

1.1 First example in C

Consider the C code in Listing 1. It has been extracted from the code of an OS where it was used to encode messages in an IPC (Inter-Processs Communication) mechanism. The code defines two record types representing a list of messages (`struct message`) and a message box (`struct message_box`) storing the head of a message list. The function `zeros_buffer` fills all the buffers in a message box with zeros.

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

struct message_box {
 7   int length;
 8   struct message *first;
 9 };
10
11 void zeros_buffer(struct message_box *box) {
12
13     struct message * first = box->first;
14     struct message * current = first;
15
16     int length = box->length;
17
18     do {
19         for (int i = 0; i < length; i++) {
20             current->buffer[i] = 0 ;
21         }
22         current = current->next;
23     } while(current != first) ;
24 }
```

Listing 1: First example ex1.c

To run our analysis on this example, we provide as input to CODEX the C code file `ex1.c` and a specification file `ex1.typ` defining the correct memory layout. The simplest specification we could provide is given in Listing 2; it includes the first lines of the program (from line 1 to 9) defining the C types used, as well as the declaration of the profile of `zeros_buffer`.

Listing 2: Specification file ex1.typ

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

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

void zeros_buffer(struct message_box *box);
```

If CODEX has been successfully installed (see `README.md` file in the distribution) then the binary `frama-c-codex` is reachable in the command line and may be called using the following command line

Listing 3: Command line to run our analysis

```
$ dune exec frama_c_codex -- -codex -codex-use-type-domain \
  -codex-debug 0 -codex-exp-dump ex1.cdump -machdep gcc_x86_32 \
  ex1.c -codex-html-dump ex1.html -codex-type-file ex1.typ \
  -main zeros_buffer
```

This command line specifies, in addition to the file including the C code (argument `ex1.c`) and the specification file (option `-code-type-file ex1.typ`),

- the trigger of the type analysis by option `-codex-use-type-domain`,
- the machine dependent architecture by option `-machdep gcc_x86_32`
- the basic verbosity level for (debug) messages printed by option `-codex-debug 0`,
- the function representing the entry point of the analysis by option `-main zeros_buffer`,
- the name of files to which is outputted the result of the analysis in text form (option `-codex-exp-dump ex1.cdump`) and in HTML format (option `-codex-html-dump ex1.html`).

The output of the analysis in the file `ex1.cdump`[1] contains the following lines divided in five sections:

```
1  ex1.c:13.29-39: `box->first' -> {0} or ([1..0xFFFFFFFF] : (Name(struct message))[{0}].0*)
2  ex1.c:13.29-32: `box' -> {0} or ([1..0xFFFFFFFF] : (Name(struct message_box))[{0}].0*)
3  ...
4  ... See full output in next section
5  ...
6  ex1.c:23.23-28: `first' -> {0} or ([1..0xFFFFFFFF] : (Name(struct message))[{0}].0*)
7  Unproved regular alarms:
8  ex1.c:13: Memory_access(box->first, read) {true;false}
9  ex1.c:20: Memory_access(current->buffer, read) {true;false}
10 ex1.c:20: Memory_access(*(current->buffer + i), write) {true;false}
11 ex1.c:22: Memory_access(current->next, read) {true;false}
12 Unproved additional alarms:
13 + ex1.c:20 : { *(current->buffer + i) = (char)0; } -> store_param_nonptr ;
14 Proved 1/5 alarms
15 Unproved 4 regular alarms and 1 additional alarms.
```

- The first part, before "`Unproved regular alarms`" (excerpt given at lines 1 to 6), gives the abstract states computed by the analysis at each program point of `zeros_buffer`, the function analyzed. The meaning of these lines is explained in Section 2.2.
- The second part starts with "`Unproved regular alarms`" (lines 7 to 11) and lists the potential memory vulnerabilities detected. For instance, at line 8 above, the memory access done by the read of `box->first` at line 13 in `ex1.c` is reported to be a potential null pointer dereferencing. Similar read vulnerabilities are listed at lines 20 and 22. In addition, at line 20 of `ex1.c`, the write at the memory address `(current->buffer + i)` is reported to be a potential out-of-bounds access.

  Notice that, once an alarm has been reported for the access to an address, the possibly invalid pointer is assumed to be valid to prevent the generation of redundant alarms

[1]The format of files `.cdump` allows to navigate in the source code using emacs compilation mode by click on the location part (first characters) of each line.

for this address. For instance, the memory access at line 13 is repeated at line 16 of `ex1.c`, but it is not reported again as a memory vulnerability.

- The third part starts with "`Unproved additional alarms`" (lines 12 and 13 above) and reports vulnerabilities inside the program's expressions. These expressions appear in the analysis's transfer functions. Because a transfer function may be called several times, these alarms may rbe epeated and may include some alarms of the second part above. In our example, the analysis reports a vulnerable memory write inside a buffer at line 20 of `ex1.c`, which is exactly the third regular alarm, i.e., reported at line 10 of the listing above.

- The fourth part, starting with "`Proved`", gives the ratio of memory accesses proved safe over the total number of memory accesses.

- The fifth part, starting with "`Unproved`", summarizes the total number of alarms already explained in the second and third parts.

The regular alarm reported at line 13 of `ex1.c` is a false alarm for our code because the `zeros_buffer` function is always called with a not null argument. To specify that the parameter `box` is always not null, we refine the initial specification by replacing '`*`' by '`+`' before `box` in the declaration of `zeros_buffer` function as follows:

Listing 4: Refined specification ex1-1.typ

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

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

void zeros_buffer(struct message_box +box);
```

By running the analyzer with the new specification `ex1-1.typ` for the code `ex.c`, we obtain the output below, where the memory accesses at lines 13 and 16 are reported to be safe.

```
...
Unproved regular alarms:
ex1.c:20: Memory_access(current->buffer, read) {true;false}
ex1.c:20: Memory_access(*(current->buffer + i), write) {true;false}
ex1.c:22: Memory_access(current->next, read) {true;false}
Unproved additional alarms:
+ ex1.c:20 : { *(current->buffer + i) = (char)0; } -> store_param_nonptr ;
Proved 2/5 alarms
Unproved 3 regular alarms and 1 additional alarms.
```

Four memory accesses are still unproved because the memory layout has a stronger invariant than the one given by the new specification. We provide in Section 4 the full process of proving the spatial memory safety for `zeros_buffer` by refining the specification of the memory layout.

1.2 First program in binary

We can also run our analysis on a binary file which shall be executable (have all addresses resolved). To illustrate this analysis on our running example, we add the following `main` function to Listing 1 in order to obtain an executable file `ex1_full.c`

```c
int main(void) {
  // Allocates the message box
  struct message_box *box = malloc(sizeof(struct message_box));
  box->length = 20;
  box->first = NULL;
  for (int i = 0; i < 10; i++) {
    struct message *lst = malloc(sizeof(struct message));
    lst->buffer = malloc(sizeof(char) * box->length);
    lst->next = box->first;
    box->first = lst;
  }

  // Fills the content of message box with zeros
  zeros_buffer(box);

  return 0;
}
```

Listing 5: First example with main function

The source code is compiled using the following command line to generate the executable `ex1_full.exe`:

Listing 6: Command line to compile `ex1_full.c`

```
$ gcc -O0 -o ex1_full.exe -m32 -fno-stack-protector  ex1_full.c
```

The compilation is done using `gcc` and the following options:

- option `-O0` limits the optimizations during the compilation in order to keep the executable close to the original program and this helps the user to interpret the messages sent by the analyzer;
- option `-m32` compiles the program for 32-bits architecture, which is the only one supported currently by our analysis;
- option `-fno-stack-protector` removes stack protection against stack writing overflow to easily compute the addresses on stack.

If CODEX has successfully been installed, the binary `binsec-codex` is reachable in the command line and may be called as follows for our example:

Listing 7: Command line to compile C code

```
$ dune exec binsec_codex -- -codex ex1_full.exe \
  -entrypoint zeros_buffer -codex-type-file ex1.typ
```

The command sets the name of the function used as entry point of the analysis (option `-entrypoint`) and the name of the specification file (option `-codex-type-file ex1.typ`). The command produces the following output:

```
1  ...
2  [codex:result] [0xfedcba98] Fixpoint reached at iteration 1.
3  [codex:result] [0xfedcba98]
4                 Over ################################################################
5  [codex:result] [0xfedcba98] Nodes in the graph (with call stack): 37
6  [codex:result] [0xfedcba98] Number of instructions (no call stack): 37
7  [codex:result] [0xfedcba98] End of analyze log
8  ### Alarms ###
9
10 == _none_ ==
11 -alarm count-,-alarm- invalid_load_access,0,1,[0x000011e6]
12 -alarm count-,ptr_arith,26,4,[0x000011ac 0x000011cc 0x000011d2 0x000011ee]
13 -alarm count-,store_param_nonptr,1,1,[0x000011d4]
14
15
16 -total alarm count-,6
17
18 Analysis time: <dummy>
19 Total alarms: 6
20 Preprocessing time : 0.003372s
21 Analysis time : 0.105409s
```

- The first part, before "`### Alarms ###`", was shorten as it is a full journal of the operations done by the binary analysis.

- The second part, starting with "`### Alarms ###`", reports the vulnerabilities found. For instance, the line 11 reports that the instruction at address `0x000011e6` attempts an invalid read in the memory (similar to a null pointer dereferencing in C). To obtain the mnemonic of an instruction at some address, you could use `objdump` as explained below. At line 12, the alarm `ptr_arith` points out that a pointer arithmetic operation may return an invalid pointer. This is an unnecessary test done by the analysis because the code may not use this address, but it is useful to prevent wrong memory accesses. Finally, the vulnerability `store_param_nonptr` reported at line 13 corresponds to the one reported in the analysis of the C code for the out-of-bound memory write at line 20.

- The third part, starting with "`Total alarms`", summaries the number of alarms found.

- The fourth part, starting with “`Preprocessing time`”, gives the time used for parsing the specification and building the initial memory configuration fixed by the executable file.
- The fifth part, starting with “`Analysis time`”, gives the time actually spent analyzing the binary program.

When interpreting the output of the binary analysis, it is useful to decompile the executable in order to obtain the instructions at each address. This may be obtained by calling `objdump` on the program `ex1_full.exe` as follows:

```
$ objdump -d ex1_full.exe > ex1_full.objdump
```

For instance, the file `ex1_full.objdump` includes the following code corresponding to the function `zeros_buffer`:

```
...
00001199 <zeros_buffer>:
    1199:       55                      push   %ebp
    119a:       89 e5                   mov    %esp,%ebp
    119c:       83 ec 10                sub    $0x10,%esp
    119f:       e8 00 01 00 00          call   12a4 <__x86.get_pc_thunk.ax>
    11a4:       05 34 2e 00 00          add    $0x2e34,%eax
    11a9:       8b 45 08                mov    0x8(%ebp),%eax
    11ac:       8b 40 04                mov    0x4(%eax),%eax
    11af:       89 45 f4                mov    %eax,-0xc(%ebp)
    11b2:       8b 45 f4                mov    -0xc(%ebp),%eax
    11b5:       89 45 fc                mov    %eax,-0x4(%ebp)
    11b8:       8b 45 08                mov    0x8(%ebp),%eax
    11bb:       8b 00                   mov    (%eax),%eax
    11bd:       89 45 f0                mov    %eax,-0x10(%ebp)
    11c0:       c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%ebp)
    11c7:       eb 12                   jmp    11db <zeros_buffer+0x42>
    11c9:       8b 45 fc                mov    -0x4(%ebp),%eax
    11cc:       8b 50 04                mov    0x4(%eax),%edx
    11cf:       8b 45 f8                mov    -0x8(%ebp),%eax
    11d2:       01 d0                   add    %edx,%eax
    11d4:       c6 00 00                movb   $0x0,(%eax)
    11d7:       83 45 f8 01             addl   $0x1,-0x8(%ebp)
    11db:       8b 45 f8                mov    -0x8(%ebp),%eax
    11de:       3b 45 f0                cmp    -0x10(%ebp),%eax
    11e1:       7c e6                   jl     11c9 <zeros_buffer+0x30>
    11e3:       8b 45 fc                mov    -0x4(%ebp),%eax
    11e6:       8b 00                   mov    (%eax),%eax
    11e8:       89 45 fc                mov    %eax,-0x4(%ebp)
    11eb:       8b 45 fc                mov    -0x4(%ebp),%eax
    11ee:       3b 45 f4                cmp    -0xc(%ebp),%eax
    11f1:       75 cd                   jne    11c0 <zeros_buffer+0x27>
    11f3:       90                      nop
    11f4:       90                      nop
    11f5:       c9                      leave
    11f6:       c3                      ret
...
```

2 C analysis interface

In this section, we give an overview of the interface of our analyzer for the analysis of C code, i.e., the options of the analysis and the output it produces. The command line to call the analysis has the following form:

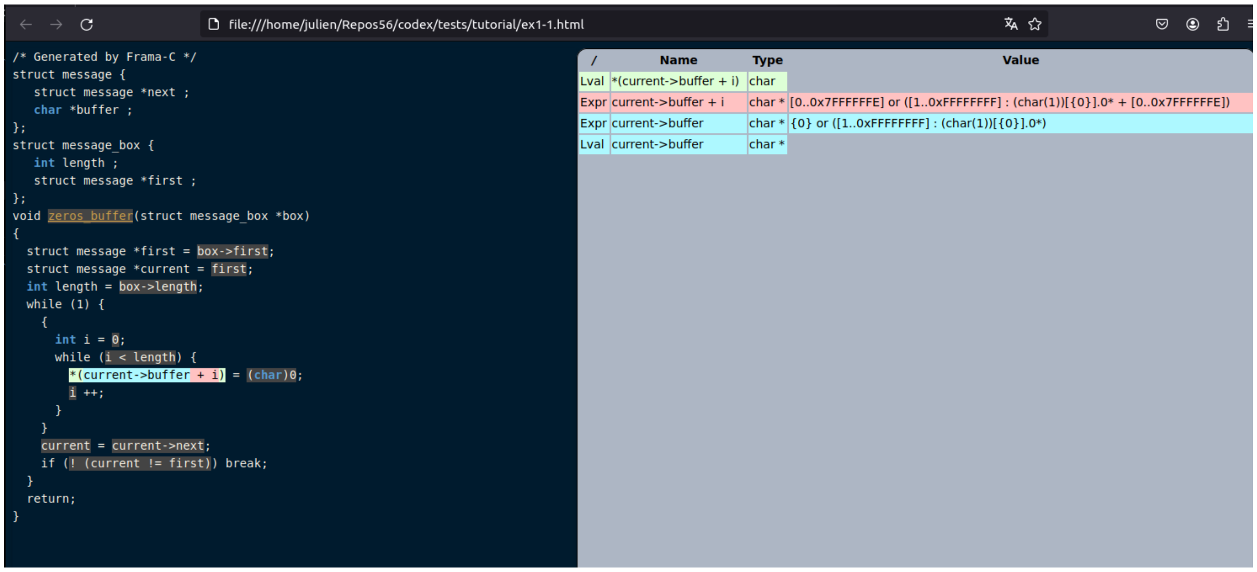
Listing 8: Command line to run our analysis

```
$ dune exec frama_c_codex -- -codex <list of options> input_code.c
```

2.1 Options overview

The analysis may be tuned using the following options:

- Option `-codex-use-type-domain` selects our analysis that employs the domains based on types (we used it in all our examples until now).
- Option `-codex-type-file fname` fixes `fname` as input specification file; by convention, the extension of this file is `.typ`.
- Option `-codex-use-loop-domain` triggers the usage of a specific abstract domain, called induction variable analysis domain, which allows a precision gain when analyzing loops with inductive invariants.
- Option `-codex-debug n` activates the level `n` of debug messages (0 is less verbose than 3).
- Option `-focusing` includes the points-to predicate domain from [3], which keeps track of the content of the memory during some analysis steps in order to increase the precision.
- Option `-codex-serialize-cache`, used in combination with `-focusing`, activates the join and widen operations in the points-to memory domain from [3]; this increases precision, but also may degrade slightly the time performance or may produce some crashes.
- Option `-codex-use-weak-types` enables the usage of "weak types" for the dynamically allocated values not yet initialized. This permit to infer the type of the allocated value semantically during the analysis of the initialization code, rather than syntactically using the type used at the allocation point.
- Option `-codex-html-dump fname.html` sets `fname.html` as output file for an interesting graphical representation of our analysis that details the computed abstract states at each point of the analyzed functions. Figure 1 illustrates this output on our running example.



file:///home/julien/Repos56/codex/tests/tutorial/ex1-1.html

```
/* Generated by Frama-C */
struct message {
   struct message *next ;
   char *buffer ;
};
struct message_box {
   int length ;
   struct message *first ;
};
void zeros_buffer(struct message_box *box)
{
  struct message *first = box->first;
  struct message *current = first;
  int length = box->length;
  while (1) {
    {
      int i = 0;
      while (i < length) {
        *(current->buffer + i) = (char)0;
        i ++;
      }
    }
    current = current->next;
    if (! (current != first)) break;
  }
  return;
}
```

/	Name	Type	Value
Lval	*(current->buffer + i)	char	
Expr	current->buffer + i	char *	[0..0x7FFFFFFE] or ([1..0xFFFFFFFF] : (char(1))[{0}].0* + [0..0x7FFFFFFE])
Expr	current->buffer	char *	{0} or ([1..0xFFFFFFFF] : (char(1))[{0}].0*)
Lval	current->buffer	char *	

Figure 1: HTML output for our example

- Option `-main fun` set `fun` as entry point of the analysis starting with the specification of this function. The functions called by the entry point are analyzed in an inter-procedural way, i.e., their specification is used to create a summary and this summary is used by the analysis at each call point. This behaviour is disabled for functions specified to be `inline` (see Section 4).

2.2 Output overview

Abstract state: Our analysis first outputs the abstract states computed at each program point. We provide a semantic to this output by illustrating it on the output produced for the running example of the first section.

For instance, the abstract state computed for line 13 of the file `ex1.c` is given below. It states that the value stored by the field `first` at address given by `box` is either 0 or a value in the interval `[1..0xFFFFFFFF]` (the interval domain is used to abstract sets of numerical values). In addition to the interval of values, the type of `box->first` is reported to be a pointer at offset 0 inside a region starting at an address of type `struct message`. Notice that `Name()[0]` is used to unify the notation for simple allocated pointers and the start of an array.

```
ex1.c:13.29-39: `box->first' -> {0} or ([1..0xFFFFFFFF] : (Name(struct message))[{0}].0*)
```

For the variables of numeric type, like `i` in Listing 1, the abstract value computed at line 19 is a 32-bits integer value, strictly less than `length`, so the computed interval is `[0..0x7FFFFFFE]`. For this reason the expression used at line 20, `current->buffer + i`, is associated with the abstract value denoting the interval `[0..0x7FFFFFFE]` (when `current->buffer` is 0) or a pointer in the interval `[1..0xFFFFFFFF]`. The type of the last pointer is pointer to `char` (where `char` is of 1-byte size) inside a region at a distance from the start of the region in the interval `[0..0x7FFFFFFE]`.

```
ex1.c:19.24-32: ‘i < length’ -> {0; 1}
ex1.c:19.24-25: `i' -> [0..0x7FFFFFFF]
...
ex1.c:20.12-27: `current->buffer + i' -> [0..0x7FFFFFFE]
                                   or ([1..0xFFFFFFFF] : (char(1))[{0}].0* + [0..0x7FFFFFFE])
ex1.c:20.12-27: `current->buffer' -> {0} or ([1..0xFFFFFFFF] : (char(1))[{0}].0*)
ex1.c:20.12-19: `current' -> {0} or ([1..0xFFFFFFFF] : (Name(struct message))[{0}].0*)
```

The following output line states that the value of the expression `current != first` at line 23 is either 0 or 1:

```
ex1.c:23.12-28: `current != first' -> {0; 1}
```

Reported alarms: The following parts of the output concern the potential vulnerabilities detected. The location of a reported vulnerability is given by the line in the source code and the columns of the memory access expression in this line. In the analysis of C code, two types of alarms are reported:

Regular alarms: mostly related to memory vulnerabilities or arithmetic errors. Once reported, these vulnerabilities are assumed to be false in the remainder of the analysis in order to avoid redundant alarms.

- `Memory_access(addr, read)` signals that an attempt to read the memory at `addr` may be invalid because the pointer `addr` may be invalid.
- `Memory_access(addr, write)` signals that an attempt to write in the memory at `addr` may be invalid because the pointer `addr` may be invalid.
- `Division_by_zero` signals that a division may use a zero divider.

Additional alarms: are related to vulnerabilities found inside the transfer functions and can be generated multiple times for the same reason.

- `s -> load_param_nonptr` indicates that an attempt to read the memory in statement `s` may be an invalid pointer, usually because the pointer reads out of bounds.
- `s -> store_param_nonptr` indicates that an attempt to write in the memory in statement `s` may be an invalid pointer, usually because the pointer writes out of bounds.
- `s -> array_offset_access` indicates that an attempt to access the memory in statement `s` in some operations using arrays may be invalid.
- `s -> serialize` indicates that a join (or widen) operation tried to join (respectively widen) two abstract pointers of different types.
- `s -> weak-type-use` indicates that a recently allocated pointer whose type is not yet known is potentially used to access the memory.

- `s -> typing_store` indicates that the type of a value may be incorrect for the following reasons:
  - the type of the value written in memory may be wrong,
  - the type of the parameter given does not correspond to the pre-condition of a function summary,
  - the returned value of a function does not have the correct type.
- `s -> free-on-null-address` indicates that a potentially null pointer was deallocated by statement `s`.
- `Return -> weak-type-leak` indicates that the type of a recently allocated pointer is not known. This happens at the exit point of a function, when the analysis detects that a pointer was not written in the memory, neither returned from the function, thus leading to a memory leak.
- `Return -> incompatible-return-type` signals that the type of the returned value does not correspond to the function profile in the specification when the specified return type is `void` or the body does not return a value for a non-void return type.

3 Binary analyzer interface

In this section, we give an overview of the interface of the binary analysis: the options given in the command line and the meaning of the output produced. The analysis is called with the following line:

Listing 9: Command line to compile C code

```
$ dune exec binsec_codex -- -codex <list of options> input_file.exe
```

3.1 Options overview

The binary analysis may be tuned using the following options, some of them being shared with the C code analysis:

- Option `-codex-type-file fname` fixes `fname` as input specification file; by convention, the extension of this file is `.typ`.
- Option `-codex-use-loop-domain` triggers on the usage of a specific abstract domain, called induction variable analysis domain, which allows a precision gain when analyzing loops with inductive invariants.
- Option `-codex-debug-level n` activates the level `n` of debug messages (0 is less verbose than 3).
- Option `-no-focusing` removes the points-to predicate domain from [3], which keeps track of the content of the memory during few analysis steps which decreases precision.

- Option `-codex-serialize-cache`, used in combination with `-focusing`, activates the join and widen operations in the points-to memory domain from [3]; this increases precision, but also may degrade slightly the time performance or may produce some crashes.

- Option `-codex-use-weak-types` enables the usage of "weak types" for the dynamically allocated values not yet initialized. This permit to infer the type of the allocated value semantically during the analysis of the initialization code, rather than syntactically using the type used at the allocation point.

- Option `-hooks` allows us to stub some function calls or statements (given by their address) in the analysis. For instance:

  ```
  hooks = \
      0x0255d4=stop, \
      iniTabTrans=stop, \
      0x025598=stop, \
      0x100055=skip_to(0x10005f), \
      0x0242d7=nop, \
      0x02558f=return_unknown(int), \
  ```

  This indicates to the analysis to stop the exploration beyond instructions `0x0255d4` and `0x025598` and when reaching a call to function `iniTabTrans`. The hook `skip_to` states that the analysis shall skip statements from line `0x100055` to line `0x10005e`. The hook `return_unknown(int)` states that the statement at line `0x02558f` is replaced by a store of a random value of `int` type in register `ax`. As for hook `nop`, it works as a way to skip to the next address.

- Option `-config fname` sets `fname` as input configuration file; by convention, the extension of this file is `.ini`, in which the above options can be grouped instead being passed directly in the command line. Such configuration files may be used by several command lines.

3.2 Output overview

The binary analysis outputs a list of alarms of the following kinds:

- `invalid_load_access [addrs]` indicates that an attempt to read the memory in instructions at addresses `addrs` is possibly invalid because the pointer used is possibly null.

- `invalid_store_access [addrs]` indicates that an attempt to write in the memory in instructions at addresses `addrs` is possibly invalid because the pointer used is possibly null.

- `load_param_nonptr [addrs]` indicates that an attempt to read the memory is invalid because the pointer is out of bounds
- `store_param_nonptr [addrs]` indicates that an attempt to write in the memory is invalid because the pointer is out of bounds.
- `array_offset_access [addrs]` indicates that an attempt to do some operation using arrays was invalid.
- `serialize [addrs]` indicates that a join or widen operation tried to join respectively widen two abstract pointers of different types.
- `weak-type-use [addrs]` indicates that a recently allocated pointer whose type is not yet known is used to access the memory.
- `typing_store [addrs]` indicates that the type of a value is incorrect. This corresponds to a write of a value of wrong type or to a return value of wrong type.
- `free-on-null-address [addrs]` indicates that a possibly null pointer is deallocated.
- `weak-type-leak [addrs]` indicates that the type of a recently allocated pointer is not known. This happens at the exit point of a function, when the analysis detects that a pointer was not written in the memory, neither returned from the function, thus leading to a memory leak.
- `incompatible-return-type [addrs]` signals that the returned value does not correspond to the function profile given in the specification file.

4 Specification using dependent types

Our analysis does not need to modify the input programs with annotations. To specify the layout of the memory for which the memory vulnerabilities are avoided, the user has to provide a specification file. In this section, we present the specification language used by our analysis. This presentation is done gradually, starting with the automatic generation of a rough specification from the types used in the program (using CPROTO) and ending with a refined specification including most of the constructs of the specification language. The concrete syntax of our specification language is given in Section A.

4.1 Automatic generation from C programs

The concrete syntax of our specification language is based on the C syntax for type and function declarations. This helps to easily generate a first specification using CPROTO, a tool available in any GNU based distribution.

For instance, to call CPROTO on the code at Listing 5, we use the following command:

```
$ cproto -E 0 -f 3 -n -q -T -o ex1.typ ex1_full.c
```

The command produces the following output stored in the file `ex1.typ` (specified by the option `-o` above):

Listing 10: Specification produced with CPROTO from `ex1_full.c`

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

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

void zeros_buffer(struct message_box *box);
int main(void);
```

4.2 Defining type names

Our analysis is based on a nominal type system, like in C. This means that two values may not be equal if they have different types, even if these types correspond to the same domain of values.

The only predefined type name for our type system is `byte`, corresponding to a one byte. To define new type names, we employ the `type` definition, similar to `typedef` in C. For instance, the following specification defines a new type called `foo` to be an array of four `byte`, like the type `int`:

```
type char = byte;
type int  = byte[4];
type foo  = byte[4];
```

However, pointers to type `int` and `foo` may not alias because of the nominal feature of the type system.

The syntax used in C to define record type, like in Listing 4.1, is internally dealt as a syntactic sugar and translated to:

Listing 11: Type definitions obtained from the ones produced with CPROTO from `ex1_full.c`

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

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

After the equal sign, the `struct` type expression defines a record type with the same syntax as in C.

Our `type` construct is equivalent to a C type definition `typedef` without the attribute `may_alias` available for GNU C[2]. Therefore, the equality test fails between values of different types even if their value domain is the same, i.e., between value of type `foo` and `int` defined above. Like in C, our construct prevents writing at an address a value typed by a different type name than the address. This semantics of memory stores in presence of aliasing is called weak update in [2]. To make clear this behaviour of our type system, our specification language forbids the usage of `typedef`; if this construct is used, the parsing craches with an error message.

Another feature of our type system is that it allows only pointer to type names, like in C. Notice that in our running example from Listing 4.1, `struct message` is a type name.

4.3 Refined types

To introduce the refined types, let us consider the example at Listing 1. For the specification given at Listing 4.1, a possible memory layout is provided at the bottom of Figure 2: the first line illustrates a possible content for the memory while the second line gives the type of each memory region.

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

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

void zeros_buffer(struct message_box* box);
```

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

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

void zeros_buffer(struct message_box* box);
```

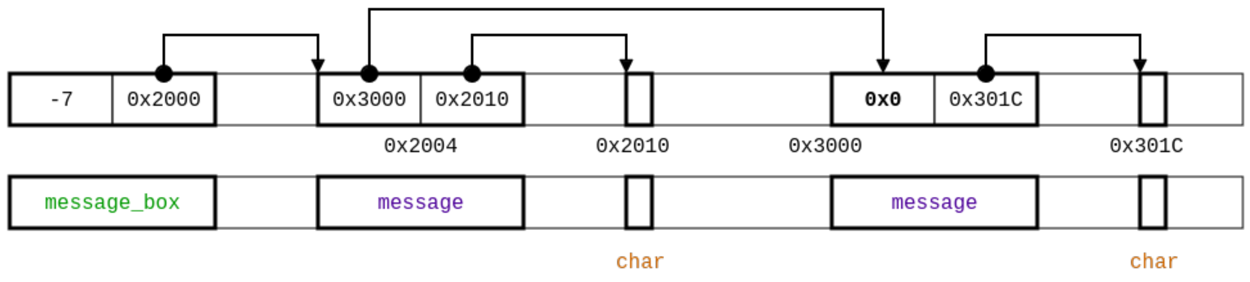


Figure 2: Memory layout allowed by the initial specification

This specification of the memory layout allows negative values for the `length` field of a `message_box` and a buffer in `message` containing a single `char`. Our analysis reports potential

[2]https://gcc.gnu.org/onlinedocs/gcc/Common-Type-Attributes.html

vulnerabilities while accessing `current->buffer` at line 20 in read and write.

```
Unproved alarms:
...
ex1.c:20: Memory_access(current->buffer, read) {true;false}
ex1.c:20: Memory_access(*(current->buffer + i), write) {true;false}
...
```

This vulnerability does not appear if the `length` field is positive. To specify this, we may use a refined type expression "`int with self >= 0`". Similarly, the vulnerability for memory read may be removed if the pointer `current` is not null, which may be expressed using the refined type "`struct message* with self != 0`". We propose to denote the above refined type expression for the non null pointers by the syntactic sugar "`struct message+`".

Moreover, to control the aliasing of the `length` field inside the `struct message_box` by a variable typed by an `int*`, we introduce a new type `integer` of 4-`byte`, different from `int`.

The new specification, given in Figure 3 integrates these changes. To recall the initial types, we reproduce the original type definitions in C at the right of the figure.

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

struct message_box {
  integer with self >= 0 length;
  struct message* first;
};

void zeros_buffer(struct message_box +box);
```

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

struct message_box {
  integer length;
  struct message *first;
};

void zeros_buffer(struct message_box *box);
```

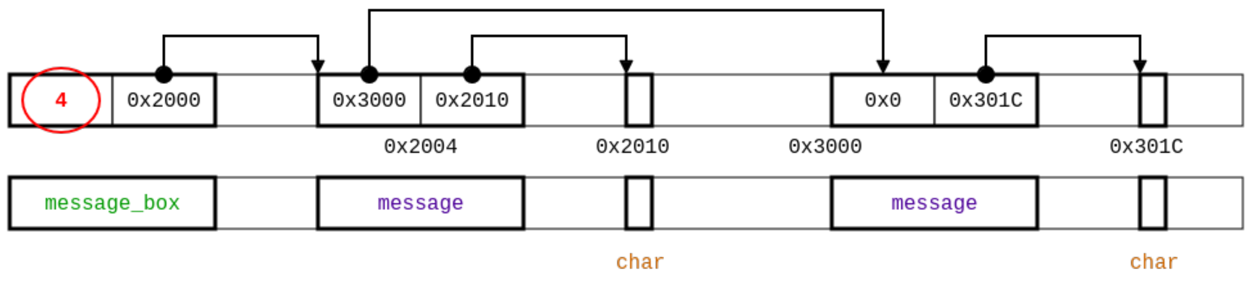


Figure 3: Refined type for length and non null pointer as function parameter

The new specification does not allow to remove the alarms reported for null pointer dereference while accessing pointer variables. By looking at both the code and the reported alarms, we identify that the list of `message` stored in `first` is circular. To specify this property, we set `next` field to be of type non null pointer, which over-approximates the circular list shape by a lasso shape (since the number of memory addresses is finite). Furthermore, we observe that the code employs both `buffer` and `first` fields as non null pointers.

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

struct message_box {
  integer with self >= 0 length;
  struct message+ first;
};

void zeros_buffer(struct message_box+ box);
```

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

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

void zeros_buffer(struct message_box* box);
```

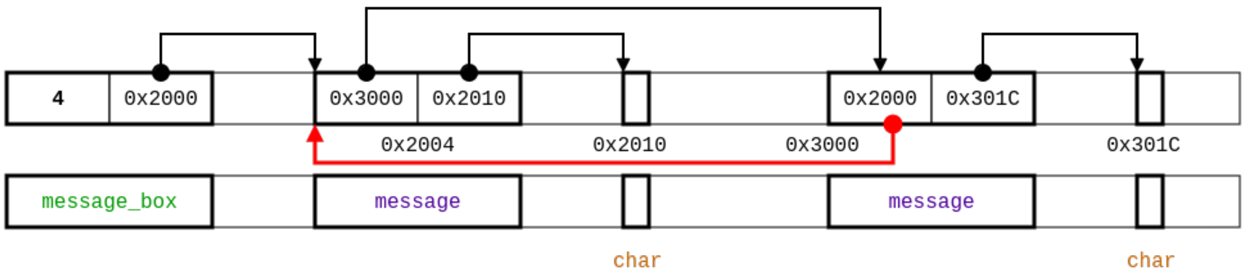


Figure 4: Non null pointers to specify a lasso shaped list

With the specification above, our analysis reports the following additional alarm for an out of bound access to the memory:

```
Unproved additional alarms:
+ ex1.c:20 : { *(current->buffer + i) = (char)0; } -> store_param_nonptr ;
```

Indeed, stating that `buffer` is not null does not mean that more than one `char` is allocated at this address.

4.4 Existential types

A way to remove the previous reported alarm on out of bound memory access is to specify that `buffer` is the start address of an array of some size. This is done using the existential type expression "`∃x:T. U`" which introduces a local variable x in a type U. In our example, the variable `len`, of positive integer value is introduced as the size of the allocated memory region at the address `buffer`. Therefore, buffers may contain multiple character not just one and, moreover, each buffer may have a different length. The resulting specification and an allowed memory layout are given by Figure 5. Therefore, the existential types allow to express local invariants in a type expression, i.e., invariants between the fields of the same memory region.

```
∃ len:integer with self >= 0.
struct message {
   struct message+ next;
   char[len]+ buffer;
};

struct message_box {
  integer with self >= 0 length;
  struct message+ first;
};

void zeros_buffer(struct message_box+ box);
```

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

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

void zeros_buffer(struct message_box* box);
```

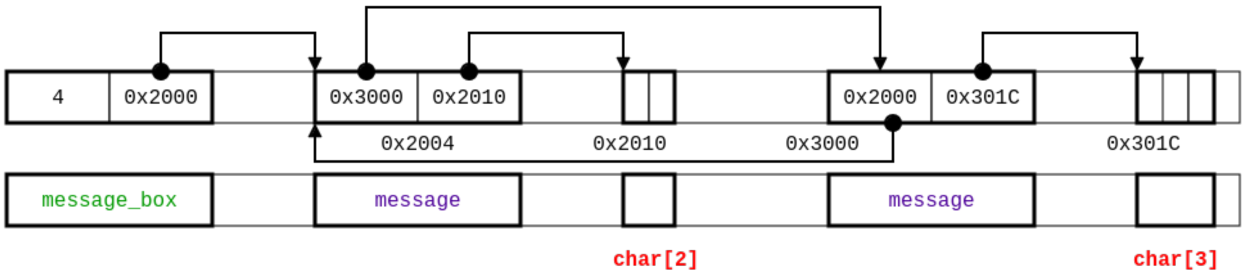


Figure 5: Existential type to specify an allocated memory buffer of some length

4.5 Parameterized types

The specification in Figure 5 does not remove the false alarm reported for array out of bound access because the function `zeros_buffer` expects that all the buffers in a message box have the same length given by the field `length` in `message_box`. This is not a local invariant for a memory region because the value stored in some memory region (field `length` in a region typed by `message_box`) is related with a value (length of a `buffer`) in another region of a different type (e.g., `message`). To exchange this information between memory regions, we specify that the type `message` is parameterized by a value of type `integer` and we use existential and refined type expressions to pass the value of the field `length` as actual parameter of the `first` pointer to a `message(mlen)` type. This specification is given in Figure 6.

```
struct message(len) {
   struct message(len)+ next;
   char[len]+ buffer;
};

∃ mlen:integer with self > 0.
struct message_box {
  integer with self = mlen length;
  struct message(mlen)+ first;
};

void zeros_buffer(struct message_box+ box);
```

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

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

void zeros_buffer(struct message_box* box);
```

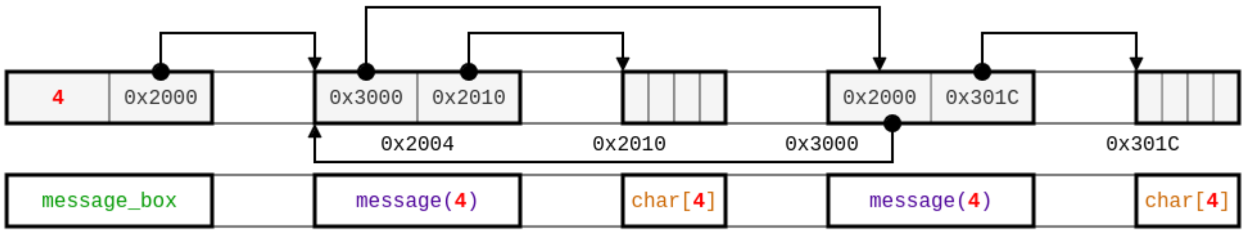


Figure 6: Parameterized type to link properties of various memory regions

With this specification, our analyzer does not report memory vulnerabilities, allowing us to prove that the code of the actual OS is spatially memory safe for the memory layout given by the specification.

4.6 Union types

To illustrate union type expression, we take an example extracted from the `Olden` benchmark, used by the tool *Checked-C* of Microsoft to illustrate its features for dynamic verification. The code defines a binary tree type `node`, a function creating a complete binary tree and a function traversing the tree to print it. By applying CPROTO, we obtain the following specification file:

```
struct node {
  int value;
  struct node *left;
  struct node *right;
};
struct node* RandTree(int n);
void PrintTree (int n, struct node *node);
int  main(void);
```

Using union type expressions, we could express that a binary tree is full, i.e., each node had two or zero children. The union type collects the two types of nodes: an interior node has two non null pointers two its children, a leaf node has two null values as children.

```
type node = struct {
  byte[4] value;
  union {
    struct {      /* interior node */
      node+ left;
      node+ right;
    } interior;

    struct {      /* leaf node */
      byte[4] with self = 0 left;
      byte[4] with self = 0 right;
    } leaf;
  } succ;
};
```

Union types combined with parameterized types are useful to express the stronger invariant of a perfect binary tree. For this, we introduce a parameter for the type `node` to denote the height `h` of the node. If `h` is greater than 1 then the node is interior so it has both children not null and at an height decreased by one; otherwise the node is a leaf and has both children null. The type for a non null pointer to a complete binary tree of any height is `nodeptr` type, whose definition employs existential types to obtain an unbounded union type.

```
type node(h:byte[4]) = struct  {
  byte[4] value;
  union {

    (struct {     /* interior node */
      node(h-1)+ left;
      node(h-1)+ right;
    } with (h > 1)) interior;

    (struct {     /* leaf node */
      byte[4] with self = 0 left;
      byte[4] with self = 0 right;
    } with (h = 1)) leaf;
  } succ;
};

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

4.7 Specifying functions

Our specification language follows the C language syntax for function declaration. To this, we add several advanced features:

- Annotation `inline` forces the analyzer to inline the function code rather than applying an inter-procedural analysis; if the function is recursive, the analysis will fail.
- Annotation `pure` signifies that the function does only “in frame” operations on memory, like reading, writing on addresses at local variables or writing in a recently allocated region. In other words, it indicates that the abstract memory is unchanged. This allows the analyzer to be more precise in inter-procedural analysis of function calls.
- Existential types may be used to express invariants between function’s arguments and result. For instance, the following specification states that `RandTree` returns a complete binary tree of height `n`.

```
∃ h:int with self > 0. (node(h)+ RandTree( (int with self = h) n));
```

- Functions may receive as arguments pointers to functions:

```
list+ map_list_int(list + 1, ([int] -> int)+ f);
```

5 Conclusion

For more details on the specification language and the results of the analysis, please consult [1]. For explanation of the technique used by the analyser, the reference [2] is a a good start.

References

[1] CODEX site, 2024. https://codex.top.

[2] Anon. A dependent nominal physical type system for static analysis of low level code. Technical report, 2024. Extended version of this paper including the appendix.

[3] Olivier Nicole. Automated Verification of Systems Code using Type-Based Memory Abstractions. PhD thesis, University Paris-Saclay, 2022. https://theses.hal.science/tel-03962643v2.

A Concrete Syntax for the Specification

We provide here the concrete syntax to be used to specify types in the input file of our analysis.

A.1 Type definition and function profile syntax

$$
\begin{array}{rcll}
\rho & ::= & n & \text{(type name)} \\
 & \mid & n(\alpha_1, \ldots, \alpha_\ell) & \text{(type constructor)} \\
 & & & \\
\texttt{spec} & ::= & \texttt{def}{*} & \text{(specification definitions)} \\
 & & & \\
\texttt{def} & ::= & \texttt{type\_def} & \text{(type definition)} \\
 & \mid & [\texttt{inline}] \quad [\texttt{pure}] \quad \texttt{fun\_def} & \text{(function profile definition)} \\
 & & & \\
\texttt{type\_def} & ::= & \texttt{type}\ \rho = \tau & \text{(type definition)} \\
 & \mid & \texttt{c\_type}; & \text{(C type definition)} \\
 & & & \\
\texttt{c\_type} & ::= & \texttt{struct}\ \rho\ \{\tau_1\ f_1; \cdots ; \tau_n\ f_n;\} & \text{(C struct type)} \\
 & \mid & \texttt{union}\ \rho\ \{\tau_1\ f_1; \cdots ; \tau_n\ f_n;\} & \text{(C union type)} \\
 & \mid & \texttt{ex}\ \alpha : \tau.\ \texttt{c\_type} & \text{(existential C type definition)} \\
 & & & \\
\texttt{fun\_def} & ::= & \tau \quad n\,(\tau_1\ a_1, \cdots , \tau_\ell\ a_\ell) & \text{(function profile)} \\
 & \mid & \texttt{ex}\ \alpha : \tau\,.\,\texttt{fun} & \text{(existential function profile)} \\
 & & & \\
\texttt{ex} & ::= & \exists \mid \texttt{\textbackslash exists} &
\end{array}
$$

A.2 Type expressions

$$\begin{array}{rcll}
\eta & ::= & \texttt{byte} & \text{(byte name)} \\
& | & n(e_1, ..., e_\ell) & \text{(parameterized type)} \\
& | & \texttt{struct}\ n & \text{(struct with name)} \\
& | & \texttt{struct}\ n(e_1, ..., e_\ell) & \text{(struct with name)} \\
\\
\Lambda & ::= & \texttt{[}\tau_1, \cdots, \tau_\ell\texttt{] ->}\ \tau & \text{(function type)} \\
& | & \texttt{ex}\ \alpha : \tau \,.\, \Lambda & \text{(existential function type)} \\
\\
\tau & ::= & \eta & \text{(named types)} \\
& | & \eta\texttt{*} & \text{(possibly null pointer type)} \\
& | & \eta\texttt{+} & \text{(non null pointer type)} \\
& | & \texttt{struct}\ \{\tau_1\ f_1; \cdots; \tau_\ell\ f_\ell; \} & \text{(struct type)} \\
& | & \tau\ \texttt{with}\ p & \text{(refinement type)} \\
& | & \tau\,\texttt{[}e\texttt{]} & \text{(array type)} \\
& | & \texttt{ex}\ \alpha : \tau_1 \,.\, \tau_2 & \text{(existential type)} \\
& | & \texttt{union}\ \{\tau_1\ f_1; \cdots; \tau_\ell\ f_\ell; \} & \text{(union type)} \\
& | & \Lambda\texttt{+} & \text{(function pointer type)} \\
\\
p & ::= & e_1 \bowtie e_2 & \text{(comparison with } \bowtie \in \{<, <=, ==, !=, >, >=\}\text{)} \\
& | & p_1\ \&\&\ p_2 & \text{(conjunction)} \\
& | & p_1 \mid\mid p_2 & \text{(disjunction)} \\
\\
e & ::= & \texttt{self} & \text{(self expression)} \\
& | & c & \text{(constant with } c \in \mathbb{Z}\text{)} \\
& | & \alpha & \text{(symbolic variable)} \\
& | & e_1 \diamond e_2 & \text{(binary operation with } \diamond \in \{+, -, *, /, \%, |, \&\}\text{)}
\end{array}$$

Symbol ∃ corresponds to U+2203 in unicode. The brackets in mathematical style correspond to optional parts of the syntax, while the ones in typewriter font correspond to ponctuation in the concrete syntax.