Verified Abstract Interpretation Techniques for Disassembling Low-level Self-modifying Code

Sandrine Blazy, Vincent Laporte, David Pichardie

Verified Abstract Interpretation Techniques for Disassembling Low-level Self-modifying Code *

Sandrine Blazy[1], Vincent Laporte[1], and David Pichardie[2]

[1] Université Rennes 1 – IRISA – Inria
[2] ENS Rennes – IRISA – Inria

Abstract Static analysis of binary code is challenging for several reasons. In particular, standard static analysis techniques operate over control flow graphs, which are not available when dealing with self-modifying programs which can modify their own code at runtime. We formalize in the Coq proof assistant some key abstract interpretation techniques that automatically extract memory safety properties from binary code. Our analyzer is formally proved correct and has been run on several self-modifying challenges, provided by Cai et al. in their PLDI 2007 paper.

1 Introduction

Abstract interpretation [9] provides advanced static analysis techniques with strong semantic foundations. It has been applied on a large variety of programming languages. Still, specific care is required when adapting these techniques to low-level code, specially when the program to be analyzed comes in the form of a sequence of bits and must first be disassembled. Disassembling is the process of translating a program from a machine friendly binary format to a textual representation of its instructions. It requires to *decode* the instructions (i.e., understand which instruction is represented by each particular bit pattern) but also to precisely locate the instructions in memory. Indeed instructions may be interleaved with data or arbitrary padding. Moreover once encoded, instructions may have various byte sizes and may not be well aligned in memory, so that a single byte may belong to several instructions.

To thwart the problem of locating the instructions in a program, one must follow its control flow. However, this task is not easy because of the indirect jumps, whose targets are unknown until runtime. A static analysis needs to know precisely enough the values that the expression denoting the jump target may evaluate to. In addition, instructions may be produced at runtime, as a result of the very execution of the program. Such programs are called *self-modifying* programs; they are commonly used in security as an obfuscation technique, as well as in just-in-time compilation. Analyzing a binary code is mandatory when this code is the only available part of a software. Most of standard reverse engineering

* This work was supported by Agence Nationale de la Recherche, grant number ANR-11-INSE-003 Verasco.

tools (e.g., IDA Pro) cannot disassemble and analyze self-modifying programs. In order to disassemble and analyze such programs, one must very precisely understand which instructions are written and where. And for all programs, one must check every single memory write to decide whether it modifies the program code.

Self-modifying programs are also beyond the scope of the vast majority of formal semantics of programming languages. Indeed a prerequisite in such semantics is the isolation and the non-modification of code in memory. Turning to verified static analyses, they operate over toy languages [5, 16] or more recently over realistic C-like languages [18, 3], but they assume that the control-flow graph is extracted by a preliminary step, and thus they do not encompass techniques devoted to self-modifying code.

In this paper, we formalize with the Coq proof assistant, key static analysis techniques to predict the possible targets of the computed jumps and make precise which instructions alter the code and how, while ensuring that the other instructions do not modify the program. Our static analysis techniques rely on two main components classically used in abstract interpretation, abstract domains and fixpoint iterators, that we detail in this paper. The complete Coq development is available online [8].

Our formalization effort is divided in three parts. Firstly, we formalize a small binary language in which code is handled as regular mutable data. Secondly, we formalize and prove correct an abstract interpreter that takes as input an initial memory state, computes an over-approximation of the reachable states that may be generated during the program execution, and then checks that all reachable states maintain memory safety. Finally, we extract from our formalization an executable OCaml tool that we run on several self-modifying challenges, provided by Cai et al. [6].

The paper makes the following contributions.

- We push further the limit in terms of verified static analysis by tackling the specific challenge of binary self-modifying programs, such as fixpoint iteration without control-flow graph and simple trace partitioning [12].
- We provide a complementary approach to [6] by automatically inferring the required state invariants that enforce memory safety. Indeed, the axiomatic semantics of [6] requires programs to be manually annotated with invariants written in a specific program logic.

The remainder of this paper is organized as follows. First, Section 2 briefly introduces the static analysis techniques we formalized. Then, Section 3 details our formalization: it defines the semantics of our low-level language and details our abstract interpreter. Section 4 describes some improvements that we made to the abstract interpreter, as well as the experimental evaluation of our implementation. Related work is discussed in Section 5, followed by concluding remarks.

2 Disassembling by Abstract Interpretation

We now present the main principles of our analysis on the program shown in Figure 1. It is printed as a sequence of bytes (on the extreme left) as well as

under a disassembled form (on the extreme right) for readability purposes. This program, as we will see, is self-modifying, so these bytes correspond to the initial content of the memory from addresses 0 to 11. The remaining of the memory (addresses in $[-2^{32}; -1] \cup [12; 2^{32} - 1]$), as well as the content of the registers, is unknown and can be regarded as the program input.

All our example programs target a machine operating over a low-level memory made of 2^{32} cells, eight registers (`R0`, ... `R7`), and flags — boolean registers that are set by comparison instructions. Each memory cell or register stores a 32 bits integer value, that may be used as an address in the memory. Programs are stored as regular data in the memory; their execution starts from address zero. Nevertheless, throughout this paper we write the programs using the following custom syntax. The instruction `cst v → r` loads register `r` with the given value `v`; `cmp r, r'` denotes the comparison of the contents of registers `r` and `r'`; `gotoLE d` is a conditional jump to `d`, it is taken if in the previous comparison the content of `r'` was less than or equal to the one of `r`; `goto d` is an unconditional jump to `d`. The instruction `load *r → r'` and `store r' → *r` denote accesses to memory at the address given in register `r`; and `halt r` halts the machine with as final value the content of register `r`.

The programming language we consider is inspired from x86 assembly; notably instructions have variable size (one or two bytes, e.g., the length of the instruction stored at line 1 is two bytes) and conditional jumps rely on flags. In this setting, a program is no more than an initial memory state, and a program point is simply the address of the next instruction to execute.

Initial program	Possible final program	Initial assembly listing
07000607	07000607	0: cmp R6, R7
03000000	03000000	1: gotoLE **5**
00000005	**00000004**	2:
00000000	00000000	3: halt R0
00000100	00000100	4: halt R1
09000000	09000000	5: cst 4 → R0
00000004	00000004	6:
09000002	09000002	7: cst 2 → R2
00000002	00000002	8:
05000002	05000002	9: store R0 → *R2
04000000	04000000	10: goto 1
00000001	00000001	11:

Figure 1. A self-modifying program: as a byte sequence (left); after some execution steps (middle); assembly source (right).

In order to understand the behavior of this program, one can follow its code as it is executed starting from the entry point (byte 0). The first instruction compares the (statically unknown) content of two registers. This comparison modifies only the states of the flags. Then, depending on the outcome of this

comparison, the execution proceeds either on the following instruction (stored at byte 3), or from byte 5. Executing the block from byte 5 will modify the byte 2 belonging to the `gotoLE` instruction (highlighted in Figure 1); more precisely it will change the jump destination from 5 to 4: the `store R0 → *R2` instruction writes the content of register `R0` (namely 4) in memory at the address given in register `R2` (namely 2). Notice that a program may directly read from or write to any memory cell: we assume that there is no protection mechanism as provided by usual operating systems. After the modification is performed, the execution jumps back to the modified instruction, jumps to byte 4 then halts, with final value the content of register `R1`.

This example highlights that the code of a program (or its control-flow graph) is not necessarily a static property of this program: it may vary as the program runs. To correctly analyze such a program, one must discover, during the fixpoint iteration, the two possible states of the instruction at locations 1 and 2 and its two possible targets. More specially, we need at least to know, for each program point (i.e., memory location), which instructions may be decoded from there when the execution reaches this point. This in turn requires to know what are the values that the program operates on. We therefore devise a value analysis that computes, for each reachable program point (i.e., in a *flow sensitive* way) an over-approximation of the content of the memory and the registers, and the state of the flags when the execution reaches that point.

The analysis relies on a numeric abstract domain $\mathsf{N}^\sharp$ that provides a representation for sets of machine integers ($\gamma_{\mathsf{N}} \in \mathsf{N}^\sharp \to \mathcal{P}(\texttt{int})$) and abstract arithmetic operations. Relying on such a numeric domain, one can build abstract transformers that model the execution of each instruction over an abstract memory that maps locations (i.e., memory addresses[3] and registers) to abstract numeric values. An abstract state is then a mapping that attaches such an abstract memory to each program point of the program, and thus belongs to $\texttt{addr} \to \big((\texttt{addr} + \texttt{reg}) \to \mathsf{N}^\sharp\big)$.

To perform one abstract execution step, from a program point `pp` and an abstract memory state $\mathrm{m}^\sharp$ that is attached to `pp`, we first enumerate all instructions that may be decoded from the set $\gamma_{\mathsf{N}}(\mathrm{m}^\sharp(\texttt{pp}))$. Then for each of such instructions, we apply the matching abstract transformer. This yields a new set of successor states whose program points are dynamically discovered during the fixpoint iteration.

The abstract interpretation of a whole program iteratively builds an approximation executing all reachable instructions until nothing new is learned. This iterative process may not terminate, since there might be infinite increasing chains in the abstract search space. As usual in abstract interpretation, we accelerate the iteration using widening operations [9]. Once a stable approximation is finally reached, an approximation of the program listing or control-flow graph can be produced.

To illustrate this process, Figure 2 shows how the analysis of the program from Figure 1 proceeds. We do not expose a whole abstract memory but only the underlying control-flow graph it represents. On this specific example, three

[3] Type `addr` is a synonym of `int`, the type of machine integers.

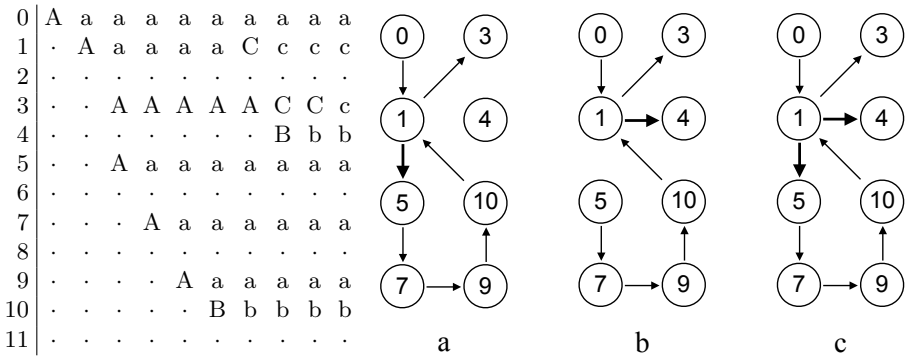

Figure 2. Iterative fixpoint computation

different graphs are encountered during the analysis. For each program point pp, we represent a node with same name and link it with all the possible successor nodes according to the decoding of the set $\gamma_{\mathsf{N}}(\mathrm{m}^\sharp(\mathrm{pp}))$. The array shows the construction of the fixpoint: each line represents a program point and the columns represent the iterations of the analysis. In each array cell lies the name of the control-flow graph representing the abstract memory for the given program point during the given iteration; a dot stands for an unreachable program point. The array cells whose content is in upper case highlight the program points that need to be analyzed: they are the worklist.

Initially, at iteration 0, only program point 0 is known to be reachable and the memory is known to exactly contain the program, denoted by the first control-flow graph. The only successor of point 0 is point 1 and it is updated at the next iteration. After a few iterations, point 9 is reached and the abstract control-flow graph a is updated into a control-flow graph b that is propagated to point 10. After a few more iterations, the process converges.

In addition to a control-flow graph or an assembly listing, more properties can be deduced from the analysis result. We can prove safety properties about the analyzed program, like the fact that its execution is never stuck. Since the semantics only defines the good behaviors of programs, unsafe programs reach states that are not final and from which no further execution step is possible (e.g., the byte sequence at current program point is not the valid encoding of an instruction).

The analysis produces an over-approximation of the set of reachable states. In particular, a superset of the reachable program points is computed, and for each of these program points, an over-approximation of the memory state when the execution reaches this program point is available. Thus we can check that for every program point that may be reached, the next execution step from this point cannot be stuck. This verification procedure is formally verified, as described in the following section.

3 Formalization

The static analyzer is specified, programmed and proved correct using the Coq proof assistant. This involves several steps that are described in this section: first, define the semantics of a binary language, then design abstract domains and abstract transformers, as well as write a fixpoint iterator, and lastly state and prove soundness properties about the results of the static analysis.

3.1 Concrete Syntax and Semantics

The programming language in which are written the programs to analyze is formalized using the syntax shown on Figure 3. So as to model a binary language, we introduce a decoding function `dec (mem: (addr → int)) (pp: int) : option (instruction * nat)` that given a memory `mem` (i.e., a function from addresses to values) and an address `pp` yields the instruction stored from this address along with its byte size. Since not all integer sequences are valid encodings, this decoding may fail (hence the `option` type[4]). In order to be able to conveniently write programs, there is also a matching encoding function. However the development does not depend on it at all.

```
Inductive reg  := R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7.
Inductive flag := FLE | FLT | FEQ.
Inductive instruction :=
(* arithmetic *)
| ICst (v:int) (dst:reg) | ICmp (src dst: reg)
| IBinop (op: int_binary_operation) (src dst: reg)
(* memory *)
| ILoad  (src dst: reg) | IStore (src dst: reg)
(* control *)
| IGoto (tgt: addr) | IGotoInd (r: reg) | IGotoCond (f: flag) (tgt: addr)
| ISkip | IHalt (r: reg).
```

Figure 3. Language syntax

The language semantics is given as a small-step transition relation between machine states. A machine state may be $\langle$`pp, f, r, m`$\rangle$ where `pp` is the current program point (address of the next instruction to be executed), `f` is the current flag state, `r` is the current register state, and `m` is the current memory. Such a tuple is called a machine *configuration* (type `machine_config`). Otherwise, a machine state is $\lceil$`v`$\rceil$, meaning that the program stopped returning the value `v`.

The semantics is defined as a set of rules of the following shape:

$$\frac{\texttt{dec m pp} = \lfloor(\texttt{i, z})\rfloor}{\langle \texttt{pp, f, r, m}\rangle \rightsquigarrow \langle \texttt{pp', f', r', m'}\rangle}.$$

[4] Values of type `option A` are either `None` or $\lfloor a \rfloor$ with a a value of type `A`.

The premise states that decoding the bytes in memory `m` from address `pp` yields the instruction `i` whose size in memory is `z`. Then each rule describes how to execute a particular instruction at program point `pp` in memory `m` with flag state `f` and register state `r`. In each case, most of the state is kept unchanged. Instructions that are not branching proceed their execution at program point `pp+z` (since `z` is the size of this instruction once encoded). The whole set of rules can be found in the Coq development [8]. We describe only some of them. Instruction `ICmp rs rd` updates the flag state according to the comparison of the values held by the two involved registers. Conditional jump instruction `IGotoCond c v` jumps to address `v` or falls through to `pp+z` depending on the current state of flag `c`. Indirect jump instruction `IGotoInd rd` proceeds at the program point found in register `rd`.

Finally, we define the semantics $[\![\texttt{P}]\!]$ of a program `P` as the set of states that are reachable from an initial state, with current program point zero and memory `P` (where $\rightsquigarrow^\star$ denotes the reflexive-transitive closure of the small-step relation).

$$[\![\texttt{P}]\!] = \{s \mid \exists \texttt{f r}, \langle \texttt{0, f, r, P} \rangle \rightsquigarrow^\star s\}$$

3.2 Abstract Interpreter

In order to analyze programs, we build an abstract interpreter, i.e., an executable semantics that operates over abstract elements, each of them representing many concrete machine configurations. Such an abstract domain provides operators that model basic concrete operations: read a value from a register, store some value at some address in memory, and so on. The static analyzer then computes a fixpoint within the abstract domain, that over-approximates all reachable states of the analyzed program.

We first describe our abstract domain before we head to the abstract semantics and fixpoint computation. An abstract memory domain is a carrier type along with some primitive operators whose signatures are given in Figure 4. The `ab_num` type refers to a numeric abstract domain, as described in [3]: we only require that this type is equipped with a concretization to sets of machine integers and abstract transformers corresponding to arithmetic operations.

The carrier type `ab_mc` is equipped with a lattice structure. An object of this type represents a set of concrete machine states, as described by the primitive `gamma`. It can be queried for the values stored in some register (`var`) or at some known memory address (`load_single`); these operators return an abstract numeric value. Other operators enable us to alter an abstract state, like `assign` that sets the contents of a register to a given abstract numeric value, and `store_single` that similarly updates the memory at a given address.

All these operators obey some specifications. As an example, the `load_sound` property states that given a concrete state `m` in the concretization of an abstract state `ab`, the concrete value stored at any address `a` in `m` is over-approximated by the abstract value returned by the matching abstract load. The γ symbol is overloaded through the use of type classes: its first occurrence refers to the concretization from the abstract memory domain (the `gamma` field of record `mem_dom`) and its second occurrence is the concretization from the numeric domain `ab_num`.

```
Record mem_dom (ab_num ab_mc: Type) :=
{ as_wl: weak_lattice ab_mc
; var: ab_mc → reg → ab_num
; load_single: ab_mc → addr → ab_num
; store_single: ab_mc → addr → ab_num → ab_mc
; assign: ab_mc → reg → ab_num → ab_mc
(* more abstract operators omitted *)
; gamma: gamma_op ab_mc machine_config
; as_adom : adom ab_mc machine_config as_wl gamma
; load_sound: ∀ ab:ab_mc, ∀ m: machine_config,
      m ∈ γ(ab) → ∀ a:addr, m(a) ∈ γ(load_single ab a)
(* more properties omitted *)  }.
```

Figure 4. Signature of abstract memory domains (excerpt)

Such an abstract memory domain is implemented using two maps: from registers to abstract numeric values to represent the register state and from values to abstract numeric values to represent the memory.

```
Record ab_machine_config :=
    { ab_reg: Map [ reg, ab_num ] ; ab_mem: Map [ addr, ab_num ] }.
```

To prevent the domain of the `ab_mem` map from infinitely growing, we bound it by a finite set computed before the analysis: the analysis will try to compute some information only for the memory addresses found in this set [1]. The content of this set does not alter its soundness: the values stored at addresses not in it are unknown and the analyzer makes no assumptions about them. On the other hand, the success of the analysis and its precision depend on it. In particular, the analyzed set must cover the whole code segment.

As a second layer, we build abstract transformers over any such abstract domain. Consider for instance the abstract load presented in Figure 5; it is used to analyze any `ILoad` instruction (`T` denotes a record of type `mem_dom ab_num ab_mc`). The source address may not be exactly known, but only represented by an abstract numeric value `a`. Since any address in γ`(a)` may be read, we have to query all of them and take the least upper bound of all values that may be stored at any of these addresses: $\bigsqcup\{$`T.(load_single) m x` $\mid$ `x` $\in \gamma($`a`$)\}$. However the set of concrete addresses may be huge and care must be taken: if the size of this set exceeds some threshold, the analysis gives up on this load and yields `top`, representing all possible values.

We build enough such abstract transformers to be able to analyze any instruction (function `ab_post_single`, shown in Figure 6). This function returns a list of possible next states, each of which being either `Hlt v` (the program halts returning a value approximated by `v`) or `Run pp m` (the execution proceeds at program point `pp` in a configuration approximated by `m`) or `GiveUp` (the analysis is too imprecise to compute anything meaningful). The computed jump (`IGotoInd`)

```
Inductive botlift (A:Type) : Type := Bot | NotBot (x:A).
Definition load_many (m: ab_mc) (a: ab_num) : botlift ab_num :=
  match concretize_with_care a with
  | Just addr_set ⇒ IntSet.fold
      (λ acc addr, acc ⊔ NotBot (T.(load_single) m addr)) addr_set Bot
  | All ⇒ NotBot top end.
```

Figure 5. Example of abstract transformer

```
Inductive ab_post_res := Hlt(v:ab_num) | Run(pp:addr)(m:ab_mc) | GiveUp.
Definition ab_post_single (m:ab_mc) (pp:addr) (instr:instruction * nat)
  : list ab_post_res := match instr with
  | (IHalt rs, z) ⇒ Hlt (T.(var) m rs) :: nil
  | (ISkip, z) ⇒ Run (pp + z) m :: nil
  | (IGoto v, z) ⇒ Run v m :: nil
  | (IGotoInd rs, z) ⇒ match concretize_with_care (T.(var) m rs) with
      | Just tgt ⇒ IntSet.fold (λ acc addr, Run addr m :: acc) tgt nil
      | All ⇒ GiveUp :: nil end
  | (IStore rs rd, z) ⇒
      Run (pp + z) (store_many m (T.(var) m rd) (T.(var) m rs)) :: nil
  | (ILoad rs rd, z) ⇒ match load_many m (T.(var) m rs) with
      | NotBot v ⇒ Run (pp + z) (T.(assign) m rd v) :: nil
      | Bot ⇒ nil end
  | (ICmp rs rd, z) ⇒ Run (pp + z) (T.(compare) m rs rd ) :: nil
  | (ICst v rd, z) ⇒ Run (pp + z) (T.(assign) m rd v) :: nil
    (* ... *) end.
Definition ab_post_many (pp: addr) (m:ab_mc) : list ab_post_res :=
  match abstract_decode_at pp m with
  | Just instr ⇒ flat_map (ab_post_single m pp) instr
  | All ⇒ GiveUp :: nil end.
```

Figure 6. Abstract small-step semantics (excerpt)

also has a dedicated abstract transformer (inlined in Figure 6): in order to know from where to continue the analysis, we have to enumerate all possible targets.

Then, function `ab_post_many` performs one execution step in the abstract. To do so, we first need to identify what is the next instruction, i.e., to decode in the abstract memory from the current program point. This may require to enumerate all concrete values that may be stored at this address. Therefore this abstract decoding either returns a set of possible next instructions or gives up. In such a case, the whole analysis will abort since the analyzed program is unknown.

Finally, the abstract semantics is iteratively applied until a fixpoint is reached following a worklist algorithm as the one found in [1, § 3.4]. However there may be infinite ascending chains, so to ensure termination we need to apply widening operators instead of regular joins frequently enough during the search.

In our setting, with no control-flow graph available, the widening is applied on every back edge, but the implementation makes it easy to try different widening strategies. So as to convince Coq that the analysis indeed terminates, we rely on a counter (known as fuel) that obviously decreases at each iteration; when it reaches zero, the analyzer must give up.

To enhance the precision, we have introduced three more techniques: a dedicated domain to abstract the flag state, a partitioning of the state space, and a use of abstract instructions. They will be described in the next section.

3.3 Soundness of the Abstract Interpreter

We now describe the formal verification of our analyzer. The soundness property we ensure is that the result of the analysis of a program `P` over-approximates its semantics $[\![\texttt{P}]\!]$. This involves on one hand a proof that the analysis result is indeed a fixpoint of the abstract semantics and on the other hand a proof that the abstract semantics is correct with respect to the concrete one.

The soundness of the abstract semantics is expressed by the following lemma, which reads: given an abstract state `ab` and a concrete one `m` in the concretization of `ab`, for each concrete small-step `m ⇝ m'`, there exists a result `ab'` in the list `ab_post_single m.(pc) ab` that over-approximates `m'`. Our use of Coq type classes enables us to extensively overload the γ notation and write this statement in a concise way as follows.

```
Lemma ab_post_many_correct :
  ∀ (m:machine_config) (m':machine_state) (ab:ab_mc),
    m ∈ γ(ab) → m ⇝ m' → m' ∈ γ(ab_post_single m.(pc) ab).
```

The proof of this lemma follows from the soundness of the various abstract domains (as `load_sound` in Figure 4), transformers and decoder.

```
Lemma abstract_decode_at_sound : ∀ (m:machine_config)(ab:ab_mc)(pp:addr),
    m ∈ γ(ab) → dec m.(mc_mem) pp ∈ γ(abstract_decode_at pp ab).
```

The proof that the analyzer produces a fixpoint is not done directly. Instead, we rely on *a posteriori* verification: we do not trust the fixpoint computation and instead program and prove a checker called `validate_fixpoint`. Its specification, proved thanks to the previous lemma, reads as follows.

```
Lemma validate_correct : ∀ (P: memory) (dom: list addr) (E: AbEnv),
    validate_fixpoint P dom E = true → [[P]] ⊆ γ(E).
```

Going through this additional programming effort has various benefits: a direct proof of the fixpoint iterator would be very hard; we can adapt the iteration strategy, optimize the algorithm and so on with no additional proof effort.

This validation checks two properties of the result E: that it over-approximates the initial state; and that it is a post-fixpoint of the abstract semantics, i.e., for each abstract state in the result, performing one abstract step leads to abstract states that are already included in the result. These properties, combined to the soundness of the abstract semantics, ensure the conclusion of this lemma.

Finally we pack together the iterator and the checker with another operation performed on sound results that checks for its safety. The resulting `analysis` enjoys the following property: if, given a program `P`, it outputs some result, then that program is safe.

```
Theorem analysis_sound : ∀ (P: memory) (dom: list addr) (fuel: nat)
  (ab_num: num_dom_index), analysis ab_num P dom fuel ≠ None → safe P.
```

The arguments of the `analysis` program are the program to analyze, the list of addresses in memory to track, the counter that enforces termination and the name of the numeric domain to use. We provide two numeric domains: intervals with congruence information and finite sets.

4 Case Studies and Analysis Extensions

The extraction mechanism of Coq enables us to generate an OCaml program from our development and to link it with a front-end. Hence we can automatically analyze programs and prove them safe. This section shows the behavior of our analyzer on chosen examples, most of them taken from [6] (they have been rewritten to fit our custom syntax). All examples are written in an assembly-like syntax with some syntactic sugar: labels refer to byte offsets in the encoded program, the `enc(I)` notation denotes the encoding of the instruction `I`. The study of some examples highlights the limits of the basic technique presented before and suggests to refine the analyzer as we describe below. The source code of all the examples that are mentioned thereafter is available on the companion web site [8].

4.1 Basic Example

The multilevel runtime code generation program of Figure 7 is a program that, when executed, writes some code to line `gen` on and runs it; this generated program, in turn, writes some more code at line `ggen` and runs it. Finally execution starts again from the beginning. Moreover, at each iteration, register `R6` is incremented.

The analysis of such a program follows its concrete execution and exactly computes the content of each register at each program point. It thus correctly tracks what values are written and where, so as to be able to analyze the program as it is generated.

However, when the execution reaches program point `loop` again, both states that may lead to that program point are merged. And the analysis of the loop body starts again. After the first iteration, the program text is exactly known, but each iteration yields more information about the dynamic content of register `R6`. Therefore we apply widening steps to ensure the termination of the analysis. Finally, the set of reachable program points is exactly computed and for each of them, we know what instruction will be executed from there.

Many self-modifying programs are successfully analyzed in a similar way: opcode modification, code obfuscation, and code checking [8].

```
      cst 0 → R6
      cst 1 → R5
loop: add R5 → R6
      cst gen → R0
      cst enc(store R1 → *R2) → R1
      store R1 → *R0
      cst enc(goto R2) → R1
      cst gen + 1 → R0
      store R1 → *R0
      cst ggen → R2
      cst loop → R0
      cst enc(goto R0) → R1
      goto gen
gen:  skip
      skip
ggen: skip
```

Figure 7. Multilevel Runtime Code Generation

```
   cst -128 → R6
   add R6 → R1
   cmp R6, R1
   gotoLT ko
   cst -96 → R7
   cmp R1, R7
   gotoLE ko
   store R0 → *R1
ko:halt R0
```

Figure 8. Array bounds check

4.2 A First Extension: Dealing with Flags

The example program in Figure 8 illustrates the abstract domain for the flags. This program stores the content of `R0` in an array (stored in memory from address -128 to address -96) at the offset given in register `R1`. Before that store, checks are performed to ensure that the provided offset lies inside the bounds of the array. The destination address is compared against the lowest and highest addresses of the array; if any of the comparisons fails, then the store is bypassed.

To properly analyze this program, we need to understand that the store does not alter the code. When analyzing a conditional branch instruction, the abstract state is refined differently at its two targets. However, the only information we have is about one flag, whereas the comparison that sets this flag operated on the content of registers. We therefore need to keep the link between the flags and the registers.

To this end, we extend our `ab_machine_config` record with a field containing an optional pair of registers `ab_reg: option (reg * reg)`. It enables the analyzer to remember which registers were involved in the last comparison (the `None` value is used when this information is unknown). With such information available, even though the conditional jump is not directly linked to the comparison operation, we can gain some precision in the various branches.

Indeed, when we assume that the first conditional branch is not taken, the flag state is abstracted by the pair $\lfloor$`(R6,R1)`$\rfloor$, so we refine our knowledge about register `R1`: its content is not less than the -128. Similarly, when we assume that the second conditional branch is not taken, the abstract flag state is $\lfloor$`(R1,R7)`$\rfloor$, so we can finally infer that the content of register `R1` is in the bounds.

This extension of the abstract domain increases a lot the precision of the analyzer on some programs, yet has little impact on the formalization: we need

to explain its lattice structure (top element, order and least upper bound) and define its concretization. Then it enables us to program more precise primitives (namely `compare` and `assume`) that we must prove correct. No other part of the development is modified.

4.3 A Second Extension: Trace Partitioning

Some self-modifying programs store in the same memory space various pieces of their code. Successfully analyzing such programs requires not to merge these different code fragments, i.e., we need to distinguish in the execution which code is being run: flow sensitivity is not enough. To this end we use a specific form of *trace partitioning* [12] that makes an analysis sensitive to the value of a particular memory location.

Consider as an example the *polymorphic* program [8] that we briefly describe below. Polymorphism here refers to a technique used by for instance viruses that change their code while preserving their behavior, so as to hide their presence. The main loop of this program repeatedly adds forty-two to register `R3`. However, it is obfuscated in two ways. First, the source code initially contains a jump to some random address. But this instruction will be overwritten before it is executed. Second, this bad instruction is written back, but at a different address. So when the execution reaches the beginning of the loop, the program stored in memory is one of two different versions, both featuring the unsafe jump.

When analyzing this program, the abstract state computed at the beginning of the loop must over-approximate the two program versions. Unfortunately it is not possible to analyze the mere superposition of both versions, in which the unsafe jump may occur. The two versions can be distinguished through, for instance, the value at address 12. We therefore prevent the merging of any two states that disagree on the value stored at this address. Two different abstract states are then computed at each program point in the loop, as if the loop were unrolled once.

More generally, the analysis is parametrized by a partitioning criterion δ: `ab_mc`$\rightarrow$`int` that maps abstract states to values (the criterion used in this example maps an abstract state m to the value stored at address 12 in all concrete states represented by m; or to an arbitrary constant if there may be many values at this address). No abstract states that differ according to this criterion are merged. Taking a constant criterion amounts to disabling this partitioning. The abstract interpreter now computes for each program point, a map from criterion values to abstract states (rather than only one abstract state). Given such an environment E, a program point `pp`, and a value v, if there is an abstract state m such that $E(\texttt{pp})(v) = \lfloor m \rfloor$, then $\delta(m) = v$. Such an environment E represents the following set of machine configurations:

$$\gamma(E) = \{c \in \texttt{machine\_config} \mid \exists v,\ c \in \gamma\left(E(c.\texttt{pc})(v)\right)\}$$

To implement this technique, we do not need to modify the abstract domain, but only the iterator and fixpoint checker. The worklist holds pairs (program

point, criterion value) rather than simple program points, and the iterator and fixpoint checker (along with its proof) straightforwardly adapted. The safety checker does not need to be updated since we can forget the partitioning before applying the original safety check.

Thanks to this technique, we can selectively enhance the precision of the analysis and correctly handle challenging self-modifying programs: control-flow modification, mutual modification, and code encryption [8]. However, the analyst must manually pick a suitable criterion for each program to analyze.

4.4 A Third Extension: Abstract Decoding

The program in Figure 9 computes the n^{th} Fibonacci number in register `R2`, where n is an input value read from address -1 and held in register `R0`. There is a for-loop in which register `R1` goes from 1 to n and some constant value is added to register `R2`. The trick is that the actual constant (which is encoded as part of an instruction and is stored at the address held in `R6`) is overwritten at each iteration by the previous value of `R2`.

When analyzing this program, we cannot infer much information about the content of the patched cell. Therefore, we cannot enumerate all instructions that may be stored at the patched point. So we introduce abstract instructions: instructions that are not exactly known, but of which some part is abstracted by a suitable abstract domain. Here we only need to abstract values using a numeric domain. With such a tool, we can decode *in the abstract*: the analyzer does not recover the exact instructions of the program, but only the information that some (unknown) value is loaded into register `R4`, which is harmless (no stores and no jumps depend on it).

This self-modifying code pattern, in which only part of an instruction is overwritten occurs also in the vector dot product example [8] where specialized multiplication instructions are emitted depending on an input vector.

The techniques presented here enable us to automatically prove the safety of various self-modifying programs including almost all the examples of [6]. Out of twelve, only two cannot be dealt with. The self-replicating example is a program that fills the memory with copies of itself: the code, being infinite, cannot be represented with our abstract domain. The bootloader example does not fit in the considered machine model, as it calls BIOS interrupts and reads files. Our Coq development [8] features all the extensions along with their correctness proofs.

```
      cst -1 → R7            gotoLE last             add R4 → R2
      load *R7 → R0          cst 1 → R7              store R3 → *R6
      cst key+1 → R6         add R7 → R1             goto loop
      cst 1 → R1             cst 0 → R3       last: halt R2
      cst 1 → R2             add R2 → R3
loop: cmp R1, R0       key:  cst 0 → R4
```

Figure 9. Fibonacci

5 Related Work

Most of the previous works on mechanized verification of static analyzes focused on standard data-flow frameworks [13] or abstract interpretation for small imperative structured languages [5, 16]. In a previous work [3], we formally verified a value analysis for an intermediate language of the Compcert C compiler toolchain. The current work shares the same notion of abstract numerical domain but develops its own notion of memory abstraction, dynamic control-flow graph reconstruction and trace partitioning.

The current work formalizes more advanced abstract interpretation techniques, targeting self-modifying low-level code, and is based on several recent non-verified static analyses. A large amount of work was done by Balakrishnan et al. in this area [1]. Control-flow graph reconstruction was specially studied by Kinder et al. [12] and Bardin et al. [2]. Still, these works are unsound with respect to self-modifying code. Bonfante et al. provide a paper-and-pencil operational semantics for self-modifying programs [4].

Our current work tackles a core subset of a self-modifying low-level programming language. More realistic formalizations of x86 semantics were proposed [15, 14, 11] but none of them handles the problem of disassembling self-modifying programs. Our work complements other verification efforts of low-level programs [7, 6, 10] based on program logics. While we provide automatic inference of loop invariants, they are able to handle more expressive correctness properties.

6 Conclusion and Perspectives

This work provides the first verified static analyis for self-modifying programs. In order to tackle this challenge, we formalized original techniques such as control-flow graph reconstruction and partitioning. We formalized these techniques on a small core language but we managed to verify ten out of twelve of the challenges proposed in [6].

An important further work is to scale these technique on more realistic Coq language models [14, 11]. Developing directly an analyzer on these representations may be a huge development task because of the number of instructions to handle. One strategy could be to relate on a good intermediate representation such as the one proposed by Rocksalt [14]. Our current work does not consider the specific challenge of call stack reconstruction [1] that may require some form of verified alias analysis [17]. This is an important place for further work.

References

[1] G. Balakrishnan and T. W. Reps. "WYSINWYX: What you see is not what you eXecute." In: *ACM Trans. Program. Lang. Syst.* 32.6 (2010).

[2] S. Bardin, P. Herrmann, and F. Védrine. "Refinement-Based CFG Reconstruction from Unstructured Programs." In: *VMCAI*. Vol. 6538. LNCS. Springer, 2011, pp. 54–69.

[3] S. Blazy et al. “Formal Verification of a C Value Analysis Based on Abstract Interpretation.” In: *SAS*. Vol. 7935. LNCS. Springer, 2013, pp. 324–344.
[4] G. Bonfante, J.Y. Marion, and D. Reynaud-Plantey. “A Computability Perspective on Self-Modifying Programs.” In: *SEFM*. 2009, pp. 231–239.
[5] D. Cachera and D. Pichardie. “A Certified Denotational Abstract Interpreter.” In: *Proc. of ITP-10*. Vol. 6172. LNCS. Springer, 2010, pp. 9–24.
[6] H. Cai, Z. Shao, and A. Vaynberg. “Certified Self-Modifying Code.” In: *PLDI*. ACM, 2007, pp. 66–77.
[7] A. Chlipala. “Mostly-automated verification of low-level programs in computational separation logic.” In: *PLDI*. ACM, 2011.
[8] *Companion website*. URL: http://www.irisa.fr/celtique/ext/smc.
[9] P. Cousot and R. Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.” In: *POPL*. ACM, 1977, pp. 238–252.
[10] J. Jensen, N. Benton, and A. Kennedy. “High-Level Separation Logic for Low-Level Code.” In: *POPL*. ACM, 2013.
[11] A. Kennedy et al. “Coq: The world’s best macro assembler?” In: *PPDP*. ACM, 2013, pp. 13–24.
[12] J. Kinder. “Towards static analysis of virtualization-obfuscated binaries.” In: *WCRE*. 2012, pp. 61–70.
[13] G. Klein and T. Nipkow. “A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler.” In: *ACM TOPLAS* 28.4 (2006), pp. 619–695.
[14] G. Morrisett et al. “RockSalt: better, faster, stronger SFI for the x86.” In: *PLDI*. 2012, pp. 395–404.
[15] M. O. Myreen. “Verified just-in-time compiler on x86.” In: *POPL*. ACM, 2010, pp. 107–118.
[16] T. Nipkow. “Abstract Interpretation of Annotated Commands.” In: *Proc. of ITP-12*. Vol. 7406. LNCS. Springer, 2012, pp. 116–132.
[17] V. Robert and X. Leroy. “A Formally-Verified Alias Analysis.” In: *CPP*. Vol. 7679. LNCS. Springer, 2012, pp. 11–26.
[18] G. Stewart, L. Beringer, and A. W. Appel. “Verified heap theorem prover by paramodulation.” In: *ICFP*. ACM, 2012, pp. 3–14.