Binsec/Codex, an abstract interpreter to verify safety and security properties of systems code

Olivier Nicole

April, 2021

Abstract

This document describes the internals of BINSEC/CODEX, an analyzer able to verify safety and security properties on machine code, notably used to verify absence of runtime errors and privilege escalation in embedded kernels. After stating our assumptions on the hardware, we give a detailed overview of the abstract domains used in the analysis, and give examples of why each domain is needed.

Contents

1 Introduction

This document describes the internals of Binsec/Codex, an analyzer for machine code based on abstract interpretation. Binsec/Codex is a module of Binsec [2]. Binsec/Codex was used to analyze and verify absence of runtime errors and privilege escalation on embedded kernels [10].

Abstract interpretation is a practical framework to build program analyzers, as well as a theoretical framework justifying the correctness of such analyzers [1]. An abstract interpreter infers properties of a pogram in a sound way, i.e. the inferred properties will hold on all possible behaviours of the program. By Rice's theorem [11], such a sound inference cannot be complete, i.e. it may include behaviours that will never appear in the actual program execution.

In the context of Binsec/Codex, we will concentrate on state properties. A state property is a predicate on a program state. What exactly constitutes a program state is defined in Section 2, but it can be seen as the state of all variable and memory locations of the program, as well as the program counter (the current code location). The absence of runtime error, for instance, is a state property. The absence of use-after-free bugs in C is not.

Inferring state properties using abstract interpretation consists in propagating an abstract state that represents a superset of all possible states, until a fixpoint is reached. It can be seen as a generalization of data flow analysis [6] with more general domains and widening operators to handle loops.

We will use the code in Figure 1 as a running example. It is an example of memory protection initialization in a simple x86 32-bit embedded OS. Each task is allocated a fixed region of memory describe by the fields `mem_base` and `mem_size` of a structure `context`. A global variable `params` contains the array of all contexts, as well as the number of tasks (assumed constant throughout kernel execution).

Memory protection in 32-bit x86 depends on a global descriptor table (`desc_table`) containing segment descriptors. Each segment descriptor encodes a range of memory addresses, along with the permissions associated with these addresses. Among other things, a segment descriptor contains a base address and a length. What the code in Figure 1 does is creating a descriptor for each task and writing it to the descriptor table. The code of the `create_descriptor` function is omitted for now.

The goal is to verify that all segment descriptor written to the descriptor table describe a memory region that is disjoint from the kernel space.

```c
typedef struct context {
  uint32_t mem_base;
  uint32_t mem_size;
  // ...
} context;

struct params {
  unsigned int nb_tasks;
  context *contexts;
  uint64_t *desc_table;
  // ...
};

struct params *params;
unsigned int i;
context *ctx;

for(i = 0; i < params->nb_tasks; i++)
{
  ctx = &params->contexts[i];
  params->desc_table[i] = create_descriptor(ctx->mem_base,
                                            ctx->mem_size);
}
```

Figure 1: Running example

If that is achieved, and that you also verify that there is no run-time error, then it entails that there cannot be a privilege escalation exploit on this kernel —except via means not considered in our hardware model, such as side channels. This verification must be performed on the binary code resulting of the compilation of the C code (see Figure 1 below), firstly because it removes the compiler from the trusted components, and secondly because such system code necessarily comprises hand-written, architecture-specific assembly (e.g. to modify memory protection). In what follows, we will see which challenges such a verification poses.

After expliciting our hardware model (Section 2), we will detail the set of abstract domains we use for the analysis (Section 3), including a novel memory domain.

2 Concrete semantics of machine code

By concrete semantics, we mean the way we model the architecture, and the semantics of analyzed programs on this architecture. A semantics consists in two things: defining what is a program state, and specifying how to transition from one state to the next.

We assume a 32-bit architecture. Values are elements of $\mathrm{V}_{32} = [0, 2^{32} - 1]$ or $\mathrm{V}_8 = [0, 2^8 - 1]$. The set of memory addresses A is a subset of V_{32}. Memories are maps from addresses to 8-bit values: $\mathrm{M} = \mathrm{A} \to \mathrm{V}_8$. We denote by R the set of register names. In 32-bit x86, it would contain for example `eax` and `ebx`, but also `CPL` (current privilege level), which is a system register only accessible through some system instructions.

The set of states is $\mathcal{S} = \mathrm{M} \times (\mathrm{R} \to \mathrm{V}_{32})$. A state includes the current code location, via the `eip` register. For $s \in \mathcal{S}$, we denote by $s[r \leftarrow x]$ the state identical to s, except that register r holds the value x; and $s\left[[\alpha]_n \leftarrow x\right]$, the state identical to s, except that the address range $\alpha, \alpha + 1, \ldots, \alpha + n - 1$ holds the value x (in the present case, in little-endian convention).

Figure 2 shows the binary and the assembly produced by `gcc -O3` on the running example. All control flow and type information is stripped away. The **`for`** loop is replaced by a simple conditional backward jump at address `0x90`.

DBA Binsec translates machine code into an intermediate representation called DBA (for Dynamic Bitvector Automata), a simple imperative language with variables, memory accesses and arbitrary jumps. The syntax of DBA is given in Figure 3. The semantics of DBA is completely standard,

```
40: a1 1c c0 04 08          mov    eax,ds:0x804c01c
45: 8b 10                   mov    edx,DWORD PTR [eax]
47: 85 d2                   test   edx,edx
49: 74 54                   je     9f <main+0x5f>
4b: 8d 4c 24 04             lea    ecx,[esp+0x4]
4f: 83 e4 f0                and    esp,0xfffffff0
52: ff 71 fc                push   DWORD PTR [ecx-0x4]
55: 55                      push   ebp
56: 89 e5                   mov    ebp,esp
58: 56                      push   esi
59: 53                      push   ebx
5a: 31 db                   xor    ebx,ebx
5c: 51                      push   ecx
5d: 83 ec 0c                sub    esp,0xc
60: 8b 50 04                mov    edx,DWORD PTR [eax+0x4]
63: 8d 0c dd 00 00 00 00    lea    ecx,[ebx*8+0x0]
6a: 8b 70 08                mov    esi,DWORD PTR [eax+0x8]
6d: 83 ec 08                sub    esp,0x8
70: 83 c3 01                add    ebx,0x1
73: 01 ca                   add    edx,ecx
75: 01 ce                   add    esi,ecx
77: ff 72 04                push   DWORD PTR [edx+0x4]
7a: ff 32                   push   DWORD PTR [edx]
7c: e8 3f 01 00 00          call   80491c0 <create_descriptor>
81: 83 c4 10                add    esp,0x10
84: 89 06                   mov    DWORD PTR [esi],eax
86: a1 1c c0 04 08          mov    eax,ds:0x804c01c
8b: 89 56 04                mov    DWORD PTR [esi+0x4],edx
8e: 39 18                   cmp    DWORD PTR [eax],ebx
90: 77 ce                   ja     60 <main+0x20>
92: 8d 65 f4                lea    esp,[ebp-0xc]
95: 31 c0                   xor    eax,eax
97: 59                      pop    ecx
98: 5b                      pop    ebx
99: 5e                      pop    esi
9a: 5d                      pop    ebp
9b: 8d 61 fc                lea    esp,[ecx-0x4]
9e: c3                      ret
9f: 31 c0                   xor    eax,eax
a1: c3                      ret
```

Figure 2: Running example compiled by GCC with -O3.

Expressions

$$
\begin{array}{rlll}
\mathscr{E} \ni e, e_{\text{loc}} & ::= & c & \text{constant } (c \in \mathbb{N}) \\
& | & r & \text{register } (r \in \mathrm{R}) \\
& | & [e_{\text{loc}}] & \text{memory read} \\
& | & \ominus e & \text{unary operation } (\ominus \in \{-, \neg\}) \\
& | & e \oplus e & \text{binary operation } (\oplus \in \{+, -, \div, ==, \leq, \cdots\})
\end{array}
$$

Instructions

$$
\begin{array}{rlll}
\mathscr{I} \ni i & ::= & r := e & \text{register assignment } (r \in \mathrm{R}) \\
& | & [e_{\text{loc}}] := e & \text{memory write} \\
& | & \texttt{jump}\ e_{\text{loc}} & \text{jump to address} \\
& | & \texttt{if}\ e\ \texttt{then}\ i\ \texttt{else}\ i\ \texttt{end} & \text{conditional}
\end{array}
$$

Figure 3: Syntax of DBA.

so we will not detail it.

Multicore systems In a multicore system, the state is an element of $\mathrm{M} \times (\mathrm{R} \to \mathrm{V}_{32})^n$, where n is the number of cores. Each core has a distinct set of registers —except for system registers which are common, but for simplicity's sake we will not consider them here. The memory is shared. Each CPU performs an independent execution governed by the same semantics.

3 Abstract domains used in the analysis

Abstract interpretation revolves around the concept of abstract domains. An abstract domain is a set, the elements of which abstract elements. Each abstract element represents a set of concrete elements (e.g. a set of program states) through a concretization function, generally denoted γ. Following tradition, we will write abstract domains with a superscript sharp symbol ($\sharp$). For instance, if the abstract domain $\mathbb{D}^\sharp$ represents sets of program states, its concretization has type:

$$\gamma_{\mathrm{D}} : \mathbb{D}^\sharp \to \mathscr{P}(\mathscr{S})$$

To be usable in static analysis, an abstract domain must be equipped with a partial order, representing precision. An abstract element is smaller than

```
40 :  eax := [ds + 0x804c01c]
      jump 0x45
45 :  edx := [ds + eax]
      jump 0x47
47 :  res32 := edx − edx
      ZF := (res32 == 0)
      CF := 0
      OF := 0
      jump 0x49
49 :  if ZF then jump 0x9f else jump 0x4b end
```

Figure 4: First few instructions from Figure 2 translated to DBA.

numeric domain	$\mathbb{N}^\sharp$ = any conjunction of numeric constraints over the values bound to $\mathbb{A}_F$ and $\mathscr{R}$	$\gamma_N : \mathbb{N}^\sharp \to \mathscr{P}(R \uplus A_F \to V_{32})$
weak shape domain	$\mathbb{W}^\sharp = \mathbb{N}^\sharp \times \left(R \uplus A_F \to \mathbb{T}^\sharp\right)$	$\gamma_W : \mathbb{W}^\sharp \to \mathscr{P}(\mathscr{S})$
control flow domain	$\mathbb{C}^\sharp = \mathscr{P}(\mathscr{G}) \times \left(\mathbb{L} \to \mathbb{W}^\sharp\right)$	$\gamma_C : \mathbb{C}^\sharp \to \mathscr{P}(\mathscr{S})$

$\mathscr{G} = \mathscr{P}(\mathbb{L} \times \mathbb{L})$ is the set of control flow graphs,
$\mathbb{T}^\sharp$ is the single type abstract domain,
A_F the set of fixed addresses in the kernel,
R the set of register names, $\mathbb{L}$ the set of program locations.

Figure 5: Implementation of the abstract domains.

another ($a \leq^\sharp b$) only if the concrete set that a represents is a subset of the one b represents:

$$\forall a, b \in \mathbb{D}^\sharp,\ a \leq^\sharp b \implies \gamma_D(a) \subseteq \gamma_D(b)$$

In this section, the abstract domains that we use will be detailed. They are summarized in Figure 5.

3.1 Control flow domain

Central to our abstractions is the notion of program locations. What is a program location is an implementation choice of the analyzer: a natural choice is to consider that a program locations is a valid code address in the program's executable. For our needs, we chose a more precise abstraction: a program location consists in a kernel address and a call stack. Denoting the set of valid code addresses by $A_C \subseteq A$ and $E^\star$ the set of finites sequences of elements of a set E, we get:

$$\mathbb{L} = A_C \times A_C^\star$$

We denote by $\mathscr{L}(s)$ the program location of a state s.

Then, our control flow abstraction is a graph between program locations (we denote $\mathscr{G} = \mathscr{P}(\mathbb{L} \times \mathbb{L})$ the set of such graphs) and a mapping from every location to an abstract state $\mathbb{M}^\sharp$ (described further):

$$\mathbb{C}^\sharp = \mathscr{P}(\mathscr{G}) \times \left(\mathbb{L} \to \mathbb{M}^\sharp\right)$$

The graph is an over-approximated control flow graph (CFG), meaning that (1) the only reachable program locations are the nodes in the graph, and (2) all possible control flow transfer are represented by an edge in the graph. Formally:

$$\gamma : \mathbb{C}^\sharp \to \mathscr{P}(\mathcal{S})$$
$$\gamma(\mathscr{G}, \text{states}) = \{s \in \mathcal{S} \mid \exists \ell \in \mathbb{L}, \text{states}(\ell) = s \wedge \ell \text{ is a node of } \mathscr{G}\}$$

The analysis works by performing multiple rounds of the following steps in sequence:

1. Perform a standard data-flow analysis using the current abstract CFG, to compute a new state for every location of the graph.
2. Iterate over all locations ℓ to compute all possible outgoing edges, given the possible states at ℓ (this uses the same `resolve` function than [7]). Newly-discovered edges are added to the over-approximated CFG.

The iteration sequence starts with an abstraction $(\mathscr{G}_0, m_0)$ where $\mathscr{G}_0$ is a single node (the start instruction address), and m_0 maps this address to the initial abstract state. The analysis terminates when the fixpoint is reached, i.e., no new edge is discovered in the CFG. In practice, several small optimisations are used to reuse results between rounds (e.g., caching the results), and to have fewer rounds (by early exploration of the newly-discovered CFG nodes).

Theorem 1. *If the transfer functions for $\mathbb{M}^\sharp$ are sound, the result $s^\sharp_{final}$ of the analysis is a sound abstraction of all the reachable states in the system (and thus a state invariant).*

In the running example, the analysis proceeds as follows:

1. While instruction `0x90` is not reached, every instruction has the next instruction as its only successor, so the graph is linear.
2. After analyzing instruction `0x90` for the first time, the conditional is false (unless `params->nb_tasks` is equal to 1, but during the analysis we do not have this information), so there is only one successor, `0x60`. This constitutes a back edge in the CFG (i.e. an edge to an ancestor in a spanning tree of the CFG). This characterizes a control flow loop, and therefore the control flow domain chooses a loop head and inserts a widening point.
3. After a fixpoint is reached for every location in the loop, there should be a new successor to `0x90` (since the loop is not an infinite one), namely `0x92`, from which new paths can be discovered.

It should be noted that the choice of including the call stack in the abstract locations has the consequence that all function calls are inlined. On the one hand, it allows for a precise interprocedural analysis, since a function is always analyzed in the most precise possible calling context. On the other hand, it prevents the analyzer to handle some recursive functions (except when all recursive calls are tail calls and are optimized into jumps), because the analyzer may compute abstract states with ever-growing call stacks, without reaching a stable invariant. Since low-level code rarely contains recursive code, this is usually not a problem.

3.2 Numeric domain

What we call numeric domain, in this context, is an abstraction of both numeric values and memory. We call this domain $\mathbb{N}^\sharp$ and it represents the contents of all memory cells which we want to fully enumerate, and registers. The designation "all memory cells which we want to fully enumerate" matters because part of memory will be represented only by type information in the weak shape domain, and not explicitly represented at the byte level.

The concretization of the numeric domain thus has type:

$$\gamma_N : \mathbb{N}^\sharp \to \mathscr{P}(R \uplus A_F \to V_{32})$$

where $A_F \subseteq A$. (The F stands for "flat memory model".)

In principle, any domain that has such a concretization could be used. In practice, of course, it must be sufficiently precise to prove the properties we want. We use a combination of standard abstractions, the full description of which is not the intent of this report.

- For values, we mainly use efficient non-relational abstract domains: intervals, based on the reduced product between the signed and unsigned meaning of bitvectors [3], with congruence information [5]. They are complemented with symbolic relational information [3, 9, 4] for local simplifications of sequences of machine code.
- Regarding the memory abstraction: our memory model is ultimately byte-level in order to deal with very low-level coding aspects of kernels. Yet, as representing each memory byte separately is inefficient and imprecise, we use a stratified representation of memory caching multi-byte loads and stores, like Miné [8]. Moreover, we do not track memory addresses whose contents is unknown.

3.3 Weak shape domain

3.3.1 Notations

We define a bitvector concatenation operation. It corresponds to the idea of joining two binary representations next to each other and interpreting them as a single integer. It depends the bit length of each operand.

$$\forall v_1, v_2, s_1, s_2 \in \mathbb{N},\ v_1 \,{}_{s_1}{::}_{s_2}\, v_2 = v_1 + 2^{s_1} v_2$$

Intuitively, s_1 and s_2 are the lengths of the two bit vectors v_1 and v_2. We may omit the lengths when they are clear from the context. We note $\mathbb{H} = \mathrm{A} \to \mathrm{V}_8$ the set of heaps. For $h \in \mathbb{H}$, we will write indifferently $h(a)$ or $h[a]$. We will write $h[a, a+n]$ to mean $h[a] :: h[a+1] :: \cdots :: h[a+n]$.

3.3.2 Types and their meaning

We want to abstract properties of memory structures using types. To that end, we are going to define a type system encompassing the low-level types of the C language: scalars, pointers, and structures (which, following the conventions of type system research, we will call product types). We do not have so-called sum types, and thus cannot express C unions; however, nothing in principle prevents sum types to be added. One of the nice features of this system is that it is easy to extend it by enriching the grammar of types, and make the necessary adjustments to the definitions and proofs. By enriching the type system, one directly enriches the abstract domain that derives from it.

Speaking of enriching the types, we also define a form of refinement types, i.e. types augmented with predicates. Such types do not exist in

C, but by using them we can specify —and verify— useful properties on values. The predicates can by any unary predicates from a given logic. In our analyzer, predicates are from a simple, first-order logic containing equality and inequality operators, as well as usual operators from integer arithmetic (addition, multiplication, ...) and bitwise operators (and, xor, ...).

The grammar of types is:

$$
\begin{array}{rcll}
\mathbb{T} \ni t & ::= & \texttt{byte} \mid \texttt{word} & \text{base types} \\
& \mid & t_a* & \text{pointer} \\
& \mid & t \times t & \text{product type} \\
& \mid & \{x : t \mid p(x)\} & \text{refinement type with a predicate } p \\
& \mid & t[s] & \text{array type } (s \in \mathbb{N}) \\
\\
\mathbb{T}_{\mathrm{A}} \ni t_a & ::= & t.o & \text{cell type } (o \in \mathbb{Z})
\end{array}
$$

Each type has a size (in bytes) given by the function:

$$
\begin{aligned}
\mathrm{size} &: \mathbb{T} \to \mathbb{N} \\
\mathrm{size}\,(\texttt{byte}) &= 1 \\
\mathrm{size}\,(\texttt{word}) &= 4 \\
\mathrm{size}\,(t*) &= 4 \\
\mathrm{size}\,(t_1 \times t_2) &= \mathrm{size}\,(t_1) + \mathrm{size}\,(t_2) \\
\mathrm{size}\,(\{x : t \mid p(x)\}) &= \mathrm{size}\,(t) \\
\mathrm{size}\,(t[s]) &= s \cdot \mathrm{size}\,(t)
\end{aligned}
$$

The size of `word` being equal to 4 comes from the fact that the architecture is 32-bit. We will give the meaning of types in terms of sets of values shortly, but for that we need to introduce labellings.

Definition 2 (Labelling). *A labelling $\mathscr{L}$ is a function of $\mathrm{A} \to \mathbb{T}_{\mathrm{A}}$ such that all instances of a type are whole and contiguous in memory, i.e. for all type $t \in \mathbb{T}$ and address $a \in \mathrm{A}$, if we define $n = \mathrm{size}\,(t)$:*

$$
(\exists k \in [0, n-1]\,,\ \mathscr{L}(a+k) = t.k) \implies \begin{cases} \mathscr{L}(a) = t.0 \\ \mathscr{L}(a+1) = t.1 \\ \vdots \\ \mathscr{L}(a+n-1) = t.(n-1) \end{cases}
$$

The set of labellings is denoted Lab.

Two address types can express similar things, with one being more precise than the other. Consider a C structure like the following:

```
struct s {
  uint8_t a;
  uint32_t *b;
};
```

In the language of our types, this would be expressed: $s = \texttt{byte} \times \texttt{word}.0*$. Now consider the type of the first memory cell in such a structure: the type $s.0$. In some sense, an $s.0$ is also a $\texttt{byte}.0$. However, $s.0$ is more precise than $\texttt{byte}.0$, in the sense that not all memory cells that contain a `byte` are necessarily part of a structure s. We say that $s.0$ subsumes $\texttt{byte}.0$.

Definition 3 (Subsumption relation)**.** *A cell type (an element of* $\mathbb{T}_A$*) can* subsume *another cell type in the following conditions:*

$$(t_1 \times t_2).k \textit{ subsumes } t_1.k \textit{ if } 0 \le k < \operatorname{size}(t_1)$$

$$(t_1 \times t_2).(\operatorname{size}(t_1) + k) \textit{ subsumes } t_2.k \textit{ if } 0 \le k < \operatorname{size}(t_2)\,.$$

$$\{x : t \mid p(x)\}.k \textit{ subsumes } t.k$$

$$t[s].k \textit{ subsumes } t.o, \textit{ if} \begin{cases} k = q \cdot \operatorname{size}(t) + o \\ 0 \le o < \operatorname{size}(t) \end{cases}$$

We define the relation $\preceq \in \mathbb{T}_A \times \mathbb{T}_A$ *as the transitive reflexive closure of the relation "subsumes".*

Note that defining $\preceq$ as reflexive requires a notion of equality on $\mathbb{T}_A$. We use for that the equality that can be defined inductively on the grammar in a straightforward way.

Lemma 4. *For all types* $u, a, b \in \mathbb{T}_A$*, if* u *subsumes* a *and* u *subsumes* b*, then* $a = b$*.*

Proof. By the definition of "subsumes", necessarily u is one of the following:

- $u = \{x : t \mid p(x)\}.k$. Then necessarily $a = b = t.k$.
- $u = t[s].k$. Then it is easy to show that $a = b$.
- $u = (t_1 \times t_2).k$. Also from the definition of subsumption, a is either the type $t_1.k$ or the type $t_2.(k - \operatorname{size}(t_1))$. Same for b. Let us proceed *ab absurdum* and assume $a \neq b$. Then, there are two symmetrical possibilities. Let us devise the case where $a = t_1.k$ and $b = t_2.(k - \operatorname{size}(t_1))$.

  Again from Definition 3, we deduce:

$$\begin{cases} 0 \le k < \operatorname{size}(t_1) \\ 0 \le k - \operatorname{size}(t_1) < \operatorname{size}(t_2) \end{cases}$$

from which we derive two contradictory statements: $k < \mathrm{size}(t_1)$ and $k \geq \mathrm{size}(t_1)$. Therefore $a = b$.

□

We now give the meaning of types in terms of an interpretation function.

Definition 5 (Interpretation of a type)**.** *The interpretation operator with respect to a labelling $\mathscr{L}$, denoted $(\!| \cdot |\!)_{\mathscr{L}} : \mathbb{T} \to \mathbb{N}$, is defined by:*

$$
\begin{aligned}
(\!| \texttt{byte} |\!)_{\mathscr{L}} &= [0, 2^8 - 1] \\
(\!| \texttt{word} |\!)_{\mathscr{L}} &= [0, 2^{32} - 1] \\
(\!| t_1 \times t_2 |\!)_{\mathscr{L}} &= \left\{ v_1 \,{}_{\mathrm{size}(t_1)}\!:\,:\!{}_{\mathrm{size}(t_2)}\, v_2 \;\middle|\; v_1 \in (\!| t_1 |\!)_{\mathscr{L}}, v_2 \in (\!| t_2 |\!)_{\mathscr{L}} \right\} \\
(\!| \{x : t \mid p(x)\} |\!)_{\mathscr{L}} &= \left\{ v \in (\!| t |\!)_{\mathscr{L}} \mid p(v) \right\} \\
(\!| t[s] |\!)_{\mathscr{L}} &= \{ v_1 :: \cdots :: v_s \mid v_1, \ldots, v_s \in (\!| t |\!)_{\mathscr{L}} \} \\
(\!| t.k* |\!)_{\mathscr{L}} &= \{ a \in \mathrm{A} \,;\, \mathscr{L}(a) \preceq t.k \} \cup \{0\}
\end{aligned}
$$

A labelling is a labelling *for* a heap h when values are laid out in memory in a manner consistent with their types.

Definition 6. *We say that a labelling $\mathscr{L} \in \mathrm{Lab}$ is a labelling for $h \in \mathbb{H}$ if:*

$$\forall a \in \mathrm{A},\ \mathscr{L}(a) = t.0 \implies h[a, a + \mathrm{size}(t) - 1] \in (\!| t |\!)_{\mathscr{L}}$$

Note that by Definition 5, we have

$$t.n \preceq u.m \implies (\!| t.n* |\!)_{\mathscr{L}} \subseteq (\!| u.m* |\!)_{\mathscr{L}}$$

So $\preceq$ can be seen as a subtyping relation on pointers.

Definition 7 (Set of addresses of a given type)**.**

$$
\begin{aligned}
\mathrm{addr} &: \mathrm{Lab} \times \mathbb{T} \to \mathscr{P}(\mathrm{A}) \\
\mathrm{addr}_{\mathscr{L}}(t) &= \bigcup_{i=0}^{\mathrm{size}(t)-1} \{ a \in \mathrm{A}; \mathscr{L}(a) \preceq t.i \}
\end{aligned}
$$

In any reasonable program of any language, values of different types should reside in different zones of memory. More precisely, *incompatible* types should reside in different zones of memory. What makes one type "compatible" with another is a subsumption relation between them. If some struct contains an `uint32` at offset 4, then it is expected that, wherever that struct exists in memory, the object existing at bytes 4 to 8 of the struct should be interpretable as an `uint32`. Therefore, we define a labelling as separated when only the types that are in a subsumption relation can be attributed to the same cells.

Definition 8 (Separated labellings)**.** *A labelling* $\mathscr{L} \in \mathrm{A} \to \mathscr{P}(\mathbb{T}_\mathrm{A})$ *is separated if incomparable cell types cannot be labels of the same cell:*

$$\mathrm{addr}_{\mathscr{L}}(t) \cap \mathrm{addr}_{\mathscr{L}}(u) \neq \emptyset \implies \exists n, m,\ t.n \preceq u.m \vee u.m \preceq t.n$$

We will prove that this desirable property holds in fact for all labellings. But first, a few properties of $\preceq$ need to be shown.

Proposition 9. $(\mathbb{T}_\mathrm{A}, \preceq)$ *is a partial order.*

Proof. $\preceq$ is reflexive and transitive by definition; it is also antisymmetric, because the elements of $\mathbb{T}_\mathrm{A}$, by their definition, cannot contain themselves. □

Lemma 10. *For any cell types* $\tau, \upsilon, \phi \in \mathbb{T}_\mathrm{A}$*:*

$$\phi \preceq \tau \wedge \phi \preceq \upsilon \implies \tau \preceq \upsilon \vee \upsilon \preceq \tau$$

Proof. Let us assume that $\phi \preceq \tau$ and $\phi \preceq \upsilon$. Then, if we write $x \xrightarrow{s} y$ for "x subsumes y", there exists two finite chains $\phi \xrightarrow{s} \tau_1 \xrightarrow{s} \tau_2 \xrightarrow{s} \cdots \xrightarrow{s} \tau_n \xrightarrow{s} \tau$, and $\phi \xrightarrow{s} \upsilon_1 \xrightarrow{s} \upsilon_2 \xrightarrow{s} \cdots \xrightarrow{s} \upsilon_m \xrightarrow{s} \upsilon$. Either one chain is included in the other, or not. If yes, then either $\tau \preceq \upsilon$ or $\upsilon \preceq \tau$, which ends the proof. If not, then let p be the minimal index such that $\tau_p \neq \upsilon_p$. But τ_{p-1} subsumes both τ_p and υ_p, so using Lemma 4, $\tau_{p-1} = \upsilon_{p-1}$, which contradicts the minimality of p. □

In other words, the graph of the order relation $(\mathbb{T}_\mathrm{A}, \preceq)$ is a forest.

Theorem 11. *All labellings are separated.*

Proof. Let us assume that the intersection of $\mathrm{addr}_{\mathscr{L}}(t)$ and $\mathrm{addr}_{\mathscr{L}}(u)$ contains some address a. Then there exists i and j such that $\mathscr{L}(a) \preceq t.i$ and $\mathscr{L}(a) \preceq u.j$. Thus by Lemma 10, either $t.i \preceq u.j$, or $u.j \preceq t.i$. □

3.3.3 The single type abstract domain

Here we show how $\mathbb{T}$, supplemented with "top" and "bottom" elements, can be seen as an abstraction for a set of values. Its concretization is only defined relatively to a labelling $\mathscr{L}$.

Definition 12. *Given a labelling* $\mathscr{L} \in \mathrm{Lab}$*, the single type abstract domain is:*

$$\mathbb{T}^\sharp = \mathbb{T} \uplus \{\bot, \top\}$$

Its concretization is:

$$\begin{aligned}
&\gamma_{\mathscr{L},\mathbb{T}} : \mathbb{T}^\sharp \to \mathscr{P}(\mathbb{N})\\
&\gamma_{\mathscr{L},\mathbb{T}}(\bot) = \emptyset\\
&\gamma_{\mathscr{L},\mathbb{T}}(t) = (\!| t |\!)_{\mathscr{L}}\\
&\gamma_{\mathscr{L},\mathbb{T}}(\top) = \mathbb{N}
\end{aligned}$$

Its abstract inclusion $\sqsubseteq_{\mathscr{L},\mathbb{T}}$ is defined by:

- $\bot \sqsubseteq_{\mathscr{L},\mathbb{T}} t$
- $t \sqsubseteq_{\mathscr{L},\mathbb{T}} \top$
- $t.n \preceq u.n \implies t.n* \sqsubseteq_{\mathscr{L},\mathbb{T}} u.m*$

Its lub $\sqcup_{\mathscr{L},\mathbb{T}}$ is the one induced by $\sqsubseteq_{\mathscr{L},\mathbb{T}}$.

Proposition 13. *$\sqsubseteq_{\mathscr{L},\mathbb{T}}$ and $\sqcup_{\mathscr{L},\mathbb{T}}$ are sound with respect to $\gamma_{\mathbb{T}}$.*

3.3.4 The weak shape abstract domain

The weak shape abstract domain $\mathbb{W}^\sharp$ is a memory domain which abstracts part of the memory using types. The weak shape domain is essentially an augmentation upon another domain $\mathbb{M}^\sharp$, which concretizes to $\mathscr{P}(\mathrm{R} \uplus \mathrm{A}_\mathrm{F} \to \mathrm{V}_{32})$, where $\mathrm{A}_\mathrm{F} \subseteq \mathrm{A}$. (In the context of our analyzer, we instantiate $\mathbb{M}^\sharp$ to the domain $\mathbb{N}^\sharp$ described above.)

$$\mathbb{W}^\sharp = \mathbb{M}^\sharp \times \left(\mathrm{R} \uplus \mathrm{A}_\mathrm{F} \to \mathbb{T}^\sharp\right)$$

Its concretization has type:

$$\gamma_\mathrm{W} : \mathbb{W}^\sharp \to \mathscr{P}(\mathrm{R} \uplus \mathrm{A}_\mathrm{F} \uplus \mathrm{A}_\mathrm{T} \to \mathrm{V}_{32})$$

where A_F and A_T are disjoint. If one assumes that $\mathrm{A}_\mathrm{F} \uplus \mathrm{A}_\mathrm{T} = \mathrm{A}$, then the weak shape domain is, in fact, an abstraction of a set of states: the concretization has type $\mathbb{W}^\sharp \to \mathscr{P}(\mathscr{S})$.

We see here that $\mathbb{W}^\sharp$ augments $\mathbb{M}^\sharp$ with the abstraction of a new memory zone A_T, disjoint from A_F. This disjoint heap is represented only by types. A type is associated to every register, and memory cell indexed by A_F. But for this representation to have meaning, it is necessary that the *values* the fixed memory be consistent with their types, relatively to a certain labelling: we say that $h_\mathrm{F} \in \mathrm{R} \uplus \mathrm{A}_\mathrm{F} \to \mathrm{V}_8$ and $\ell^\sharp \in \mathrm{A}_\mathrm{F} \to \mathbb{T}^\sharp$ are $\mathscr{L}$-consistent if $\forall a \in \mathrm{A}_\mathrm{F},\ h(a) \in \gamma_{\mathscr{L},\mathbb{T}}\left(\ell^\sharp(a)\right)$.

Let $(h^\sharp, \ell^\sharp) \in \mathbb{W}^\sharp$. We define

$$\gamma_\mathrm{W}(h^\sharp, \ell^\sharp) = \emptyset$$

if $\gamma_M(h^\sharp)$ and $\ell^\sharp$ are not $\mathscr{L}$-consistent for any labelling $\mathscr{L}$. Otherwise:

$$
\gamma_W\left(h^\sharp, \ell^\sharp\right) = \left\{ h_F \uplus h_T \;\middle|\; \begin{array}{l} h_F \in \gamma_M(h^\sharp) \\ \wedge\ \mathrm{dom}(h_F) \cap \mathrm{dom}(h_T) = \emptyset \\ \wedge\ \exists \mathscr{L} \in \mathrm{Lab},\ \mathscr{L} \text{ is a labelling for } h_T \\ \wedge\ h_F \text{ and } \ell \text{ are } \mathscr{L}\text{-consistent} \\ \wedge\ \forall a \in \mathrm{A_F},\ \mathscr{L}(a) \sqsubseteq_{\mathscr{L},\mathbb{T}} \ell^\sharp(a) \end{array} \right\}
$$

Transfer functions The `store` and `load` transfer functions need to preserve 1. the separation between the flat heap and the typed heap and 2. the types of the typed heap. The `load` function is easier. Its type is:

$$
\mathtt{load} : \mathbb{W}^\sharp \times (\mathbb{N}^\sharp \times \mathbb{T}^\sharp) \times \mathbb{N} \to (\mathbb{N}^\sharp \times \mathbb{T}^\sharp)
$$

where the parameters are the weak shape abstract heap, a typed address with a numeric and a type component, and the size of the region to read, respectively. It returns a typed value, i.e. with a numeric and a type component. $\mathtt{load}((h^\sharp, \ell^\sharp), (n^\sharp, t^\sharp), s)$ proceeds as follows:

- If $t^\sharp$ is a pointer type, or a subtype of a pointer type, i.e. if there exists $u \in \mathbb{T}$ such that $t^\sharp \sqsubseteq_{\mathscr{L},\mathbb{T}} u.0*$, then the abstract address concretizes to a subset of $\mathrm{A_T}$. The numeric component $n^\sharp$ is ignored, and the load returns $\left(\llparenthesis u \rrparenthesis_{\mathscr{L}}, u\right)$.
- Otherwise, if $\gamma_N(n^\sharp) \subseteq \mathrm{A_F}$, then the type component of the address is ignored and the load is forwarded to $h^\sharp$. We return $\left(\mathtt{load}_M(h^\sharp, n^\sharp, s), \top\right)$.
- Otherwise, $\top$ is returned (or an alarm is emitted).

`store` has type:

$$
\mathtt{store} : \mathbb{W}^\sharp \times (\mathbb{N}^\sharp \times \mathbb{T}^\sharp) \times \mathbb{N} \times (\mathbb{N}^\sharp \times \mathbb{T}^\sharp) \to \mathbb{W}^\sharp
$$

where the arguments are the weak shape abstract heap, a typed address with a numeric and type component, the size of the region to read, and a typed value to store, respectively. $\mathtt{store}((h^\sharp, \ell^\sharp), (n_a^\sharp, t_a^\sharp), s, (n_v^\sharp, t_v^\sharp))$ proceeds as follows:

- If $t_a^\sharp$ is a pointer type, or a subtype of a pointer type, i.e. if there exists $u \in \mathbb{T}$ such that $t_a^\sharp \sqsubseteq_{\mathscr{L},\mathbb{T}} u.0*$, then the abstract address points into the typed heap. We check that the types match between value and address, i.e. that $t_v\sharp \sqsubseteq_{\mathscr{L},\mathbb{T}} u$, and the abstract heap $(h^\sharp, \ell^\sharp)$ is returned unchanged. Otherwise, if the types don't match, $\top$ is returned (or an alarm is emitted).

- Otherwise, if $\gamma_N(n_a^\sharp) \subseteq A_F$, then the store is performed in the memory subdomain: we return $\left(\mathtt{store}_M(h^\sharp, n_a^\sharp, s, n_v^\sharp), \ell^\sharp_{\text{new}}\right)$ where $\ell^\sharp_{\text{new}}$ is the result of updating $\ell^\sharp$: $\forall a \in \gamma_N(n_a^\sharp),\ \ell^\sharp_{\text{new}}(a) = t_v^\sharp$.
- Otherwise, we return $\top$ (or an alarm is emitted).

Let us see how this domain enables us to verify our motivating example (Figure 1): all parameters (number of tasks, allocated regions, address of the task array) can be numerically unknown, we still can perform the verification thanks to the weak shape domain. We use the following set of type definitions:

type segment_base = $\{x\ :\ \text{uint32} \mid x > \text{kernel\_last\_addr}\}$
type context = segment_base × uint32
type desc = $\{x\ :\ \text{uint64} \mid\ \text{base}(x) > \text{kernel\_last\_addr}\}$
type params = uint32 × context$[\text{N}_{\text{tasks}}]$.0 $*$ × desc$[\text{N}_{\text{tasks}}]$.0$*$

Here, `kernel_last_addr` and N_{tasks} are what we call symbolic constants: they represent values which are not known precisely at the time of the analysis. It is simple to extend the type syntax to use such symbolic constants in type predicates and as lengths of arrays; we did not detail this point in the formalization above for clarity reasons.

- `kernel_last_addr` represents the highest address of the kernel memory. By constraining all segment bases to be greater than this constant, we make sure that the memory regions accessible to tasks never intersect kernel memory.
- N_{tasks} represents the number of tasks that run on the system. Following a common trend in embedded systems design, that number is a consant, i.e. to change the number of tasks running on the kernel, one must compile a new application image and link it with the kernel.

With this set of type definitions, the weak shape domain is able to analyze the machine code resulting from the compilation of Figure 1 and verify:

1. The absence of out-of-bounds array accesses, and other run-time errors.
2. That the two predicates constraining memory protection segments hold.

Combined, these two properties suffice to prove the absence of privilege escalation on this example kernel.

4 Conclusion

By combining standard abstract interpretation techniques with a novel weak shape abstract domain, we were able to build an analyzer to verify safety and security properties directly on machine code efficiently, with a low annotation burden.

References

[1] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." In: *POPL*. 1977.

[2] Robin David et al. "BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis." In: *SANER*. 2016 IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER). Suita: IEEE, Mar. 2016, pp. 653–656.

[3] Adel Djoudi, Sébastien Bardin, and Éric Goubault. "Recovering High-Level Conditions from Binary Programs." In: *FM 2016: Formal Methods*. Ed. by John Fitzgerald et al. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 235–253.

[4] Graeme Gange et al. "An Abstract Domain of Uninterpreted Functions." In: *VMCAI*. Verification, Model Checking, and Abstract Interpretation. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 85–103.

[5] Philippe Granger. "Static Analysis of Arithmetical Congruences." In: *Int. J. Comput. Math.* 30.3-4 (1989), pp. 165–190.

[6] Gary A. Kildall. "A Unified Approach to Global Program Optimization." In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '73*. The 1st Annual ACM SIGACT-SIGPLAN Symposium. Boston, Massachusetts: ACM Press, 1973, pp. 194–206.

[7] Johannes Kinder, Florian Zuleger, and Helmut Veith. “An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries.” In: *VMCAI*. Verification, Model Checking, and Abstract Interpretation. Ed. by Neil D. Jones and Markus Müller-Olm. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 214–228.

[8] Antoine Miné. “Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics.” In: *LCTES*. Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems. LCTES ’06. New York, NY, USA: ACM, 2006, pp. 54–63.

[9] Antoine Miné. “Symbolic Methods to Enhance the Precision of Numerical Abstract Domains.” In: *VMCAI*. Verification, Model Checking, and Abstract Interpretation. Ed. by E. Allen Emerson and Kedar S. Namjoshi. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 348–363.

[10] Olivier Nicole et al. “No Crash, No Exploit: Automated Verification of Embedded Kernels.” In: *RTAS*. IEEE Real-Time and Embedded Technology and Applications Symposium. Online, 2021.

[11] H. G. Rice. “Classes of Recursively Enumerable Sets and Their Decision Problems.” In: *Trans. Amer. Math. Soc.* 74.2 (1953), pp. 358–366. ISSN: 0002-9947, 1088-6850.