Static Analysis of x86 Assembly: Certification and Robustness Analysis

Vincent Laporte

Static Analysis of x86 Assembly: Certification and Robustness Analysis

Vincent LAPORTE

June 2011

ÉNS Cachan — Université de Rennes 1

Advisors:

Guillaume Hiet
SSIR — Supélec

Sandrine Blazy
Celtique — Irisa

David Pichardie
Celtique — Irisa

Abstract

Static analysis is a widespread method to gain confidence in the reliability of software. Such analyses are mainly carried out on source code but Reps et al. highlight the relevance of analysing executables because of what they call the WYSINWYX phenomenon. They have therefore designed the value-set analysis and implemented it in CodeSurfer/x86, their tool for analysing x86 executables. However executables may be obfuscated, i.e. made hard to analyse. Although there are specific desobfuscation methods to address the analysis of obfuscated programs, we wonder how robust generic analyses and tools are, and up to which extent they are able to yield sound and precise results. The case of soundness is also addressed from an other point of view: in order to trust an analysis result, there must be strong evidence that the analyser is correct. Formal methods can provide such evidence. In this study we first review obfuscation purposes and techniques, as well as the method of the value-set analysis. Then experiments are carried on to better understand the robustness of analysers. Finally thanks to the Coq proof assistant, a subset of the value-set analysis for a toy assembly-like language is formalized. A certified checker enables to prove the soundness of a result.

Contents

1. Introduction

Static analysis is a widespread method to gain confidence in the reliability of software; it aims at automatically determining some properties about programs without executing them. Such method can be opposed to dynamic analysis which consists in studying programs during their execution. However both methods are complementary and may be used together: this is then hybrid analysis [RM10].

There are many uses for static analysis: for instance, compilers may rely on such techniques for type-checking or performing optimizations. Other tools like Costa [Alb+09] can be used to statically prove that a program always terminates or compute how much resources will be consumed at run-time. Many static analysers rely on abstract interpretation; among them Astrée [Mau04] is able to prove the absence of runtime error in huge critical C programs. With respect to security, malware analysis often uses static methods. Even when dynamic techniques are preferred, a static analysis may be a first step [RM10]. Computer forensics also needs to understand attacks without having them run again.

Most static analyses are carried out on source-code-like representations. Unfortunately and as highlighted by Reps [Bal+08], this is not always suitable and executable analysis would be more adequate: programs should rather be studied in the form which is actually executed by the processor. A first and obvious reason is that source code may be unavailable. It is nothing absurd considering an analysis of some malware or some commercial software distributed in executable form only. In addition to third-party binary components, one could also consider analysing executables even when the source code is available: indeed — and this is the "what you see is not what you execute" (wysinwyx for short) phenomenon [Bal+08] — the source code (what you see) may not provide adequate information to prove properties on the *final* program (what you execute). One may not trust a compiler, which might introduce bugs or, more likely, do some unwanted optimization.

Moreover when analysing source code, all possible choices a compiler can make have to be taken into account; analysing an executable requires considering only the choices that have actually been made. Similarly source analysis relies on some hypotheses relative to external functions, although binary analysis can either do the same hypotheses or actually analyse the library. This shows how analysing executables can yield more accurate results.

Unfortunately analyses that work well on source code cannot be directly applied to executables expecting the same good results. Indeed there is much valuable information which is no longer available after compilation: there are no well identified variables, only a few registers and indirect accesses to memory; there is no type information, only bit-fields; neither control-flow- nor call-graph easily accessible [Rep+10]. Many of these difficulties, which are inherent in analysing binaries are addressed by Reps *et alii* studies and correctly handled [BR10].

However at least two problems remain and are studied in this report. One one hand the static analysis of a binary program may be harder than expected: particularly when source code is not available, programs to be analysed are likely to be obfuscated, that is, roughly, engineered in order to make static analyses fail. Indeed obfuscation is commonly used to achieve intellectual property protection or strengthen a malware. Therefore a first question arise: up to which extent are static analyses robust to obfuscation? Are they still sound? Do they still yield useful results or too large over approximations only?

On the other hand all the methods and techniques involved in analyses as in [BR10] are far from simple and easy. Up to which extent can one be confident in them and in their implementation? When analysing a software to decide its trustworthiness, there should be stronger reasons to trust the analyser than the program being analysed. The theoretical framework introduced in [Pic05] provides some means to acquire a high level of confidence in abstract interpretation based static analyses. They are formalized and proved sound with Coq [Tea10], before a certified implementation can be extracted.

In this work, we attempt to address these two problems. First an experimental approach is used to better understand how are related obfuscating transformations and soundness and accuracy of static analyses. Then formal methods are applied to formalize a subset of the value-set analysis for a toy assembly-like language.

This report is organized as follows. We first introduce in § 2 the field of obfuscating transformations and the static analysis technique we focus on, namely value-set analysis. Then we relate our experimental investigation about static analysis tools facing obfuscated programs in § 3. Next, in § 4, we explain our work about formal certification. Finally we conclude.

2. Related Work

This part is mostly taken from the report of the preliminary study carried out before the internship.

2.1. Obfuscation

Before it is possible to know how robust is some static analysis w.r.t. obfuscating transformations, we first have to understand what obfuscation is, why it is used and how. Mainly following [CTL97; BS05], we first define obfuscation (§ 2.1.1) and explain which properties such transformations are expected to satisfy (§ 2.1.2), then we show how they are build: obscure constructs (§ 2.1.3) are a fundamental piece of many of them (§ 2.1.4); examples show more details of actual techniques (§ 2.1.5). Finally we study in § 2.1.6 some analyses designed to handle obfuscated programs.

2.1.1. Definitions, Purposes

Obfuscating transformations aim at making programs hardly understandable while preserving their behaviours. Such a definition is really vague because there is a wide range of obfuscating transformations. How a program is understandable depends on *who* is reading it and what the reader is looking for: a lay user for its own entertainment, a skilled analyst for security flaws, a disassembler for instruction boundaries or an anti-virus software for viruses have very different purposes that may be all qualified as *understanding*. A human, a static analyser as well as a dynamic analyser may be confused by such transformations; in the context of this study, we mostly focus on techniques likely to deceive static analyses.

Besides what behaviour has to be preserved is also a fuzzy notion: is it the exact sequence of executed instructions? or the program output (from a user point of view) in terms of its input? Some transformations that do not preserve the former are common in optimizing compilers: dead code elimination for instance; thus there is no need for an obfuscator to preserve it. On the other hand mixing the actual computations within noisy ones, may also achieve obfuscation.

There are two main reasons to apply obfuscating transformations to a program: protecting intellectual property and preventing malware from being detected and impeded. A software might be distributed along with some secret embedded in it, for instance the design of an algorithm or data structures. Hence obfuscation provides some means to hide secrets without altering the quality of service of the shipped program. On the other hand, these techniques can be used to make malwares last undetected, or to stealthy hide a malicious part amid a program, etc. Moreover obfuscation techniques are also applied to watermarking: a mark can be embedded into a program using a secret. Knowing this secret the mark can be recovered but ignoring the secret, removing or tampering with the mark amounts to undo the obfuscating transformations which is expected to be hard. Finally obfuscating a program is a cheap way to improve its security making faults harder to discover.

2.1.2. How Good a Transformation Is

In order to better understand what obfuscation is, the properties obfuscating transformations must achieve shall be introduced. These evaluation criteria are mostly taken from [CTL97].

Potency Program understanding is already a complex problem. There are many measures of the complexity of a program: number of instructions or procedures, level of nesting of loops or conditionals, span of variable live-ranges, complexity of data structures or class hierarchy and so on. Provided a numerical

measure of program complexity, the potency of a transformation on a given program is the relative increase of this program complexity when transformed [CTL97]. This numerical measure expresses how a human reader is affected by an obfuscation.

Resilience Crafting a potent transformation can be easy: inserting dead code, redundant conditionals etc. indeed increases the value of some complexity measure. However it is often feasible to identify and eliminate these extra instructions using dedicated tools. Thus in addition to being potent, an obfuscating transformation must be *resilient*, that is reverting the complexity of an obfuscated program to its original value must be hard.

Such a property of program transformation is hardly numerical. Therefore a categorical scale is used: *trivial*, *weak*, *strong*, *full* and *one-way*; this takes into account the complexity class of the deobfuscation problem as well as the scope of the transformation, from *local* to *inter-procedural*. Transformations are labeled one-way when they cannot be undone: for instance the strip utility which discards symbols from object files is a one-way obfuscator; however it is not really potent.

Stealth To prevent a human analyst from guessing which parts of a program are obfuscated, e.g. where are the opaque predicates (see § 2.1.3), and using this knowledge to undo transformations or tune an automatic deobfuscator, transformations must be *stealth* [BS05]. This means that instructions introduced or modified by an obfuscator must *look like* the other ones. This property is unlikely to be measured but is worth being mentioned; it is particularly relevant when only part of the code is actually obfuscated as in [LD03] for instance to limit execution time expenses.

Cost Whichever the reason for obfuscating a program is, obfuscation is not carried out for its own sake: the program still has to be distributed and run. Obfuscating transformations are likely to increase code size, execution time or the runtime memory footprint, which could lessen the value of the program. All those drawbacks amount to the cost of a transformation. Such a property can be expressed in term of the increase of resources required to run the program after it has been obfuscated. Again a categorical scale may be used to distinguish constant, linear, polynomial and exponential increases.

Besides there are other non numerical costs. For instance an obfuscating transformation may break compatibility: as described below, after obfuscation functions may be split or merged, data represented differently, class hierarchy muddled, etc. Such a compatibility break may be unwanted hence some transformations avoided.

In addition to this general metrics, each method can be more specifically evaluated. For instance the Linn and Debray's obfuscator targets the disassembly process [LD03]; hence they measure what they call the *confusion factor*: the proportion of *actual* instructions (or basic blocks, functions or any program *units*) of the program (that is the ones that will be actually encountered when the program is executed) that are incorrectly identified by the disassembler.

Finally it has to be noticed that generally there is some trade-off between these measures: a transformation would have to be more expensive to be more resilient for instance; or cheap methods are sometimes preferred when they are enough to hinder a specific process.

2.1.3. Opaque Predicates

Before investigating general methods used for crafting obfuscating transformations, the concept of *opaque predicates* might be introduced: they are a key element of many obfuscations. Such a predicate is an expression whose value is known by the obfuscator but presumably hard to statically compute. They can then be used to introduce branches in a program that would never be taken and hence lure the reader into considering a control-flow graph much more complex than what actually is. Opaque predicates can be constructed using aliases, parallelism, intricate arithmetic properties etc. in a way controlled at obfuscation time. Indeed computing such a value would require to carry out a precise alias analysis in presence of parallelism or to prove arithmetic theorems.

Collberg *et alii* have described several methods to craft opaque predicates and how to exploit them to achieve obfuscation [CTL98]. One of them works as follow. Provably for all integer y, $7\,y^2 - 1$ is not a

square; nevertheless such property might not be known by a static analyser. Then consider two global variables X and Y and two execution threads updating from time to time those variables in such a way that at any moment X holds a square and Y seven times a square; again such invariant is unlikely to be discovered by a static analyser. Hence expression (Y−1 == X) always evaluates to false yet any static analyser must consider that both truth values can occur.

2.1.4. Obfuscation Methods

There is a tremendous number of different techniques used for obfuscation. It is neither possible nor interesting to enumerate them all; nevertheless it is suitable to know what are the main families and how they hamper analyses. Therefore we present the taxonomy introduced in [CTL97].

Since we are interested in static analysis of x86 executables, there are obfuscating transformations that we do not care about. Indeed some of them operate on source code in such a way that it becomes unreadable but the binary code after compilation is left unchanged. However some transformations that are described below may be applied to executables as well as at the source code level [Mad+06].

Collberg's classification is mainly based on what is transformed: control or data; it also exhibits a class of methods dedicated to hindering reverse engineering techniques and tools. Other classifications are also possible; Balakrishnan and Schulze propose a variation of this one using the target of the obfuscation (disassembler, static analyser, anti-virus…) as main criterion [BS05].

Control Obfuscation Programs are usually organized and structured, to ease programming, debugging or maintaining the program. To break this organization, functions can be inlined or cloned; hence one piece of code would have to be analysed several times decreasing analysis speed and precision. Conversely unrelated units can be gathered in a single one, or relevant computation mixed with dead code.

Opaque predicates enable to extend loop conditions; this can reduce analyses accuracy, since what is known after a loop relies on this loop condition. Similarly spurious edges can be introduced in the control-flow graph thanks to opaque predicates; it can be highly resilient when the resulting CFG is no longer reducible.

Table interpretation consists in storing part of the code as data in some language and embedding an interpreter for this language inside the program. It is expected to be very potent and resilient but at a hardly affordable cost.

Data Obfuscation This kind of obfuscation deals with the way values are represented and stored: a variable may be split so that on every read several values have to be combined again; conversely several variables may be stored into a same location at the same time. The layout of data inside arrays or structures can also be altered as well as, in the case of object-oriented languages, the class hierarchy. Another powerful obfuscation consists in converting static values (i.e. that are known at compile time) into dynamic ones (i.e. computed at run-time). Such techniques have no effect on dynamic analysis. All those methods are likely to be expensive.

It can be noticed that most techniques from both above categories break in some way the *good* software engineering practices.

Preventive Obfuscation This last category in Collberg's classification includes techniques that are more targeted at some reverse engineering process; they are rather resilient than potent or "boost" other transformations. For instance after a loop transformation is applied, spurious dependencies among the loop iterations can be introduced to prevent recovering the original loop.

The general packing techniques can also be seen as obfuscating transformations: when executed, a packed software first unpacks its actual code so as to run it; this enables to compress or encrypt a program for instance. Similarly analysing a packed program requires unpacking it first. Those techniques, that are a kind of self-modifying code, are likely to be used by malwares. Viruses for instance, that are usually detected through their signature, may be packed; when they copy themselves, polymorphic viruses change the packing method (ciphering algorithm, etc.) hence their signature.

2.1.5. Examples

In the following three examples of obfuscation methods are described. They show more concrete aspects of the field and how actual techniques hinder static analyses.

Flattening

Wang *et alii* present a control-flow transformation aiming at making accurate intra-procedural analyses intractable [Wan+00]. The main idea is to introduce a branching block and make execution jump to it after every basic block. Then the branching block is responsible to route execution. This is achieved using a `C` switch and the switch variable being set accordingly in every block.

Then any basic block appears to be a possible successor of any one, and discovering which are the real edges of the control-flow graph requires that the value of the switch variable is accurately tracked. This is prevented by an extensive use of pointers and aliases to access it.

Junk Bytes

Linn and Debray's preventive method targets the disassembly process [LD03]. The key idea is the introduction of *junk bytes* in the code at locations that will not be reached at run-time, for instance after an unconditional jump. When a disassembler will try to decode those bytes, it will get confused and misbehave on reachable parts of the code, i.e. provide wrong information on relevant parts of the program.

But confusion will not last more than a few bytes; they call this phenomenon self-repairing disassembly. Therefore to raise confusion, the number of places where junk bytes can be inserted is increased *via* branch flipping. This converts a conditional branch (`jc L; M:` *next*) into its negation followed by an unconditional branch (`jnc M; jmp L; M:` *next*). Hence junk bytes can be inserted after any jump.

Moreover some disassemblers do not try to decode instructions after an unconditional jump but traverse the program following the control flow and decode from the target of the jump. To confuse them as well, opaque predicates may be used to introduce fake conditional jumps into junk code. The authors have rather chosen to change jumps into calls to one *branch function*. Such a function is responsible for routing the control flow towards the jump target. Notice that junk bytes can still be inserted just after the call instruction since control flow does not continue after this call when the branch function returns.

Signals and Exceptions

Popov, Debray, and Andrews further obfuscate the control flow making it rely on unusual jumping methods [PDA07]. Indeed the branch function — as described above — is no longer called by normal `call` instruction but as a signal handler. Signals here considered are the ones raised due to hardware exceptions, such as unauthorized memory accesses or division by zero. Such exceptions are unlikely to be suspected by a static analyser. Branch instructions are replaced by ones that will induce such a signal. When the signal handler is executed, it computes where to resume execution depending on where the exception occurred.

2.1.6. Understanding Obfuscated Programs

Here we focus on techniques whose aims are to circumvent or reverse obfuscations, as opposed to section 2.2 where more general analysis techniques will be investigated. Our purpose is not to show how to steal intellectual property secrets but rather which are the limits of obfuscation and how strong such a protection is. Besides we expect to analyse executables that are potentially malicious hence likely to be obfuscated.

Kruegel *et alii* [Kru+04] focus on the obfuscations studied by Linn and Debray [LD03] and provide means to disassemble an obfuscated executable, i.e. to discover where instructions are. They proceed by reconstructing the intra-procedural control-flow graph of the program, hence one function at a time; thus functions are first identified, thanks to their characteristic prologue[1].

[1] `55 89 e5`, byte sequence corresponding on x86 architectures to the instruction sequence: `push %ebp; mov %esp, %ebp`

Any address might be the beginning of an instruction, therefore a recursive disassembly process is initiated at every address inside a function. This yields a set of basic blocks of the function. However since disassembly is performed from every address, many blocks may *conflict*, that is span over overlapping intervals of addresses. Those conflicts are then solved: first soundly, then using heuristics based on the usual distribution of instructions in executables, finally at random.

Despite the questionable heuristic of this disassembler, results are quite good. This highlights the following phenomenon: cheap techniques are enough to obstruct existing tools (Linn and Debray have noticed high confusion factors); these techniques again are likely to be bypassed by other cheap ones, and so on. Indeed disguising unconditional jumps as conditional ones with opaque predicates is enough to make inefficient Kruegel's disassembler (as they themselves say).

Udupa *et alii* [UDM05] discuss how to reverses Wang's flattening obfuscation; this amounts to recover the original control flow graph of an obfuscated program. Since their results have been evaluated using an obfuscator and a deobfuscator they had crafted themselves, the generality of their results is questionable. Nevertheless two static techniques are shown to be working.

To improve the accuracy of the static analysis, it is combined with dynamic analysis: running the program, a initial set of edges that can be taken is discovered. Doing this the analysis is no longer sound: some edges can be declared *spurious* albeit they are not; however the results are much more precise: less spurious edges are considered as belonging to the original program.

Guillot and Gazet address deobfuscation of malware which use table interpretation [GG09]. They succeed in this thanks to the framework Metasm[2], an *interactive* tool for static and dynamic analysis of executables.

There are plenty of obfuscation techniques and many target or at least confuse static analyses. However it was shown that specialized static analysis is still possible and hopefully feasible thanks to the necessary trade-off between cost and power of these obfuscating transformations. Besides dynamic or interactive methods may improve analysis accuracy.

In the following, rather than focussing on desobfuscation techniques, we will study one generic static analysis, designed to be applied to programs under their executable form. We then intend to study how robust is such an analysis to obfuscation, more precisely: 1. which hypotheses are assumed by the authors of the analysis might be not satisfied by obfuscated programs? 2. up to which extent can such an analysis be used for the study of obfuscated code? 3. how to formally certify an implementation of this analysis or the soundness of its result? Next section is therefore dedicated to the state of the art general purpose static analysis of binary programs.

2.2. Static Analysis of Executables

Several reasons, from the unavailability of source code to the WYSINWYX phenomenon, stress that analysing programs through their executable form rather than their source code is of much interest. However in order to study an executable, some high-level information has to be previously recovered; the control-flow graph (CFG) for instance. Accurate CFG reconstruction is the purpose of JAKSTAB [KV10] and of Bardin *et alii* [BPF11]. Balakrishnan and Reps [BR10] additionally address variable recovery and reconstruction of other intermediate representations. These abstractions can then be used to carry out more analyses: Reps *et alii* propose to build a *pushdown system* for model checking and answering reachability queries for instance [Bal+08], or to further *decompilation* (also mentioned in [KV10]). We here focus on Balakrishnan and Reps' value-set analysis (VSA for short). It is based on abstract interpretation [CC77], a mathematical theory of approximation of program semantics (a more recent synthesis may be found in [CC04]).

The aim of VSA is to compute an over-approximation of the set of values that every *location* may hold. Those locations are abstract representations of the concrete memory cells and registers, as described in § 2.2.1. This set of values is to be computed at any point in the program, that is for any instruction in every possible context (*context-sensitivity* is detailed in § 2.2.2).

[2] The Metasm assembly manipulation suite, http://metasm.cr0.org/

It is the result of several years of research thus cannot be described here in its slightest detail. It is mainly described in [BR10] and details are further developed in several articles from the same authors. Notice that it considers only x86 executables.

We first describe the abstract world: how memory (§ 2.2.1) and values (§ 2.2.2) are approximated; then we detail how the analysis relies on IDA Pro or any external disassembler (§ 2.2.3); finally some of the insights of VSA are introduced through an example crafted for this study (§ 2.2.4).

2.2.1. Memory Model

A common and simple view of the memory is a large byte array. However this model is not suitable for static analyses w.r.t. dynamically allocated memory and runtime stack. Indeed a call to the malloc function may return any address so that writing to this address could alter any byte of the memory, which is unlikely to yield a precise analysis. It is much more convenient to abstract this address as a value denoting "the address returned by *that* call to malloc" and then use offsets relatively to this abstract value. To address those problems the following abstraction is introduced.

Memory is split into disjoint abstract regions in which addresses (namely offsets) represent a set of concrete address. Regions are of three kinds: 1. the *global* region which holds absolute addresses as well as numeric values; 2. one *AR* region per procedure (AR stands for Activation Record, a.k.a. stack frame); the origin of such a region corresponds to the concrete address pointed to by the stack pointer when the procedure is called; and 3. one *malloc* region per "malloc site"; such a site is an instruction of the program listing: even if this instruction is executed several times and hence corresponds to several concrete memory locations, there is only one site.

There is no assumption about the relative positions or the concrete layout of these regions in memory. However during inter-procedural VSA, the initial content of an AR region, i.e. the local variables and actual parameters of a called procedure, are computed after the current content of the caller's AR region.

Every region is considered to be $4\,\mathrm{GiB}$ large: offsets range from -2^{31} to $2^{31}-1$ and address each byte of the region. Not all these bytes are of interest: only few of them are actually accessed. Besides they are usually not accessed one by one: bunches of successive bytes are likely to represent one single entity, a program variable for instance. These ranges of bytes that are accessed simultaneously (through one instruction) are called *a-loc*s (for *abstract locations*); they are slices of a region.

The space spanning above the topmost *a-loc* in an AR region as well as the one below the bottommost one are considered as *a-loc*s too; named FormalGuard and LocalGuard respectively, they allow to detect memory accesses outside the bounds of an activation record. Similarly the size of each malloc region is tracked (represented as a strided interval, see § 2.2.2 below) to report potential memory-access violation.

Additionally each register of the instruction set architecture is considered as an *a-loc*.

2.2.2. Abstract Domains

A key element of abstract interpretation is how *things* are abstracted. In the following we describe which values are considered and how they are approximated.

Strided Intervals

Values that we care about are addresses: to accurately analyse memory reads and writes, indirect jumps, indirect calls, etc. addresses must be known. In executables pointer arithmetic is ubiquitous and any integer may be used as an address: there is no way to distinguish integers from addresses. Therefore VSA aims at computing the set of integers any location may hold. There is only a finite number of integers that can be stored in a location (2^{32} for 32-bit locations) hence a finite number of sets of such integers ($2^{2^{32}}$, which is a lot); however those sets are not tracked directly but through an abstraction which is therefore an approximation.

This abstraction, the *k-bit strided interval* [RBL06], represents an interval, i.e. a set of integers with a lower and an upper bound, in which any point is congruent to the lower bound modulo the stride; in addition these integers have to be in the interval $[-2^k; 2^k-1]$. Formally the strided interval with lower bound l, upper bound u and stride s, written $s\,[l;u]$, represents the set $\{n \in \mathbf{Z} \mid l \leqslant n \leqslant u \wedge n \equiv l \mod s\}$, provided $-2^k \leqslant l \leqslant u \leqslant 2^k - 1$; the empty set is canonically denoted by $\bot$.

A strided interval correctly captures the set of addresses of the elements of an array. However this approximation is poor at representing accurately sets of jump targets [BPF11].

Strided intervals are (partially) ordered by the inclusion relation on the sets they represent. A little arithmetic enables to soundly carry out joins, meets, as well as numeric and bitwise operations on strided intervals. Those operations are performed at a bit level so as to accurately model them as they are carried out on a microprocessor (taking care of overflows for instance).

Processor Flags

Several instructions behave depending on the values of some flags: conditional jumps and moves, some arithmetic instructions, etc. Therefore it can be valuable to accurately track the state of those flags. The analysis only focuses on a reduced number of the flags: overflow (OF), carry (CF and AF), zero (ZF), sign (SF) and parity (PF). The direction flag (DF) is ignored for undocumented reasons. Arithmetic operations on strided intervals take into account those flags and update them accordingly. A three-valued logic is used to ensure the soundness of the approximation. This domain $\mathsf{Bool3} = \{\textsc{True}, \textsc{False}, \textsc{Maybe}\}$ is equipped with join and usual boolean operators in a straightforward way.

Value Sets

A strided interval represents a set of integers; an integer can always be seen as an offset in some memory region. Thus an integer alone is not meaningful enough, is has to be associated with the region in which it can be correctly interpreted. Hence the following domain is introduced: to abstract a set of numeric values or addresses, a *value-set* is a map from memory regions to strided intervals.

It is sometimes possible to do some arithmetic with value-sets, particularly when they represent numeric values (i.e. in the global region only). In such cases it amounts to performing the corresponding operation on strided intervals; in the other cases however, it is not possible to tell anything. Consider for instance the addition of an offset in some AR region with an offset in *another* AR region. Fortunately such operations are unlikely to happen.

Formally value-sets are as follow.

$$\mathsf{ValueSet} = \mathsf{MemRgn} \to \mathsf{StridedInterval}$$

with 1. MemRgn being a set with one element representing the global region, one for each procedure, one for each malloc site, and no more; 2. StridedInterval being the complete lattice of strided intervals.

Abstract Environments

This context-sensitive abstract interpretation tracks, for each possible context, the set of values every abstract location may hold. There are two kinds of locations: 1. the *a-loc*s that can hold integers represented by value-sets; and 2. the flags whose possible values are abstracted by the Bool3 domain. Context is modeled by the *call-string*. Starting from the program entry-point, the call-string at a given point is the sequence of call instructions the program went through to reach the considered point. Such a potentially infinite domain has to be bounded: only the suffix of length k of the call-string[3] is taken as context. The larger k, the more accurate the analysis; but also more expensive. Taking $k = 0$ corresponds to the context-*insensitive* analysis. Moreover only call-strings that may occur are investigated (hence the $\rightharpoonup$ below, denoting partial map).

The knowledge acquired by the abstract interpretation is gathered in an abstract environment whose type follows.

$$\mathsf{CallString}_k \rightharpoonup (\textit{a-loc} \to \mathsf{ValueSet}) \times (\mathsf{Flag} \to \mathsf{Bool3})$$

$$\text{with} \qquad \mathsf{Flag} = \{\mathrm{CF}, \mathrm{ZF}, \mathrm{SF}, \mathrm{PF}, \mathrm{AF}, \mathrm{OF}\} \qquad \mathsf{Bool3} = \{\textsc{True}, \textsc{False}, \textsc{Maybe}\}$$

The VSA computes for each program point such an abstract environment.

[3] or the full string if its length is less than k

```
1  push   %ebp                      ;                          ; esp ↦ (⊥, −4)
2  mov    $0x80483b4,%ecx           ; ecx = a                  ; ecx ↦ (a, ⊥)
3  mov    %esp,%ebp                 ;                          ; ebp ↦ (⊥, −4)
4  sub    $0x8,%esp                 ;                          ; esp ↦ (⊥, −12)
5  mov    0x8(%ebp),%edx            ; edx = arg_0              ; edx ↦ (⊤, ⊤)
6  mov    0xc(%ebp),%eax            ; eax = arg_4              ; eax ↦ (⊤, ⊤)
7  movl   $0x80483b4,(%esp)         ; var_8 = a                ; var_8 ↦ (a, ⊥)
8  movl   $0x80483bb,0x4(%esp)      ; var_4 = b                ; var_4 ↦ (b, ⊥)
9  mov    −0x4(%eax,%edx,4),%eax    ; eax = arg_4[edx−1]       ; eax ↦ (⊤, ⊤)
10 xor    %edx,%edx                 ; edx = 0                  ; edx ↦ (0, ⊥)
11 mov    (%eax),%eax               ; eax = eax[0]             ; eax ↦ (⊤, ⊤)
12 test   $0x1,%eax                 ; if eax is even           ;
13 je     22                        ; then jump                ;
14 mov    $0x1,%ecx                 ; ecx = 1                  ; ecx ↦ (1, ⊥)
15 mov    %ecx,%ebx                 ; ebx = ecx                ; ebx ↦ (1, ⊥)
16 sar    %eax                      ; eax = eax / 2            ; eax ↦ (⊤, ⊤)
17 sub    %edx,%ebx                 ; ebx = ebx − edx          ; ebx ↦ (1 [0; 1] , ⊥)
18 test   $0x1,%eax                 ; if eax is odd            ;
19 mov    %ebx,%edx                 ; edx = ebx                ; edx ↦ (1 [0; 1] , ⊥)
20 jne    15                        ; then jump                ;
21 mov    (%esp,%ebx,4),%ecx        ; ecx = var_8[ebx]         ; ecx ↦ ((b − a) [a; b] , ⊥)
22 call   *%ecx                     ; (ecx)()                  ;
23 add    $0x8,%esp                 ;                          ;
24 mov    %ebp,%esp                 ;                          ;
25 pop    %ebp                      ;                          ;
26 ret                              ;                          ;
```

Figure 2.2.: Assembly program with an indirect call (left), C-like translation (middle) and some bindings computed by VSA (right)

2.2.3. Disassembly

An executable is nothing else than a sequence of bytes. Hence the first step of any analysis is disassembly, that is parsing this raw data, finding where instructions are and decoding them (identifying the operation and operands), and building a control-flow graph (finding jump/call sites and targets). In CodeSurfer/x86 [Bal+08], a tool implementing VSA, this work is performed by IDA Pro. This disassembly toolkit also provides an initial set of *a-loc*s, call and malloc sites.

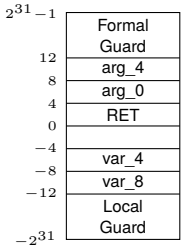

Figure 2.1.: alocs

In order to better understand how the analysis works, an example shall be considered. The raw bytes of the example executable are not shown but rather given to IDA Pro which recognized, among others, the procedure whose assembly code is shown on the left of figure 2.2. Besides were identified four local variables arg_0, arg_4, var_4 and var_8; this enables to get an initial picture of the activation record of this procedure, that is which are the abstract locations and where they are (see figure 2.1 aside, depicting the static *a-loc* layout in the AR region). For instance the location named var_8 by IDA Pro ranges from offset -12 to offset -8. Notice that an extra *a-loc* is added at offset zero: it is supposed to hold the return address, i.e. where to jump after the end of the function. FormalGuard and LocalGuard are as described in § 2.2.1.

Unfortunately IDA Pro can fail at recovering correct locations. This does not impact the soundness of the analysis but could alter its accuracy.

A partial call-graph is also recovered: here, because of indirect calls, this procedure is an isolated node in this graph. There are no edges to or from it. We will see in the following how analysing from the beginning of this procedure enables to correctly recover the possible targets of the indirect call.

2.2.4. Value-Set Analysis

After disassembly and *a-loc*s identification, VSA can take place from the program entry-point discovered by the disassembler, following the control-flow graph of the program: nodes correspond to instruction addresses and edges are labelled with the semantics of instructions. Graph exploration is done maintaining a work-set [NNH99], i.e. the set of instructions left to analyse. It is initially set to the program entry-point.

Let's see how it works on the example program depicted figure 2.2; left column is the output of disassembly in AT&T assembly language syntax,[4] middle one is a C-like equivalent to ease reader understanding and right one shows some new knowledge recovered by VSA as explained below. The example program illustrates an indirect call; which function to call is computed after the value of the program arguments. It works as follows: the addresses of two functions are stored on the stack (lines 7 and 8); then *some* computation is carried out in a loop (lines 15 to 20); finally depending on how many times the loop was executed, one of the two addresses is read and the corresponding function indirectly called (line 22).

Since there is only one procedure and no malloc site, two memory regions have to be considered: 1. the global one and 2. the AR one corresponding to the procedure activation record. Therefore value-sets may be represented as pairs with one component per memory region, as in $(0, \bot)$ representing the numeric constant zero[5] and no possible offset in the AR region.

Initially for the empty call-string, every abstract location may hold any value; this (absence of) knowledge is represented by binding each *a-loc* to the *top* value-set — i.e. $(\top, \top)$ with $\top = 1\left[-2^{31}; 2^{31}-1\right]$ being the strided interval denoting the set of all possible integers — and each flag to the MAYBE value. There is one exception however: the stack pointer esp is known to hold the address of the current stack frame, hence by definition of the AR memory region, this register is bound to $(\bot, 0)$.

Updating Memory Line 2 a constant (call it a) is stored into register ecx, therefore knowledge about the values that ecx may hold can be refined to value-set $(a, \bot)$. This register indeed has one possible value, which is numerical hence belonging to the global region; besides it is not an offset in the AR region, expressed by the second component set to $\bot$. Considering that a numeric value cannot express an address inside the activation record may seem unsound; indeed it may happen, by chance or on purpose, that such a value is some actual address. However this is correctly taken into account when such a value-set is used as an address: an access to it has to be considered as an access to any part of any *a-loc*. Notice that those cases, which would lead to a tremendous loss of accuracy, are unlikely to happen in executables produced by compilers.

A bit further, on line 7, the same constant is stored into memory through the pointer esp. Transforming the abstract environment is done in two steps. First the set of *a-loc*s that may be accessed via esp is computed from the value-set currently bound to this register, namely $(\bot, -12)$; using the *a-loc* layout (see figure 2.1 page 12), one can deduce that this instruction writes exactly one *a-loc*, the one earlier called var_8. Now that the *a-loc* to be modified is identified, there are two ways in which the value bound to it can be updated: either *weak* update or *strong* update. Indeed one *a-loc* may represent at the same time several concrete addresses, when it belongs to the activation record of a recursive procedure (one says this region is a *summary* one). In such a case, writing to one of these concrete locations does not change the values hold by the other locations; therefore the newly written value has to be joined with the one previously bound to the abstract location: this is a weak update. Conversely a strong update replaces the value bound to an *a-loc* (the old one is overwritten). Whether a region is summary or not is approximated looking at the call-string and the call-graph; here since the call-string is empty, the region is non-summary and a strong update is carried out.

Idioms The purpose of line 10, where the bitwise xor of the value of register edx with itself is computed, is not to compute a xor, but to set that register to zero. Replacing a more obvious mov by such a xor is supposed to shorten the binary code[6] as well as the execution time[7]. There are several such tricks in x86 assembly.

[4] General instruction syntax is: mnemonic source, destination

[5] This number is actually represented as the strided interval $0\,[0; 0]$, but written 0 to keep simple.

[6] Two bytes for xor (31 d2), five for mov (ba 00 00 00 00).

[7] This is less obvious on modern architectures.

Taking those widespread idioms into account obviously increases the precision of the analysis: a naive xor between unknown values would have yielded a still unknown value, although the exact result can be computed.

Conditional Jumps Eventually a conditional jump arises on line 13. The branch to be taken depends on the value of the ZF flag. Since it is currently bound to MAYBE, both branches have to be considered. More generally conditional jumps only take into account the state of some flags. However when analysing one of the two branches, so as to carry out an accurate analysis and recover a higher level predicate used as jumping condition, the instruction that set the flag must be investigated. Here it is the test instruction on the previous line; therefore the edge corresponding to the jump is labeled with the predicate eax $\& \ 1 = 0$ and the fall-through one with predicate eax $\& \ 1 \neq 0$ ($\&$ denoting the bitwise "and" operation). Such information is more likely to be used to refine the knowledge about *a-loc*s than information about flags only.

The instruction which sets the flag values used in a conditional jump is not immediately available. For instance line 20, the test occurs two instructions before branching. This kind of situation often happens since it enables scattering instructions that use a common register. One could also imagine that a merge point lies in between so that one jump instruction corresponds to different high-level predicates depending on which branch the execution flow comes from. This might happen in obfuscated executables only.

Loops Let's consider first the fall-through branch of the conditional jump of line 13. Execution flow is here entering a loop, at the beginning of which ecx is bound to numerical constant one and edx to zero. After executing loop's body, ecx still holds 1 but edx is bound to value-set $(1, \bot)$: indeed line 17 register ebx is updated through arithmetic on value-sets, before it is copied into register edx. Therefore when the conditional jump at the end of the loop (line 20) is explored, both its successors are added to the work-set: the fall-through one since it is new; the beginning of the loop as well since this loop has to be explored in a new environment. Depending on where execution flow comes from, some *a-loc*s may hold different values; this is taken into account merging the environment at the end of the loop with the one before entering it. In this merged environment register edx is bound to $(1\,[0;1]\,, \bot)$, the first component being the result of joining the two strided intervals $0\,[0;0]$ and $0\,[1;1]$; it denotes the set $\{0, 1\}$. Therefore the loop is traversed an other time, merging fresh knowledge with the one previously gained.

Eventually when nothing new is discovered — i.e. when a fix-point is reached — no nodes are added to the work-set and analysis goes on after the loop.

In order to decrease the time needed to compute a fix-point, widening techniques are used. They are standard to abstract interpretation and not described here.

Indirect Jumps and Calls After exiting the loop, one of the two *a-loc*s var_8 or var_4 is read depending on the actual value hold by register ebx (either 0 or 1, as discovered analysing the loop). Hence register ecx has to be bound to the result of joining the value-sets bound to both *a-loc*s: $((b - a)\,[a; b]\,, \bot)$. Then line 22 is another merge point: environments corresponding to different execution paths are joined (nothing new is learned).

Next, two new possible targets of the indirect call are discovered: namely the a and b addresses. Those locations are not explored now: current analysis is completed before the two new edges are added to the call-graph and analysis started from the beginning again. Notice that if register ecx were bound to the *top* value-set, that indirect call could jump to *any* place in the code and the analysis would not be tractable any longer. Hence in such cases no edge is added to the CFG and the user is warned that the analysis is unsound.

In this section we have shown the main parts of VSA; many details and improvements have not been mentioned however. Some cases for low precision have been underlined; they are potential targets for obfuscations. Besides this analysis is known to be unsound in some cases, for instance when the code modifies itself; even though the analysis detects these cases and warn the user, having a program with the right property prevents it from being analysed. Again this weakness could be exploited by specific obfuscators.

After this review of obfuscation techniques and of a state-of-the-art static analysis for binary programs, we further investigate the actual behaviour of static analyses when carried on obfuscated binaries.

3. Experimental Robustness Study

The previous review of obfuscating transformations and description of the state of the art in generic static analysis of binary programs have shown some gaps between the assumptions that are made about analysed programs and what these programs actually may be. For instance an analyser expects the analysed program to be correctly disassembled and many obfuscation techniques are dedicated to hampering disassembly: analysing a wrong assembly listing might yield results that do not match the actual behaviour of the program intended to be analysed. Additionally programs to be analysed are supposed to follow a standard compilation model although obfuscation often breaks common software engineering practices.

Indeed one purpose of obfuscation is to *protect* programs against analysis and many papers previously cited do prove the effectiveness of these transformations. On the other hand it should not be forgotten that for each obfuscating transformation there may be a dedicated desobfuscation method. It is then unclear up to which extent complex static analyses like VSA and tools based on them are effective when facing obfuscated binaries.

Therefore the following points are to be experimentally investigated: how is VSA affected by obfuscation? Can it still yield sound and accurate results when analysing obfuscated binaries? How does this abstract-interpretation based analysis compares to other analysis methods? What could be a practical use for VSA in the field of obfuscated executable analysis?

3.1. Method

So as to carry on this experimental study, we have to be able to both obfuscate and analyse programs. This section describes the tools to use, the programs to analyse and how.

3.1.1. Obfuscating

The main object of this study is static analysis rather than obfuscating transformations. However to achieve our experiments, obfuscators are needed. They have to meet at least the following requirements. An obfuscating tool should be *available*: we do not plan to implement a custom obfuscator. Besides we expect to *understand* what kind of transformations are carried on. Finally there are obfuscating transformations that are known to be too *powerful* for the static analyses here considered: packing techniques or more generally the ones relying on self-modifying code and those using virtual machines / table interpretation. Hence obfuscators providing these transformations are discarded.

The reader of [Coo+09] may argue that value-set analysis is actually used to statically unpack binaries. However in that case VSA is applied to the unpacking program — which is supposed not to be obfuscated — rather than to the packed one.

Among the tools that satisfy those requirements, many are dedicated to transforming (CIL or Java) byte code; they are discarded as well.

On the other hand two tools seem suited for our experiments.

objobf is a mild "x86/linux ELF object obfuscator" which provides a few obfuscating transformations that can be applied between compilation and linking. These transformations are: 1. basic blocks shuffling across the whole file; 2. dead code insertion; and 3. basic block splitting.

Loco Diablo is a binary rewriting framework and a static linker, intended to be a link-time optimiser; its extension Loco provides obfuscating transformations [MVD06]. The most interesting one is intra-procedural flattening. Indeed the other documented transformations either target something else than static analysers — junk code insertion adds bytes after unconditional jumps, *i.e.* at unreachable places, and is expected to disturb linear sweep disassembly — or does not work — inter-procedural flattening or *branch function* sound promising but available implementations are unsound.

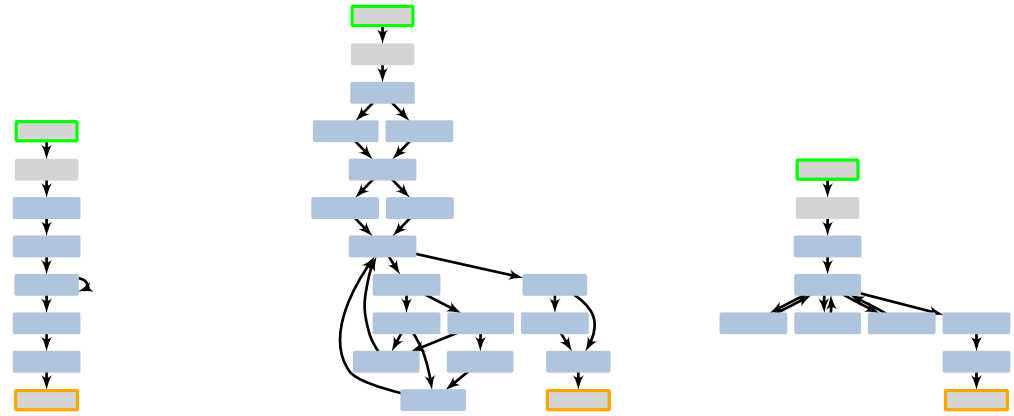

Figure 3.1.: Control-Flow Obfuscation

Link-time rewriting requires much more information than what is usually available at the end of the compilation. Hence a special compilation tool chain is provided and for these technical reasons it is not possible to use both tools to obfuscate the same program copy.

Then it is possible to realistically obfuscate programs that we ourselves write. An example is depicted on figure 3.1; a brief program with one loop in a function, whose control flow graph is rendered on the left, can be obfuscated, for instance, in the two following ways: each basic block is split (middle) or each function is flattened (right).

3.1.2. Analysing

As for obfuscation, there are many tools for static study of binary programs. We focus on two families of such tools. On one hand are state-of-the-art operational tools such as Metasm, Radare or IDA Pro, that are actually used in security domain. They are basically a scriptable disassembler enhanced by many dynamic and static analysis features. On the other hand some research projects provide more complex analyses. Among them CodeSurfer/x86, Jakstab and BitBlaze claim to provide value-set analysis.

Those two kinds of tools have different purposes, constraints and approaches. Hence one tool is chosen in each family for the experimentations, so as to study both and compare them:

Metasm This already mentioned tool suite is essentially a Ruby library suited for binary reading, parsing, disassembly, etc. An interactive user interface may be used to disassemble an executable and display the computed listing as a control flow graph. Among interesting analyses provided by this library is dead code elimination.

Jakstab This tool mostly due to J. Kinder provides program disassembly and static analyses [KV10]. The *configurable program analysis* approach it uses is based on abstract interpretation and various abstract domains can be used, at the same time or independently: bounded address tracking exactly tracks a *finite* set of values; value-set analysis (VSA) has already been introduced. These analyses are path sensitive.

This implementation of VSA appeared to be much more complete than BitBlaze and usable for our experiments, being free software unlike CodeSurfer/x86.

Those tools are able to work with many binary file formats. Nevertheless in the following only ELF/x86 will be used; this is indeed what the chosen obfuscators are able to produce. Such an ELF file is a complex object from which code, data, and entry point have to be retrieved. This is not trivial and utilities *à la* `sstrip`[1] hamper this retrieval. Indeed this tool is said to remove all unnecessary bytes from an ELF

[1] http://www.muppetlabs.com/~breadbox/software/elfkickers.html

file. A `sstrip`ed file is still correctly executable but has no sections any longer: notice that `objdump` for instance is not able to tell anything relevant about such a sectionless program.

3.1.3. Protocol

Basically an experiment is carried on as follows: 1. a source program is considered; 2. two executables are built from this code: an obfuscated version and a control one; and 3. both programs are statically analysed and the analysis results compared. We then discuss which programs are considered and how the analysis results are interpreted.

Sample Programs

So as to be able to understand accurately what, in obfuscation, affects the results of analyses, the programs on which the experiment is carried must be short and simple. Indeed analysis behaviours and results are manually scrutinized.

During the experiment process, only source code is controlled. Indeed we only have limited power on the obfuscating transformation, despite the many parameters and options obfuscators provide. A major limitation of Loco is that only *static* linking is available. Hence we are tied to study programs which do not use libraries that we will not be able to analyse: `libc`, system calls and so on.

Besides to better control the obfuscated code submitted to static analysers, hand-made samples are to be used. For example the hand-written assembly listing shown on figure 3.2 is close to what Loco actually produces. It computes the sum of all natural numbers less than n, an input parameter, and is made of the following *basic blocks*:

`init` checks if the input value is actually provided;

`begin` retrieves this value and sets up local variables including a counter;

`test` checks if the counter is greater than n;

`comp` updates the sum and counter;

`end` returns the computed value and exits.

This program is given a *flat* control flow: a jump table holds the addresses of the above basic blocks and a `jump` block routes the control flow depending on the value held by register `eax` (`.Ljump` is the address of the jump table). Hence each basic block is in charge of setting register `eax` with the appropriate value (index of the next block to execute in the jump table). Jumps are thus replaced by moves, and conditional jumps by conditional moves.

The shown listing features `.p2align` directives which ensure that the addresses of the basic blocs are multiples of 2^5: this pattern does not match actual obfuscated programs but enables to explain some results about strided intervals, the abstract domain of value-set analysis described in § 2.2.2.

Checking After Obfuscation

Our experiments have shown that obfuscators may be incorrect, *i.e.* for some input, the resulting program does not behave as the original one. For instance it has not been possible to get any working program when asking Loco to perform inter-procedural flattening. Therefore the output of an obfuscator has to be checked. The only way to do this is to *test* the resulting program and compare actual and expected behaviours. Any other verification method would amount to desobfuscation.

On the recovery of the address of the `main` function

Linux `libc`-linked binaries have a `main` function, which corresponds to the program. However the entry point of the binary does not match the beginning of this function. Indeed a `__libc_start_main` function is responsible for initializing the environment and calling the `main` function (afterwards) and handling its return value (if any).

```
1   .section  .rodata
2 .Ljump:
3   .long   .init
4   .long   .begin
5   .long   .end
6   .long   .test
7   .long   .comp
8
9   .text
10 .globl main
11   .type   main, @function
12 main:
13   pushl   %ebp
14   movl    %esp, %ebp
15   movl    $0, %eax
16   push    $-1
17   pushl   $0
18   pushl   $0
19 .jump:
20   jmp     *.Ljump(,%eax,4)
21   .p2align  5,0x90
22 .init:
23   movl    8(%ebp), %eax
24   cmpl    $1, %eax
25   movl    $2, %ebx
26   movl    $1, %eax
27   cmovlel %ebx, %eax
28   jmp     .jump
29   .p2align  5,0x90
30 .begin:
31   xorl    %eax, %eax
32   movl    %eax, -4(%ebp)
33   movl    0xc(%ebp), %eax
34   addl    $4, %eax
35   movl    (%eax), %eax
36   movzbl  (%eax), %eax
37   movsbl  %al, %eax
38   subl    $0x30, %eax
39   movl    %eax, -0x8(%ebp)
40   movl    $0, -0xc(%ebp)
41   movl    $3, %eax
42   jmp     .jump
43   .p2align  5,0x90
44 .test:
45   movl    -0xc(%ebp), %eax
46   movl    -0x8(%ebp), %ebx
47   cmpl    %ebx, %eax
48   movl    $2, %ebx
49   movl    $4, %eax
50   cmovgl  %ebx, %eax
51   jmp     .jump
52   .p2align  5,0x90
53 .comp:
54   movl    -0xc(%ebp), %eax
55   addl    %eax, -4(%ebp)
56   incl    %eax
57   movl    %eax, -0xc(%ebp)
58   movl    $3, %eax
59   jmp     .jump
60   .p2align  5,0x90
61 .end:
62   movl    -4(%ebp), %eax
63   leave
64   ret
65   .size main, .-main
```

Figure 3.2.: Sample flat program

Analysing standard (and somehow complex) code for each analysed binary is not so useful and leads to a loss of precision. Therefore it seems cleverer to start the analysis at the beginning of the `main` function rather than at the binary entry point.

These binaries all begin with the same start-code which sets up a stack and call `__libc_start_main`. Hence provided the analysed binary conforms to this rule, the address of the `main` function can be found twenty-four bytes after the entry-point; this heuristic seems really weak but turned out to be quite effective.

Interpreting Analysis Results

Analyses yield various kinds of results with at least an assembly listing expected to match what has been analysed. Automatically checking that this is correct is at least hard: there is no reference listing to which compare the output one, even though we have control over source code. Indeed in case of an automatically obfuscated program, matching the output against the source amounts to desobfuscation. And if the obscure listing is known (as the one from figure 3.2), comparison would be possible after syntax normalization only. Therefore small programs are preferred so as to enable manual verification.

When analysing binaries, a main problem after disassembly is recovering the control flow of the program. Locally for a given instruction (or basic block) it amounts to compute what are its possible successors. This is specially relevant in case of indirect jumps or calls, that is when execution flows towards an instruction whose address is in a register. There is a much more complex case that we purposely ignore: when external events drive the control flow, like signals as described in [PDA07].

Jakstab builds the whole control-flow graph of the program fragment it analyses and is able to tell

whether it is correct (sound over-approximation) or not (unsound under-approximation). Tools like Metasm use *ad-hoc* heuristics — that do not claim to be sound — to enhance the readability of the disassembly output; therefore it has to be checked manually.

Hence either we rely on the tool itself to know if its result is correct, or we have to manually convince ourselves. This advocates on one hand for small examples; on the other hand a formal evidence that the result is correct can be obtained *a priori* through machine checked proof. Important is also the precision of the result, that is how tight is the over-approximation. Finally relevant information is the *cost* of the analysis, that is how much resources were used to get a result.

3.2. Results

In this section, for the various obfuscating transformations we studied, we describe the results that we have observed.

3.2.1. Dead Code Insertion

The first obfuscating transformation we study is provided by objobf: the program to transform is diluted within dead code. This obfuscator adds instructions that may be executed without altering the expected behaviour (neither control flow nor *relevant* computed values). It hence increases both code size and execution time.

This obfuscation uses the wide range of the x86 instruction set and adds exotic instructions (`aaa` ascii adjust after addition, `bts` bit test and set, `sbb` integer subtraction with borrow, and so on). Unfortunately such barely used instructions appear not to be handled by analysis tools: they are kept after dead-code elimination in Metasm and unsoundly (and silently: only a warning is issued) replaced by `skip` in Jakstab.

This highlights a weakness of the implementation rather than one of the analysis method: it would be possible to complete the analysers so as to handle all possible instructions. However this would be *long* and error-prone.

3.2.2. Basic Block Splitting

This obfuscation is also provided by objobf. It mainly affects code size: splitting a block consists in choosing a point between two instructions, duplicating all that occurs after this point (*i.e.* creating a brand new block) and adding a conditional jump to flow either towards the original or the new block. Since both target blocks are identical, the jumping condition is irrelevant.

Metasm is not affected, but its user is: this disassembler faithfully outputs the obfuscated program. Jakstab whose analyses are path sensitive, has a larger code and much more feasible paths to analyse. The analysis time hence increases a lot.

3.2.3. Flattening

Behaviours observed on the program shown in figure 3.2 are typical: the description of what has been observed about this program faithfully summarizes all observations about flat programs, handcrafted or generated by Loco.

Jump Table

When a basic block has many successors, as a `switch` statement in C or the *jump* nodes of a flattened program, the addresses of these successors can be gathered in a table, that is to say in consecutive memory locations. Then where the execution goes on after this block is specified by an *index* in this table: e.g. in order to execute the third successor, the third table entry must be fetched; this yields the address of the block to run. Hence an indirect jump instruction has the following shape:

```
jmp *base(,%eax,4)
```

where `base` is the address of the table, `%eax` the register holding the value of the index and `4` the size of each table entry.

Ad-Hoc Jump Table Recovery

In order to know what are the possible successors of an indirect jump, the jump table has to be recovered. To achieve this task, Metasm attempts to recognize some common compiling patterns. Usually — e.g. for compiled `switch` statements — before any indirect jump there is a test ensuring that the index lies inside the bounds of the table. This test may be used to recover these bounds. Then one may assume that any value occurring in the table, *i.e.* within the computed bounds, is a possible target of the jump.

Unfortunately Loco, when flattening the control flow of a function, does not introduce such a test. Indeed since the index register is explicitly set on every path leading to the indirect jump, checking at runtime that the actual value is a valid index is useless. Therefore Metasm has to *guess* what are the table bounds. We noticed that the first non-zero entry at address `base` (hard wired in the instruction) is taken as the beginning of the table, and that a following zero in the table is considered as an end marker.

There are several weaknesses in such heuristics: 1. there may be a *hole* in a jump table; 2. the table may not lie at the `base` address (constant offset); 3. there may be fake addresses mixed amid valid ones. Some of this possibilities actually happen with studied obfuscators.

When disassembling the program of figure 3.2, Metasm correctly recovers the exact set of successors of the indirect jump of line 20. However adding a zero in the table, e.g. between the first an second entries, and accordingly shifting the index values, makes Metasm recover as possible successors the first block only. It is common for Loco to leave such holes in jump tables.

Worse is the case when *spurious* addresses are interleaved with the valid ones in the table: Metasm is unable to distinguish between valid and spurious entries; it therefore outputs all blocks registered in the table as possible successors.

Combining these two techniques leads to this following extreme case: the jump table is made of, at the beginning, spurious entries only, then a zero, and finally the valid ones. In such a case, only spurious addresses are considered as possible targets.

What is important to notice, is that there is no sensible difference between an exact complete result and a completely wrong one. Therefore the Metasm user has to check the retrieved control flow. Otherwise he or she could be bound to study a program that is not the expected one.

Those tricks do not affect Jakstab at all.

Interval Analysis: Misaligned Targets

When `.p2align` directives are used, there is a strided interval exactly representing the set of all targets of the indirect jump. The bounded address tracking analysis of Jakstab fails to analyse this program; the value-set analysis, using strided intervals as abstract domain, succeeds. Without those alignment directives, both analyses fail. However when these two analyses are *jointly* carried on, they succeed.

Strided interval are generally not suited to represent accurately the targets of an indirect jump [BPF11]. Thanks to its ability to use several kinds of abstract domains at the same time, Jakstab can circumvent this intrinsic limitation of VSA-abstract domain.

Stack Height Consistency

Consider a flat program in which a value is pushed onto the stack in some block and popped in another one. This usually happens in programs flattened by Loco to save a register required for control flow computation. In such a program, the height of the stack at the branching block varies: it increases when one block is executed and decreases when another one is executed.

The intra-procedural control flow is correctly recovered but the stack state is over approximated in such a way that, at the end of the function, the return address is considered as possibly overwritten. Hence Jakstab conservatively fails, unable to tell where control flows at the end of the flattened function.

Early Widening

Widening is a standard extrapolation technique of abstract interpretation [CC77]: a post fixpoint is *guessed* and analysis can then be proved to terminate. Such a method is required in some cases to ensure termination or useful to reach a sound result within a reasonable time. In particular it has to be used at least once in each loop [Bou93].

Nevertheless each use of this operation introduces an additional approximation. Hence it has to be used with care. We've noticed that the widening policy of Jakstab lead to a dramatical loss of precision during the analysis of flat programs: the value held by register `eax` at line 20 is approximated by $\top$ (any value) after a few steps. Then nothing can be told about the target of the indirect jump and the analysis is stuck (or goes on after assuming an unsound under-approximation of the set of the possible targets).

To improve the precision of the result, the use of widening can be delayed: the analysis is first carried on without it and after a while only this extrapolation is allowed [Folk]. We altered Jakstab so as to hold up its use of widening; we naïvely used a counter so that, during an execution, the n first calls to `widen` (the merge and extrapolate operation) are changed into calls to `join` (the most precise merge operation), where n is a parameter to tune.

Indeed this patched analysis yields more precise results, in some cases, but is always slower.

3.3. Conclusion

We have there described some experiments that have been carried on small programs, obfuscated either automatically or by hand. As expected from obfuscation, we have observed that there is no analysis that is robust to all obfuscating transformations: for all analysis, there are a program and a transformation such that this transformation prevents this analysis to yield relevant results about this program. This may find a theoretical foundation in the Rice theorem. Dually there are obfuscations that do not affect the analysis methods we studied.

Between these two extrema, analyses 1. show the limits of their implementations, 2. require more resource, or 3. yield unsound results. First this shows that some assumptions made at design time such as "this instruction is never used, thus does not deserve to be considered" may be sound on *usual* programs but invalidate the results about some obfuscated programs. Second obfuscations like basic block splitting which actually increases the number of feasible paths, dramatically increases the cost of Jakstab's value-set analysis, which is path-sensitive. Third an analysis can be stuck due to obfuscation, and an unsound hypothesis has to be assumed in order to continue.

The study of flat programs has highlighted some strengths of advanced static analyses such as VSA. In particular such analyses *know* whether their answer is sound or not, and which unsound hypotheses have been assumed.

Hence there must be possible to improve the results — along with their reliability – of interactive security oriented analysis tools like Metasm. Indeed the availability of VSA-like analyses in such a tool could enable the user to get interesting sound information. However to fit in an interactive setting, the many parameters of the analysis have to be controlled by the user; in addition, the ability to *help* the analyser with unchecked assumptions about the environment (like "this is the only pointer towards this memory location") could improve the accuracy of the result (the soundness of the assumption being under the responsibility of the user).

Nevertheless in the implementation of such a complex analysis about such a rich assembly language, there might be some bug. The next part of this report addresses the certification of an implementation of an analysis.

4. Certified Analysis

Previous experiments have highlighted the fact that analyses which are based on abstract interpretation benefits from their strong theoretical foundation: there is a mathematical proof of the soundness of the results they yield.

However there is a gap between what is actually proved – the algorithm, the analysis method – and the program that is executed. How to be sure that the program is a faithful implementation of the algorithm? An answer can be found in proof assistants like Coq that enable to express within the same formalism an implementation of an analysis along with the proof of its soundness.

In the following we study how an analysis such that VSA can be certified in this way. This work is based on the certified analysis of a toy imperative language described in [CP10]. It is a first step towards a full certification of the value-set analysis: we focus on *some* specificities of the analysis of binary programs, mainly on the fact that such a program has no explicit control flow-graph. Other features like function calls or dynamic memory allocation are ignored in this first approach.

4.1. Language

A toy assembly-like programming language is introduced so as to be able: 1. to write obfuscated programs like the ones studied previously; and 2. to write a sound analyser within the span of the internship. This language is intended to be much simpler than x86 assembly: all instruction have the same size, code cannot self-modify, there are neither functions nor dynamic memory allocations, memory locations do not overlap and so on. However it features an indirect jump that makes the control flow recovery a difficult task.

The *machine* supposed to run programs written in this language is simple as well: it features a few registers —say five R_0 to R_4— and a random access memory where to store *values* —say 32-bit signed integers—; any value is a valid memory address. The machine runs one program at a time and may halt, returning a value, the answer of the run program. The program is stored outside the memory and there is no way to alter it at runtime.

A program is a list of instructions whose syntax is given figure 4.1. Values occurring in control instructions denote where to jump in the program: a *valid program point* is a value greater than zero (the *address* of the first instruction) and strictly less than the length of the program.

Conditional jumps are carried on in two steps: first some comparison is performed, setting some boolean flags and later, a branch is taken or not depending on the value of some flags. Here there is only one flag: `FLE`. After a comparison between a source and a destination registers, this flag is set if and only if the destination is less than or equal to the source.

On the left-hand side of figure 4.3 a small example program is written. Labels, stars and arrows have no special meaning and are expected to ease the understanding of the code. This program reads the content of memory at address 12 and loads it in register R0. If this value is positive, it is returned and the machine stops; otherwise its absolute value is computed using a loop.

Given a program, a machine is either about to execute the instruction at some valid program point or stopped after the program halted. This is referred to as the *state* of the machine. In every state, what actual value are in the memory and the registers and how are the flags is the *configuration* of the machine.

All these concrete semantic domains are summarized below and enable to define the operational semantics of this language.

```
Inductive Instruction :=
  (* arithmetic *)
  | ICst  (V: Value) (R:Register)
  | IAdd  (Rs: Register) (Rd: Register)
  | ICmp  (Rs: Register) (Rd: Register)
  (* control *)
  | ISkip
  | IHalt (R: Register)
  | IGoto (V: Value)
  | IGotoLE:  (V: Value)
  | IGotoInd: (R: Register)
  (* memory *)
  | ILoad  (Rs: Register) (Rd: Register)
  | IStore (Rs: Register) (Rd: Register)
  .
```

Figure 4.1.: Formal syntax

$$\begin{aligned}
\mathsf{int} &= \left\{\, n \in \mathbf{Z} \mid 2^{-31} \leqslant n \leqslant 2^{31} - 1 \,\right\} \\
\mathrm{ProgPoint}_P &= \{\, n \in \mathbf{Z} \mid 0 \leqslant n < |P| \,\} \\
\mathrm{State}_P &= \mathrm{ProgPoint}_P \cup \{\mathrm{Stopped}\} \\
\mathrm{Configuration} &= (\mathsf{Flag} \to \mathsf{Bool}) \times (\mathsf{Reg} \to \mathsf{int}) \times (\mathsf{int} \to \mathsf{int})
\end{aligned}$$

4.1.1. Operational Semantics

Now that it is known how to *write* programs in this language, let us describe how to *run* them. The small-strep operational semantics tells, how every single instruction is to be interpreted.

This semantics of this language is given figure 4.2 as it is formalized in Coq; this definition is parameterised by c, a program. It is defined as a relation between pairs of a state and a configuration, which is to be understood as follow: the relation between $\langle s, M\rangle$ and $\langle t, N\rangle$ holds when running program P from state s and configuration M leads[1] to state t and configuration N. We write $\langle s, M\rangle \underset{P}{\rightrightarrows} \langle t, N\rangle$.

Each constructor of the inductive definition can be seen as an inference rule. For instance the constructor SSKip can be seen as the following one.

$$\mathrm{SSkip}\,\frac{\mathrm{instructionAt}(p) = \mathsf{ISkip} \qquad \mathrm{valid\_prog\_point}(p+1)}{\langle p, f, r, m\rangle \underset{P}{\rightrightarrows} \langle p+1, f, r, m\rangle}$$

It tells that from state p and configuration M (given as a triple (f, r, m) whose components describe the configuration of the flags, registers and memory respectively), the state $p+1$ and configuration M can be reached provided: 1. the instruction at point p is ISkip; and 2. the program point $p+1$ is valid. In the Coq version of this rule, the right-hand side state is written [p+1 | HL]. This construct packs a program point (an integer) along with the proof of its validity (indeed HL is a proof that p+1 is a valid program point of program c). This is a design choice to ensure that only valid program points can be built, *i.e.* the integer p+1 cannot be taken as a program point unless a proof of its validity is provided.

As an other example, the execution of **add** Rs →Rd from $\langle p, f, r, m\rangle$ leads to $\langle p+1, f, r', m\rangle$ where r' is the same register configuration as r but for register Rd which is bound to n, the result of the addition:

$$\mathrm{SAdd}\,\frac{\mathrm{instructionAt}(p) = \mathsf{IAdd\ Rs\ Rd} \qquad \mathrm{valid\_prog\_point}(p+1)}{\langle p, f, r, m\rangle \underset{P}{\rightrightarrows} \langle p+1, f, r[\mathsf{Rd} \mapsto r(\mathsf{Rs}) + r(\mathsf{Rd})], m\rangle}$$

[1]Indeed this relation is functional: for any pair $\langle s, M\rangle$, there is at most one pair $\langle t, N\rangle$ such that $\langle s, M\rangle \underset{P}{\rightrightarrows} \langle t, N\rangle$.

```
Inductive small_step : configuration c → configuration c → Prop :=
| SCst : ∀ p r f m R V (HS: instructionAt p = ICst V R)
    (HL: valid_prog_point c (p+1)),
    ⟨ p , f , r , m ⟩ ⇉ ⟨ [ p+1 | HL ], f , r#R ← V , m ⟩
| SAdd : ∀ p r f m Rs Rd n (HS: instructionAt p = IAdd Rs Rd)
    (HAdd: VAdd (r Rs) (r Rd) = n) (HL: valid_prog_point c (p+1)),
    ⟨ p , f , r , m ⟩ ⇉ ⟨ [ p+1 | HL ], f , r#Rd ← n , m ⟩
| SCmp : ∀ p r f m Rs Rd f' (HS: instructionAt p = ICmp Rs Rd)
    (HCmp: FCmp f (r Rd) (r Rs) = f') (HL: valid_prog_point c (p+1)),
    ⟨ p , f , r , m ⟩ ⇉ ⟨ [ p+1 | HL ], f', r , m ⟩
| SSkip : ∀ p r f m (HS: instructionAt p = ISkip)
    (HL: valid_prog_point c (p+1)),
    ⟨ p , f , r , m ⟩ ⇉ ⟨ [ p+1 | HL ], f , r , m ⟩
| SHalt : ∀ p r f m R (HS: instructionAt p = IHalt R),
    ⟨ p , f , r , m ⟩ ⇉ ⟪ r R , f , r , m ⟫
| SGoto : ∀ p r f m ℓ p' (HS: instructionAt p = IGoto ℓ)
    (HL: prog_point_of_value c ℓ = Some p'),
    ⟨ p , f , r , m ⟩ ⇉ ⟨ p', f , r , m ⟩
| SGotoLEtrue : ∀ p r f m ℓ p' (HS: instructionAt p = IGotoLE ℓ)
    (HLE: ifle f = true) (HL: prog_point_of_value c ℓ = Some p'),
    ⟨ p , f , r , m ⟩ ⇉ ⟨ p', f , r , m ⟩
| SGotoLEfalse : ∀ p r f m ℓ (HS: instructionAt p = IGotoLE ℓ)
    (HLE: ifle f = false) (HL: valid_prog_point c (p+1)),
    ⟨ p , f , r , m ⟩ ⇉ ⟨ [ p+1 | HL ], f , r , m ⟩
| SGotoInd : ∀ p r f m R p' (HS: instructionAt p = IGotoInd R)
    (HL: prog_point_of_value c (r R) = Some p'),
    ⟨ p , f , r , m ⟩ ⇉ ⟨ p', f , r , m ⟩
| SLoad : ∀ p r f m Rs Rd (HS: instructionAt p = ILoad Rs Rd)
    (HL: valid_prog_point c (p+1)),
    ⟨ p , f , r , m ⟩ ⇉ ⟨ [ p+1 | HL ], f , r#Rd ← m (r Rs), m ⟩
| SStore : ∀ p r f m Rs Rd (HS: instructionAt p = IStore Rs Rd)
    (HL: valid_prog_point c (p+1)),
    ⟨ p , f , r , m ⟩ ⇉ ⟨ [ p+1 | HL ], f , r , m +[r Rd ↦ r Rs] ⟩

  where "a ⇉ b" := (small_step a b).
```

Figure 4.2.: Small-step semantics

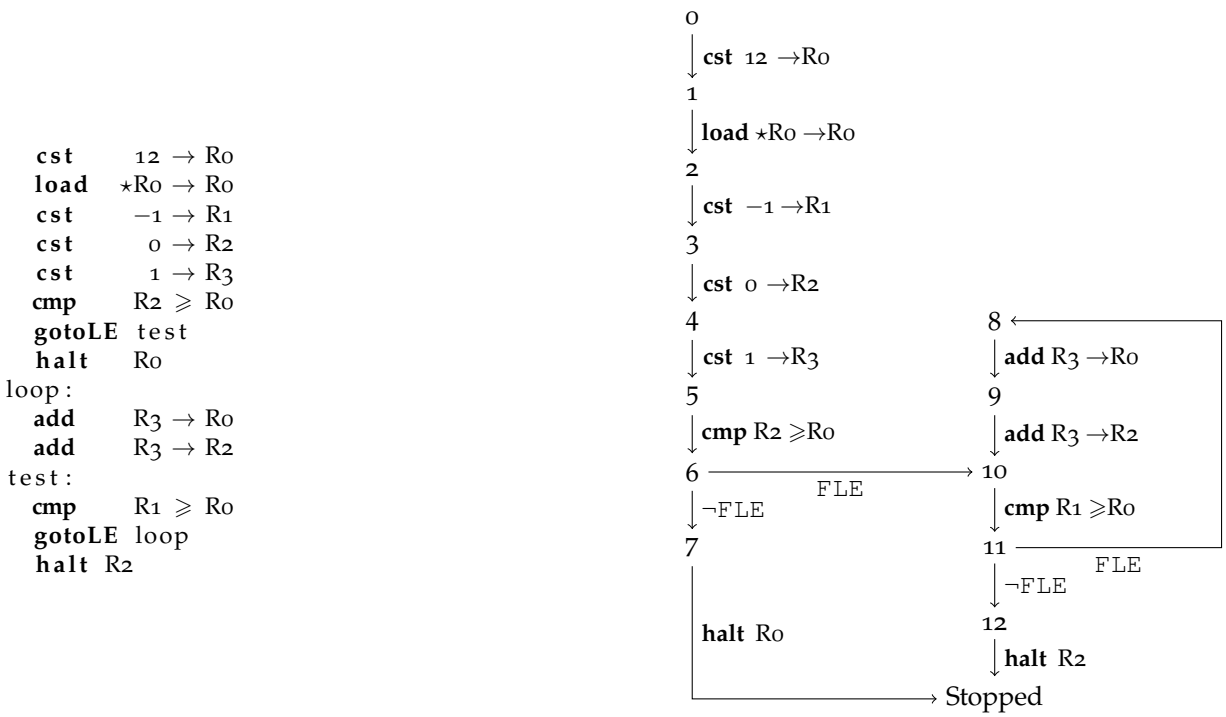

Figure 4.3.: A sample program and its control-flow graph

Recall example program of figure 4.3. Such a program can be represented as a graph (on the right of the figure) whose nodes are the machine states and edges are the instructions, or more accurately how the instructions are interpreted. Indeed branching instructions can lead to several states depending on the actual configuration when they are executed.

This representation of a program as a graph may suggest that the control flow graph is statically known. However in case of an indirect jump, the set of its targets is exactly the set of values a register may hold. Hence recovering this graph is an undecidable problem whose solution is to be approximated by the analysis.

We then introduce, for any program P the set of all configurations that can be reached during some execution of this program. This information, which characterizes the dynamic behaviours of P, is not computable and is to be over-approximated through abstract interpretation.

Definition 1 (Initial Configuration)**.** The execution of a (non empty) program starts in state 0 and some configuration M. Such a pair $\langle 0, M\rangle$ is called an *initial configuration*.

Definition 2 (Reachable Configurations)**.** For a program P the image of the set of initial configurations through the reflexive transitive closure of the small step semantics is the set of *reachable configurations*:

$$\mathcal{R}^\star(P) = \left\{ \langle s, C\rangle \mid \exists C_0, \langle 0, C_0\rangle \underset{P}{\rightrightarrows}^\star \langle s, C\rangle \right\}$$

4.2. Abstract Domains

Let us first describe how sets of integers are abstracted.

```
Record strided_interval := StridedInterval {
  stride : Z;
  l_bound: Z;
  u_bound: Z;
  s_pos: 0 ⩽ stride;
  min_le_lb: min_int ⩽ l_bound;
  lb_le_ub : l_bound ⩽ u_bound;
  ub_le_max: u_bound ⩽ max_int
}.
```

Figure 4.4.: Defining Strided Intervals

4.2.1. Strided Intervals

Definition

Following [RBL06], strided intervals can be defined as a Coq record (see figure 4.4). This data structure can be compared to an OCaml record or a C `struct`. However Coq enables to pack values with properties about them; for instance the member `s_pos` is a proof that the member `stride` is non negative. Notice that the *type* of that member depends on the actual *value* of the stride: such a record is hence called *dependent record*.

Then, using the formalization of machine integers from CompCert [LB08; BL09], the *meaning* of such a strided interval is defined through a concretization relation: $\gamma(s\,[l;u]) = \{n \in \mathsf{int} \mid l \leqslant n \leqslant u \wedge n \equiv l \pmod{s}\}$ as already explained in § 2.2.2.

This concretization induces an equivalence relation on strided intervals. Having several abstract objects represent the same concrete set is a waste; hence only one strided interval is considered in each equivalence class:

Definition 3 (Reduced Strided Interval, RSI)**.** A *reduced* strided interval (RSI) is a strided interval $s\,[l;u]$ satisfying:

$$l \equiv u \pmod{s} \tag{4.1}$$
$$l = u \implies s = 0. \tag{4.2}$$

Condition (4.1) enforces the upper bound to be minimal; indeed for $s\,[l;u]$ a reduced strided interval, $s\,[l;u+1]$, …, $s\,[l;u+s-1]$ all represent the same set. Condition (4.2) enforces the stride to be maximal (w.r.t. divisibility); hence a RSI concretizes as a singleton if and only if its stride is null.

For completeness, an extra value written $\bot$ is taken as an RSI and represents the empty set.

Order and Lattice

The set of RSI is ordered taking $\bot$ as minimal element and $s_1\,[l_1;u_1] \sqsubseteq^{\mathsf{si}} s_2\,[l_2;u_2] \overset{\text{def}}{\iff} l_1 \geqslant l_2 \wedge u_1 \leqslant u_2 \wedge s_2 | s_1 \wedge l_1 \equiv l_2 \pmod{s_2}$. Concretization is then a *monotone* operation: $\forall x, y, x \sqsubseteq^{\mathsf{si}} y \implies \gamma(x) \subset \gamma(y)$.

Lattice join and meet operations are now to be defined.

Definition 4 (Join of reduced strided intervals)**.** Let $s_1\,[l_1;u_1]$ and $s_2\,[l_2;u_2]$ be two RSI. Their least upper bound $s_1\,[l_1;u_1] \sqcup^{\mathsf{si}} s_2\,[l_2;u_2]$ is the reduced strided interval $s\,[l;u]$ where:

$$l = \min(l_1, l_2) \qquad u = \max(u_1, u_2)$$
$$s = \gcd(l_2 - l_1, \gcd(s_1, s_2))$$

Definition 5 (Meet of reduced strided intervals)**.** Let $s_1\,[l_1;u_1]$ and $s_2\,[l_2;u_2]$ be two RSI. Their greatest lower bound $s_1\,[l_1;u_1] \sqcap^{\mathsf{si}} s_2\,[l_2;u_2]$ is defined as follow.

Consider the set of integers that are: 1. congruent with l_1 modulo s_1; 2. congruent with l_2 modulo s_2; 3. greater than l_1 and l_2; and 4. smaller than u_1 and u_2. It is the concretization of the RSI we are looking for. If this set is empty, then its abstraction is $\bot$; otherwise let l be the smallest integer in this set and let

u be the greatest one; this yields $s_1\,[l_1; u_1] \sqcap^{\text{si}} s_2\,[l_2; u_2] = s\,[l; u]$ where $s = 0$ if $l = u$ and $s = \operatorname{lcm}(s_1, s_2)$ otherwise.

Those operations are proved correct in Coq: they are respectively least upper bound and greatest lower bound.

Arithmetics

A few arithmetic operators are defined on RSIS, so as to abstract the one encountered in programs: addition, comparison and constant.

Given two (reduced) strided intervals $s_1\,[l_1; u_1]$ and $s_2\,[l_2; u_2]$, the result of their addition have to encompass all integers that are the result of two integers taken it these two RSIS (respectively). For instance $2\,[0; 4]$ (denoting the set $\{\,0, 2, 4\,\}$) added to $3\,[5; 8]$ (denoting $\{\,5, 8\,\}$) yields the smallest RSI holding $\{\,5, 7, 8, 9, 10, 12\,\}$ which is $1\,[5; 12]$. This suggests as result of the addition: $(\gcd(s_1, s_2)\,[l_1 + l_2; u_1 + u_2]$. However special care has to be taken due to possible overflow. In [RBL06] since k-bit integers are represented using k-bit integers, detecting overflows requires sophisticated bit-level manipulations as described in [War03]. Here the use of unbounded binary integers (Coq datatype **Z**) eases this detection.

So as to better understand what the abstract comparison operator is, consider the instruction sequence taken from figure 4.3: **cmp** R2 $\geqslant$R0 ; **gotoLE** test. The last instruction is interpreted in to ways, depending on the actual value of the flag: either the branch is taken (and execution continues from label `test`) or it is not. When analysing this instruction, it is pointless to try to decide which branch is taken: generally both branches may be followed, and both have to be analysed. However since each branch is analysed independently (both are not taken *at the same time*) the content of these registers is better known after the instruction is analysed: if execution jumps towards `test`, it is known that the actual value held by register R0 is less than or equal to the one held by register R0; in the other case, the opposite holds. Then the knowledge about these two registers can be refined to take into account the information about which branch is analysed.

4.2.2. Flag

During a concrete execution of a conditional jump, the branch that is actually taken is decided using the value of the flag only. Hence analysing one of the branches directly gives the value of the flag and nothing more.

Therefore to be able to use the abstract comparison described above, we need to know which registers were involved in the comparison that set the flag. Hence the abstract domain tracks this kind of information: the flag is abstracted by the pair of registers that were involved in the last comparison.

4.2.3. Lattices

Abstract domains are formalized as *decidable* lattices with bottom, *i.e.* lattices whose operations — equality, order, join, meet — are decidable (or computable) and in which there is a unique minimal element. This formalization is taken from [CP10] and is designed in a modular and convenient way, using Coq type classes [SO08].

For instance two lattices K and L can be composed (using *functors*) to build the product lattice $K \times L$ on which operations are defined componentwise.

Similarly, given a set L, the flat semi-lattice with top is defined, by means of functors, and noted $\lceil L \rceil$: an extra top element $\top$ is appended to L, with a flat order ($s \leqslant t \overset{\text{def}}{\iff} (s = t) \vee (t = \top)$), and a least upper bound operation ($s \cup s = s$ and $s \cup t = \top$ when $s \neq t$).

So as to abstract functions, we rely on maps, as defined in [Les10]. This data structure is suited to represent functions with finite domain, or that are partially defined: a value with no image in such a map is considered as bound to the top element of the codomain lattice. Therefore for untractable domains as the one of machine integers, there is no way to build the *bottom* function, which maps each integer to bottom.

Nevertheless for any (ordered) domain A and any codomain B that is a decidable lattice with top, it is possible to provide the set of maps from A to B with a decidable lattice structure with top.

Finally using many lattice functors, the following abstract domains are defined, where $\bot_c^\sharp$ stands for an extra *bottom* abstract configuration.

$$\text{Configuration}^\sharp = \lceil \mathsf{Reg} \times \mathsf{Reg} \rceil \times (\mathsf{Reg} \to \textsc{rsi}) \times (\mathsf{int} \to \textsc{rsi})$$
$$\text{Env}_P^\sharp = \text{State}_P \to \text{Configuration}^\sharp \cup \bot_c^\sharp$$

Since there are finitely many states (for any given program P), the bottom abstract environment (mapping each state to $\bot_c^\sharp$) is computable.

Whithin such a setting, consider for instance the following abstract configuration.

$$\left((\mathsf{R2}, \mathsf{R0}), \left\{ \mathsf{R1} \mapsto 0\,[-1;-1], \mathsf{R2} \mapsto 0\,[0;0], \mathsf{R3} \mapsto 0\,[1;1] \right\}, \left\{ \right\}\right)$$

First component (R2,R0) tells which was the last comparison; second is a map binding some registers to constants: R1 is known to hold -1 but which value is held by R0 is unknown; last is the empty map, meaning that nothing is known about the memory state.

4.3. Computing in the Abstract

An abstract environment binds an abstract configuration to each node (or program point, or state). The purpose of the analysis is to compute such an environment that is: 1. sound, *i.e.* an over-approximation of the reachable configurations; and 2. precise, *i.e.* smallest according to lattice order.

Computing the abstract semantics of a program amounts to run it *in the abstract*. Therefore we have to define how to interpret each instruction in an abstract environment. More accurately a program is seen as a graph in which nodes are the intructions and edges match the control flow; the abstract semantics describes how a configuration evolves when an edge is taken.

4.3.1. Abstract small step

The abstract semantics of a program P is computed through an abstraction of the small step relation:

$$\mathbf{post}_P^\sharp : \text{State}_P \times \text{Configuration}^\sharp \to \mathbf{list}\left(\text{State}_P \times \text{Configuration}^\sharp\right)$$

The application of this function to $(pp, C^\sharp)$ executes the instruction at point pp in program P on the abstract configuration $C^\sharp$. This yields a finite set — represented by a list — since an instruction has possibly zero or many successors.

The actual behaviour of this function is directed by the instruction to execute. Indeed for each instruction — or more precisely for each semantic rule — there is an abstract transformer. Some of them are described below. An extract from the Coq code corresponding to the abstract small step may be found in appendix B.

Skip

Albeit trivial, the skip instruction has two possible behaviours. If this instruction is (syntactically) the last in the program, then it has no semantics: the concrete execution would be stuck. In such a case, its abstract execution yields the empty set. Nevertheless in the general case, it yields the singleton with next state and same configuration.

Comparison

The cmp instruction mostly behaves as skip; only the flag is altered. Hence in the resulting configuration (if any) the abstract flag is set to the pair of registers involved in the comparison.

Conditional Jump

Consider for instance the interpretation of gotoLE test at point 6 in the program shown figure 4.3 in a configuration $C^\sharp$ in which: 1. the abstract flag is the pair (R2,R0); 2. R0 is bound to top; and 3. R2 is bound to $0\,[0;0]$. There are two edges to follow, leading to points 7 and 10. In a concrete setting, the configuration is not altered during the execution of such an instruction. However in the abstract both edges are analysed independently and the configuration can be refined: if execution flows from 6 to 7, this means that the last comparison has set the flag to zero, hence that the actual content of register R0 is greater than the actual content of register R2. Therefore the resulting configuration bound to state 7 is the same as $C^\sharp$ but for these two registers that are updated using the comparison function defined on RSIS: R0 is then bound to $1\,[1;+\infty]$ and R2 unchanged.

Constant

As for skip, the execution of **cst** $\mathrm{V} \rightarrow$R may be stuck. If not, it has two effects of the abstract configuration. First the register R is bound to the abstract value representing the constant V. The more subtle effect affects the flag. Indeed when the abstract flag is used to refine information about registers, the current values of the registers need to be the same as the values they held by the time of the comparison. Hence whenever a register is updated between a comparison and a conditional jump, the abstract flag has to be *invalidated*, *i.e.* set to $\top$ which stands for "no information".

Indirect Jump

The instruction **goto** $\star$R routes the exectution towards the program point stored in register R, provided that point is valid. Unfortunately in an abstract configuration, this register may hold several values. Therefore to interpret that instruction, the set of possible targets of the jump has to be computed: this is exactly the set of values that are valid program points among all values the register may hold. Such a set can be computed by filtering the set of valid program points since testing if a concrete value is in the concretization of a RSI is decidable.

As for conditional jumps, the knowledge about which edge is followed can be taken into account and the abstract value of register R set to the RSI that exactly represents the actual target. A register is indeed updated — in the abstract — but there is no need to invalidate the flag: the concrete execution does not alter the configuration.

Memory load and store

Among all 2^{32} memory locations, only a few are *tracked* during an analysis: these ones are called *a-loc*s. The content of any other location is soundly over-approximated by the top strided interval (any value). A memory location is accessed through its address, held by a register. The content of this register is approximated by a strided interval which thus may represent several locations.

Therefore a load results in a join of all possibly accessed locations (including non *a-loc*s); a store updates all possibly accessed *a-loc*s.

The set of *a-loc*s is built iteratively when analysing a program: the program is first analysed considering no *a-loc*. Then the result of this *round* — which is sound: the whole memory is approximated to *top* — is used to build a first set of interesting locations: for each load or store instruction in which the register holding the address of the accessed location is bound to a RSI whose concretization is *small* (*i.e.* holds a few values only), this concretization is added to the set of *a-loc*s. For instance reading from $1\,[2;3]$ leads to considering 2 and 3 as *a-loc*s; a read from $1\,[0;+\infty]$ is ignored: otherwise the set of *a-loc*s would be too large.

Using this new set of locations to track, the program is analysed once again. Such rounds of analysis followed by *a-loc*s discovery are repeated until no new *a-loc* is discovered.

The right way of formalizing this set of *interesting* locations is still unclear are requires a better understanding; thus the soundness of the abstract semantics of loads and stores is currently admitted (and may be wrong).

4.3.2. Work-set iteration

The abstract semantics we look for is an abstract environment $E^\sharp$ that is a *fixpoint*: for all state s, carrying on an abstract small step from $\big(s, E^\sharp(s)\big)$ yields bindings that are already in $E^\sharp$.

We compute the abstract semantics of a program exploring its (unknown) control flow graph, starting from the entry point and following the edges. From a node and current configuration at this node, an abstract small step yields the set of edges that can be taken from this node along with the reached configurations at the end of these edges. Whenever such a reached configuration tells something new, the corresponding state is added to the work-set so as to continue the graph exploration taking the new piece of information into account.

When there is no more node in the work-set, a fixpoint is reached.

4.3.3. Widening oracle

Even though this algorithm will terminate this may take a long time. Indeed recall the example program from figure 4.3: in the loop, at instruction 9, register R2 is incremented by one. When first analysed, this instruction will lead to a configuration in which this register is bound to one. This configuration is then merged (in state 10) with a previous one, leading to R2 bound to $1\,[0;1]$. The loop is therefore analysed once more, yielding $1\,[0;2]$ as abstract value for R2. This analyse of the loop can be repeated about 2^{31} times without reaching a fixpoint.

Hence the analysis needs to be accelerated. The standard acceleration method of abstract interpretation is widening: extrapolate a (post) fixpoint so as to ensure termination.

Widening may occur whenever two configurations are merged, as for program point 10 in the previous example. It has to be used often enough to ensure termination, but warily lest the final result be inaccurate, as mentioned in § 3.2.3.

Our implementation of the analysis relies on an *oracle* which, given an edge, decides if widening has to be used rather than the usual join. Widening has to be applied in every loop. Unfortunately the language is not structured enough to tell where loops are. A simple approximation is to consider any backward jump (*i.e.* from point p to point q where $q < p$) as a back-edge of a loop. Indeed it is not possible to write a loop with only forward jumps.

There is nothing to prove about such a widening policy, hence its name *oracle* [Ler06].

We provided a stateful instance of the oracle, advising to use widening when a backward edge is taken and only after this edge has been visited at least ten times.

With such a strategy, the analysis of the program of figure 4.3 eventually applies widening when reaching point 8: abstract value of register R2 is therefore extrapolated to $1\,[0;+\infty]$. Then the addition at point 9 is interpreted: because of possible overflow, R2 is bound to $\top$, and the analysis will terminate.

4.4. Proving a Result Sound

Our purpose is to prove that the abstract environment computed when analysing a program is a sound over approximation of the semantics of this program.

4.4.1. Intermediate Semantics

To ease the reasonning, an intermediate semantics is introduced as a first approximation. There are two main differences between the original operational semantics and this one, so as to better fit the abstract domains: 1. the configuration remembers which registers were involved in the last comparison; and 2. the semantic domain is turned into a function rather than a relation.

An extended configuration is first introduced:

$$\text{Configuration}' = \text{Configuration} \times (\mathsf{Reg} \times \mathsf{Reg})$$

Then the small step semantics is refined to handle this new piece of information: all instructions but ICmp propagate it and ICmp Rs Rd sets it to (Rs, Rd). This new relation is written $\underset{P}{\rightrightarrows}'$.

The set of reachable configurations is also restated as a function telling for each state what are the possible configurations. Such a binding is called an *environment*.[2]

$$\mathrm{Env}_P = \mathrm{State}_P \to \mathcal{P}\left(\mathrm{Configuration}'\right)$$

Running one step of a program P in an environment E yields a new environment:

$$\mathbf{post}\left[\underset{P}{\rightrightarrows}'\right](E) = s \mapsto \left\{C \mid \exists s_0 \in \mathrm{State}_P, \exists C_0 \in E(s_0), \langle s_0, C_0\rangle \underset{P}{\rightrightarrows}' \langle s, C\rangle\right\}$$

Finally the semantics of a program P is defined as the least fix point of the following monotone operator, defined in terms of the initial environment I:

$$\begin{aligned} I &= s \mapsto \begin{cases} \mathrm{Configuration}' & \text{if } s = 0 \\ \emptyset & \text{otherwise} \end{cases} \\ F_P &= E \mapsto I \sqcup \mathbf{post}\left[\underset{P}{\rightrightarrows}'\right](E) \\ [\![P]\!] &= \mathbf{lfp}\, F_P \end{aligned}$$

The relationship between the operational semantics and the intermediate one is stated by:

Theorem 6. Let P be a program. Its intermediate semantics $[\![P]\!]$ is an over-approximation of the reachable-configurations semantics:

$$\forall \langle s, C\rangle \in \mathcal{R}^\star(P), \quad \exists q \in \mathsf{Reg} \times \mathsf{Reg}, \quad (C, q) \in [\![P]\!](s)$$

4.4.2. Concretization

Definition 7 (Concretization of abstract environments)**.**

$$\gamma : \begin{cases} \mathrm{Env}^\sharp \to \mathrm{Env} \\ E \longmapsto \gamma_c \circ E \end{cases}$$

where $\gamma_c : \mathrm{Configuration}^\sharp \cup \bot_c^\sharp \to \mathcal{P}(\mathrm{Configuration}')$ relates abstract and concrete configurations: the abstract register state $r^\sharp$ approximates the concrete one; similarly the abstract memory soundly approximates the concrete one; and the abstract flag holds either $\top$ either the pair of registers that were involved in the last comparison.

We proved that this function is monotone and a meet morphism.

4.4.3. Subject Reduction

The key lemma in proving the soundness of the abstract interpretation states that the abstract small step semantics preserves the soundness of an approximation.

Theorem 8. Let P be a program, s_1, s_2 be states and C_1, C_2 configurations such that $\langle s_1, C_1\rangle \underset{P}{\rightrightarrows}' \langle s_2, C_2\rangle$. For all abstract configuration $C_1^\sharp$ that is a correct abstraction of C_1, i.e. $C_1 \in \gamma_c(C_1^\sharp)$, there is an abstract configuration $C_2^\sharp$ such that

$$(s_2, C_2^\sharp) \in \mathbf{post}_P^\sharp(s_1, C_1^\sharp) \qquad \text{and} \qquad C_2 \in \gamma_c(C_2^\sharp)$$

[2] The set of environments is pointwise (partially) ordered: $E_1 \sqsubset E_2 \overset{\text{def}}{\iff} \forall s, E_1(s) \subset E_2(s)$ and equipped with a pointwise lattice structure: $E_1 \sqcup E_2 \overset{\text{def}}{=} s \mapsto E_1(s) \cup E_2(s)$ and $E_1 \sqcap E_2 \overset{\text{def}}{=} s \mapsto E_1(s) \cap E_2(s)$

4.4.4. Fixpoint validation

Given an abstract environment $E^\sharp$, it can be checked whether it is a fixpoint or not. Indeed it suffices, from each state s, to perform an abstract small step from $(s, E^\sharp(s))$ and to compare the reached configurations with the ones in $E^\sharp$. This verification is achieved by the function validate_fix_point which decides the following property:

$$E^\sharp(0) = \top \quad \wedge \quad \forall s : \text{State}_P,\ \forall (s_2, C_2^\sharp) \in \mathbf{post}_P^\sharp(s, E^\sharp(s)),\ C_2^\sharp \sqsubseteq^\sharp E^\sharp(s_2)$$

Theorem 9. Let P be a program, and $E^\sharp$ an abstract environment.

$$\text{validate\_fix\_point}_P(E^\sharp) = \text{true} \implies [\![P]\!] \sqsubset \gamma(E^\sharp)$$

Corollary 10. Let P be a program, and $E^\sharp$ an abstract environment.

$$\text{validate\_fix\_point}_P(E^\sharp) = \text{true} \implies \forall s, \forall C, \langle s, C\rangle \in \mathcal{R}^\star(P) \implies \exists q \in \mathsf{Reg} \times \mathsf{Reg}, (C, q) \in \gamma(E^\sharp)(s)$$

4.5. Discussion

After studying in § 2.2 the value-set analysis from an algorithmic point of view, and through an experimental approach in § 3, we addressed here the *certification* of such an analysis of binary programs. That is, how to mechanically prove the soundness of the analyser.

A minimalist language has been introduced. It features an indirect jump instruction that is enough to illustrate the problem of control flow graph reconstruction. Then based on the informal description of the value-set analysis [BR10], an abstract interpretation analysis has been formalized using the Coq proof assistant. In the same formalism, a function that validates an analysis result has been programmed and proved sound: such a certified validator is able to prove (with strong formal guaranties) that a result of an analysis is sound.

Coq *extraction* facility enables to compile and run a static analyser: programs written in the toy assembly language can be interpreted *concretely* (on some input) as well as analysed. For instance carried on the program said to compute the absolute value of its input (figure 4.3), the analyser is unable to prove that the answer value is positive. Indeed the addition (in the loop) may overflow…

More interesting is the case shown as appendix A: 1. lines 0 to 5 set up a jump table starting at address 10; 2. the block jmp (lines 7 to 11) uses register R0 as index in this table; 3. block A (lines 12 to 16) stores B (seventeen) at address B; 4. block B (lines 17 to end) reads and returns the value stored in memory by block A. This is a *flat* program as the ones studied in § 3. The analysis of this program is carried on in two rounds: first without tracking any memory location; then tracking those locations that might be of interest according to the results of the first round.

The *raw* output is indeed hardly readable. One may however notice: 1. a fixpoint is quickly reached; 2. the first round does not provide much information; 3. the first round does provide enough information so as to successfully carry on the second round; 4. in the final result, in state 11, R0 is bound to $5\,[12; 17]$ which means: "the target of the indirect jump is twelve or seventeen"; 5. the *flat* control flow graph of the program is recovered: there is no clue about the fact that block B cannot be run before block A. This last sentence highlights that there is no path-sensitivity at all in VSA.

The analyser described here address only a few questions that arise when analysing executables. This work could be extended taking into account function calls, dynamic memory allocation, or be applied to the whole set of x86 instructions. It does not seem possible to soundly cover the hundreds of x86 instructions using a manual approach as we did. More automated techniques like the one presented by Junghee Lim in [Lim11] "that provides a systematic solution to the problem of creating retargetable tools for analyzing machine-code" may be further investigated.

5. Conclusion

This report has first reviewed common obfuscation techniques and described the value-set analysis intended to be applied to programs under their executable form. Analysing this rather than source code is mostly motivated by the WYSINWYX phenomenon as well as by source code being possibly unavailable. However those programs may be obfuscated. We then described our experimental investigation about the behaviour of various actual static analysers when they are given obfuscated programs to analyse, trying to understand what makes them robust or weak. We have focused on two analysers, namely Metasm and Jakstab, which represent two classes of analysis tools; we therefore studied obfuscating transformations that were relevant w.r.t. these tools. Finally we presented our machine checked formalization in a proof assistant of an analysis of a toy assembly-like language: we have formalized the VSA abstract domain, *i.e.* strided intervals, programmed an abstract interpreter following VSA informal description from [BR10], and certified a checker of the results of this interpreter.

The experiments have shown that when a program to analyse is obfuscated, the analysis cost increases and the precision of its result decreases, as expected. In addition, a loss in soundness is observed: analysers are more prone to return unsound results when the analysee is obfuscated. A difference between the two kinds of tools is that there is no differentiation between sound and unsound results in Metasm although abstract interpretation based analyses like the ones used in Jakstab do know whether their results are sound or not. Additionally the use of strided intervals as an abstract domain during the analysis of executable programs have shown its relevance to represent the entries of a jump table, enabling Jakstab to get better result analysing flat programs. However the interactive approach of Metasm is really interesting and an implementation of VSA like features in it could improve the current heuristics, so that its users could benefit from it.

Our last contribution is the formalization of a analyser for a low-level language in which a program control-flow graph is a dynamic property. A tool that is able to prove that an analysis result is actually sound has been certified so that there is no possible doubt about the validity of the result. However the considered language is simple and small; other features as functions must be investigated. A more realistic language also have to be handled: the hundreds of instructions from the x86 set would require the use of more automated techniques, for instance refining works described in [Lim11].

A. Sample

A.1. Source

```
0     cst    B → R1
1     cst    10 → R0
2     store  R1 → ⋆R0
3     cst    A → R1
4     cst    11 → R0
5     store  R1 → ⋆R0
6     cst    1 → R0
7   jmp:
8     cst    10 → R1
9     add    R1 → R0
10    load   ⋆R0 → R0
11    goto   ⋆R0
12  A:
13    cst    0 → R0
14    cst    B → R1
15    store  R1 → ⋆R1
16    goto   jmp
17  B:
18    load   ⋆R0 → R0
19    halt   R0
```

A.2. Analysis: round one

```
Alocs (0):
Stopped ->
  { Flags:
Regs : [R1 -> 0[10 ; 10]]
Mem  :  }
R@ 0: cst 17 -> R1 ->
  { Flags:
Regs :
Mem  :  }
R@ 1: cst 10 -> R0 ->
  { Flags:
Regs : [R1 -> 7[10 ; 17]]
Mem  :  }
R@ 2: store R1 -> *R0 ->
  { Flags:
Regs : [R0 -> 8[2 ; 10]] [R1 -> 7[10 ; 17]]
Mem  :  }
R@ 3: cst 12 -> R1 ->
  { Flags:
Regs : [R0 -> 1[2 ; 10]] [R1 -> 7[10 ; 17]]
Mem  :  }
R@ 4: cst 11 -> R0 ->
  { Flags:
Regs : [R0 -> 1[2 ; 10]] [R1 -> 2[10 ; 12]]
Mem  :  }
R@ 5: store R1 -> *R0 ->
  { Flags:
Regs : [R0 -> 6[5 ; 11]] [R1 -> 2[10 ; 12]]
Mem  :  }
R@ 6: cst 1 -> R0 ->
  { Flags:
Regs : [R0 -> 1[5 ; 11]] [R1 -> 2[10 ; 12]]
Mem  :  }
R@ 7: skip ->
  { Flags:
Regs : [R0 -> 1[0 ; 16]] [R1 -> 1[10 ; 17]]
Mem  :  }
R@ 8: cst 10 -> R1 ->
   Flags:
Regs : [R0 -> 1[0 ; 16]] [R1 -> 1[10 ; 17]]
Mem  :  }
R@ 9: add R1 -> R0 ->
  { Flags:
Regs : [R0 -> 1[0 ; 16]] [R1 -> 0[10 ; 10]]
Mem  :  }
R@ 10: load *R0 -> R0 ->
  { Flags:
Regs : [R0 -> 1[10 ; 26]] [R1 -> 0[10 ; 10]]
Mem  :  }
R@ 11: goto *R0 ->
  { Flags:
Regs : [R1 -> 0[10 ; 10]]
Mem  :  }
R@ 12: skip ->
  { Flags:
Regs : [R0 -> 0[12 ; 12]] [R1 -> 0[10 ; 10]]
Mem  :  }
R@ 13: cst 0 -> R0 ->
  { Flags:
Regs : [R0 -> 1[12 ; 13]] [R1 -> 0[10 ; 10]]
Mem  :  }
R@ 14: cst 17 -> R1 ->
  { Flags:
Regs : [R0 -> 14[0 ; 14]] [R1 -> 0[10 ; 10]]
Mem  :  }
R@ 15: store R1 -> *R1 ->
  { Flags:
Regs : [R0 -> 1[0 ; 15]] [R1 -> 7[10 ; 17]]
Mem  :  }
R@ 16: goto 7 ->
  { Flags:
Regs : [R0 -> 1[0 ; 16]] [R1 -> 7[10 ; 17]]
Mem  :  }
R@ 17: skip ->
  { Flags:
Regs : [R0 -> 0[17 ; 17]] [R1 -> 0[10 ; 10]]
Mem  :  }
R@ 18: load *R0 -> R0 ->
  { Flags:
Regs : [R0 -> 1[17 ; 18]] [R1 -> 0[10 ; 10]]
Mem  :  }
R@ 19: halt R0 ->
  { Flags:
Regs : [R1 -> 0[10 ; 10]]
Mem  :  }
R@ 20: skip ->
  { Flags:
Regs : [R0 -> 0[20 ; 20]] [R1 -> 0[10 ; 10]]
Mem  :  }

After 35
```

A.3. Analysis: final round

```
Alocs (19): 2 5 11 12 13 14 15 16 19 20 21
  22 23 24 25 26 10 17 18
Stopped ->
  { Flags:
Regs : [R1 -> 0[10 ; 10]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]
       [17 -> 1[-∞ ; +∞]]  }
R@ 0: cst 17 -> R1 ->
  { Flags:
Regs :
Mem  :  }
R@ 1: cst 10 -> R0 ->
  { Flags:
Regs : [R1 -> 0[17 ; 17]]
Mem  :  }
R@ 2: store R1 -> *R0 ->
  { Flags:
Regs : [R0 -> 0[10 ; 10]] [R1 -> 0[17 ; 17]]
Mem  :  }
R@ 3: cst 12 -> R1 ->
  { Flags:
Regs : [R0 -> 0[10 ; 10]] [R1 -> 0[17 ; 17]]
Mem  : [10 -> 0[17 ; 17]]  }
R@ 4: cst 11 -> R0 ->
  { Flags:
Regs : [R0 -> 0[10 ; 10]] [R1 -> 0[12 ; 12]]
Mem  : [10 -> 0[17 ; 17]]  }
R@ 5: store R1 -> *R0 ->
  { Flags:
Regs : [R0 -> 0[11 ; 11]] [R1 -> 0[12 ; 12]]
Mem  : [10 -> 0[17 ; 17]]  }
R@ 6: cst 1 -> R0 ->
  { Flags:
Regs : [R0 -> 0[11 ; 11]] [R1 -> 0[12 ; 12]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]  }
R@ 7: skip ->
  { Flags:
Regs : [R0 -> 1[0 ; 1]] [R1 -> 5[12 ; 17]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]
       [17 -> 1[-∞ ; +∞]]  }
R@ 8: cst 10 -> R1 ->
  { Flags:
Regs : [R0 -> 1[0 ; 1]] [R1 -> 5[12 ; 17]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]
       [17 -> 1[-∞ ; +∞]]  }
R@ 9: add R1 -> R0 ->
  { Flags:
Regs : [R0 -> 1[0 ; 1]] [R1 -> 0[10 ; 10]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]
       [17 -> 1[-∞ ; +∞]]  }
R@ 10: load *R0 -> R0 ->
  { Flags:
Regs : [R0 -> 1[10 ; 11]] [R1 -> 0[10 ; 10]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]
       [17 -> 1[-∞ ; +∞]]  }
R@ 11: goto *R0 ->
  { Flags:
Regs : [R0 -> 5[12 ; 17]] [R1 -> 0[10 ; 10]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]
       [17 -> 1[-∞ ; +∞]]  }
R@ 12: skip ->
  { Flags:
Regs : [R0 -> 0[12 ; 12]] [R1 -> 0[10 ; 10]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]  }
R@ 13: cst 0 -> R0 ->
  { Flags:
Regs : [R0 -> 0[12 ; 12]] [R1 -> 0[10 ; 10]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]  }
R@ 14: cst 17 -> R1 ->
  { Flags:
Regs : [R0 -> 0[0 ; 0]] [R1 -> 0[10 ; 10]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]  }
R@ 15: store R1 -> *R1 ->
  { Flags:
Regs : [R0 -> 0[0 ; 0]] [R1 -> 0[17 ; 17]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]  }
R@ 16: goto 7 ->
  { Flags:
Regs : [R0 -> 0[0 ; 0]] [R1 -> 0[17 ; 17]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]
       [17 -> 0[17 ; 17]]  }
R@ 17: skip ->
  { Flags:
Regs : [R0 -> 0[17 ; 17]] [R1 -> 0[10 ; 10]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]
       [17 -> 1[-∞ ; +∞]]  }
R@ 18: load *R0 -> R0 ->
  { Flags:
Regs : [R0 -> 0[17 ; 17]] [R1 -> 0[10 ; 10]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]
       [17 -> 1[-∞ ; +∞]]  }
R@ 19: halt R0 ->
  { Flags:
Regs : [R1 -> 0[10 ; 10]]
Mem  : [10 -> 0[17 ; 17]] [11 -> 0[12 ; 12]]
       [17 -> 1[-∞ ; +∞]]  }
R@ 20: skip ->

After 25
Fini !
```

B. Abstract Small Step

```
[...]

Record ab_config := AbConfig {
  ab_flags : option (Register * Register);
  ab_reg: Map [ Register , rsi k];
  ab_mem: Map [ Value, rsi k]
}.

Definition cst_rsi (V:Value) : rsi _ := match V with Vint i ⇒
  Some (const_rsi k i)
end.

Definition invalidate_flag (R:Register) F :=
match F with
| None ⇒ None (* a.k.a. top *)
| Some (r1, r2) ⇒ if register_dec R r1 then None else if register_dec R r2 then None else F
end.

Definition abstract_cst (V: Value) (R:Register) (conf: ab_config) : ab_config :=
  AbConfig
  (invalidate_flag R (ab_flags conf))
  (MapInterface.add R (cst_rsi V) (ab_reg conf))
  (ab_mem conf).

Definition abstract_add (Rs Rd: Register) (conf: ab_config) : ab_config :=
  let svalue := find_default Rs (ab_reg conf) in
  let dvalue := find_default Rd (ab_reg conf) in
    AbConfig
    (invalidate_flag Rd (ab_flags conf))
    (MapInterface.add Rd (add_rsi svalue dvalue)
      (ab_reg conf))
    (ab_mem conf).

Definition abstract_cmp (Rs Rd: Register) (conf: ab_config) : ab_config :=
  AbConfig (Some (Rs,Rd)) (ab_reg conf) (ab_mem conf).

Definition abstract_le (conf : ab_config) : ab_config :=
match ab_flags conf with
| None ⇒ conf
| Some (Rx, Ry) ⇒
  let x := find_default Rx (ab_reg conf) in
  let y := find_default Ry (ab_reg conf) in
  let (new_y, new_x) := le_rsi y x in
    AbConfig (ab_flags conf)
    (MapInterface.add Rx new_x
      (MapInterface.add Ry new_y
        (ab_reg conf)))
    (ab_mem conf)
end.

Lemma abstract_le_mem : ∀ conf,
  ab_mem (abstract_le conf) = ab_mem conf.
Proof.
 intros C; unfold abstract_le.
 destruct (ab_flags C) as [(r1, r2)|];[|reflexivity].
 destruct (le_rsi (find_default r2 (ab_reg C)) (find_default r1 (ab_reg C)));reflexivity.
Qed.

Definition abstract_nle (conf : ab_config) : ab_config :=
match ab_flags conf with
```

```
| None ⇒ conf
| Some (Rx, Ry) ⇒
  let x := find_default Rx (ab_reg conf) in
  let y := find_default Ry (ab_reg conf) in
  let (new_y, new_x) := nle_rsi y x in
    AbConfig (ab_flags conf)
    (MapInterface.add Rx new_x
      (MapInterface.add Ry new_y
        (ab_reg conf)))
    (ab_mem conf)
end.

Lemma abstract_nle_mem : ∀ conf,
  ab_mem (abstract_nle conf) = ab_mem conf.
Proof.
 intros C; unfold abstract_nle.
 destruct (ab_flags C) as [(r1, r2)|];[|reflexivity].
 destruct (nle_rsi (find_default r2 (ab_reg C)) (find_default r1 (ab_reg C)));reflexivity.
Qed.

Variable max_deref : Z.

Definition abstract_deref (R : Register) (conf: ab_config) : rsi _ :=
  let r := find_default R (ab_reg conf) in
    if Z_le_dec (rsi_cardinal r) max_deref
      then
        fold_left
        (fun l x ⇒ join_bot k (find_default (Vint x) (ab_mem conf)) l)
        (concretize_rsi r)
        None
      else
        Some (si_top k).

Definition abstract_load (Rs Rd: Register) (conf: ab_config) : ab_config :=
  AbConfig
  (invalidate_flag Rd (ab_flags conf))
  (MapInterface.add Rd
    (abstract_deref Rs conf)
    (ab_reg conf))
  (ab_mem conf).

(* strong update ! *)
Definition abstract_store_mem (alocs: list Value) (value destination : rsi k) mem
  : Map [Value, rsi k] :=
 fold_left
    (fun m addr ⇒
      match addr with Vint i ⇒
        if Γ_dec destination i
          then MapInterface.add addr value m
          else m
      end)
    alocs
    mem
.

Definition abstract_store (alocs: list Value) (Rs Rd: Register)
  (conf: ab_config) : ab_config :=
  let value := find_default Rs (ab_reg conf) in
  let destination := find_default Rd (ab_reg conf) in
    AbConfig (ab_flags conf)
    (ab_reg conf)
    (abstract_store_mem alocs value destination (ab_mem conf)).

Definition gotoIndTargets (c:Program) (R: Register) (conf: ab_config):
  list (prog_point c) :=
let targets := find_default R (ab_reg conf) in
  filter
  (fun pp:prog_point c ⇒ if Γ_dec targets (Z_of_nat pp) then true else false)
  (prog_point_list c)
```

```
.

(* Altering the flag is not required: the register is not set here; only the abstract
   knowledge is refined. *)
Definition abstract_goto_ind {c: Program} (pp: prog_point c) (R:Register)
  (conf: ab_config) : ab_config :=
  AbConfig (ab_flags conf)
  (MapInterface.add R (cst_rsi (value_of_pp pp)) (ab_reg conf))
  (ab_mem conf).

Definition abstract_small_step_at (c:Program) (alocs: list Value) (pp:prog_point c)
  (conf: ab_config) : list (State' c * ab_config) :=
match instructionAt pp with
| ICst V R ⇒
   match valid_prog_point_dec c (pp+1) with
   | left H ⇒ cons (Running' (exist _ _ H), abstract_cst V R conf) nil
   | _ ⇒ nil
   end
| IAdd Rs Rd ⇒
   match valid_prog_point_dec c (pp+1) with
   | left H ⇒ cons (Running' (exist _ _ H), abstract_add Rs Rd conf) nil
   | _ ⇒ nil
   end
| ICmp Rs Rd ⇒
   match valid_prog_point_dec c (pp+1) with
   | left H ⇒ cons (Running' (exist _ _ H), abstract_cmp Rs Rd conf) nil
   | _ ⇒ nil
   end
| ISkip ⇒
   match valid_prog_point_dec c (pp+1) with
   | left H ⇒ cons (Running' (exist _ _ H), conf) nil
   | _ ⇒ nil
   end
| IHalt _ ⇒ cons (Stopped', conf) nil
| IGoto l ⇒
   match prog_point_of_value c l with
   | Some p ⇒ cons (Running' p, conf) nil
   | _ ⇒ nil
   end
| IGotoLE l ⇒
   match valid_prog_point_dec c (pp+1), prog_point_of_value c l with
   | left H, Some p ⇒
     cons (Running' (exist _ _ H), abstract_nle conf)
       (cons (Running' p, abstract_le conf) nil)
   | left H, None ⇒ cons (Running' (exist _ _ H), abstract_nle conf) nil
   | right _, Some p ⇒ cons (Running' p, abstract_le conf) nil
   | right _, None ⇒ nil
   end
| IGotoInd R ⇒
  List.map
  (fun l ⇒ (Running' l, abstract_goto_ind l R conf))
  (gotoIndTargets c R conf)
| ILoad Rs Rd ⇒
   match valid_prog_point_dec c (pp+1) with
   | left H ⇒ cons (Running' (exist _ _ H), abstract_load Rs Rd conf) nil
   | _ ⇒ nil
   end
| IStore Rs Rd ⇒
   match valid_prog_point_dec c (pp+1) with
   | left H ⇒ cons (Running' (exist _ _ H), abstract_store alocs Rs Rd conf) nil
   | _ ⇒ nil
   end
end.
```

Bibliography

[Alb+09] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, Germán Puebla, Diana Ramirez, Guillermo Román-Díez, and Damiano Zanardini. "Termination and Cost Analysis with COSTA and its User Interfaces". In: *PROLE'09*. Elsevier, Sept. 2009, pp. 109–121.

[Bal+08] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. "WYSINWYX: What You See Is Not What You eXecute". In: *Verified Software: Theories, Tools, Experiments*. Springer, 2008, pp. 202–213.

[BL09] Sandrine Blazy and Xavier Leroy. "Mechanized semantics for the Clight subset of the C language". In: *Journal of Automated Reasoning* 43.3 (2009), pp. 263–288.

[Bou93] François Bourdoncle. "Efficient Chaotic Iteration Strategies with Widenings". In: *Proc. of the Int. Conf. on Formal Methods in Programming and Their Applications*. Springer, 1993, pp. 128–141.

[BPF11] Sébastien Bardin, Herrmann Philippe, and Védrine Frank. "Refinement-based CFG Reconstruction from Unstructured Programs". In: *VMCAI*. 2011.

[BR10] G. Balakrishnan and T. Reps. "WYSINWYX: What you see is not what you eXecute". In: *ACM Trans. Program. Lang. Syst.* 32.6 (2010), pp. 1–84.

[BS05] Arini Balakrishnan and Chloe Schulze. *Code Obfuscation Literature Survey*. Tech. rep. CS701 Construction of Compilers. University of Wisconsin, Madison, 2005.

[CC04] P. Cousot and R. Cousot. "Basic Concepts of Abstract Interpretation". In: *Building the Information Society*. Kluwer Academic Publishers, 2004, pp. 359–366.

[CC77] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Symposium on Principles of Programming Languages*. ACM Press, 1977, pp. 238–252.

[Coo+09] Kevin Coogan, Saumya K. Debray, Tasneem Kaochar, and Gregg M. Townsend. "Automatic Static Unpacking of Malware Binaries". In: *WCRE*. 2009, pp. 167–176.

[CP10] David Cachera and David Pichardie. "A Certified Denotational Abstract Interpreter". In: *Proc. of Int. Conf. on Interactive Theorem Proving*. Springer-Verlag, 2010, pp. 9–24.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. *A Taxonomy of Obfuscating Transformations*. Tech. rep. 148. The University of Auckland, July 1997.

[CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs". In: *POPL*. Jan. 1998.

[GG09] Y. Guillot and A. Gazet. "Désobfuscation automatique de binaire-The Barbarian Sublimation". In: *SSTIC*. 2009.

[Kru+04] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. "Static disassembly of obfuscated binaries". In: *Proceedings of USENIX Security*. 2004, pp. 255–270.

[KV10] Johannes Kinder and Helmut Veith. "Precise Static Analysis of Untrusted Driver Binaries". In: *FMCAD*. Oct. 2010, pp. 43–50.

[LB08] Xavier Leroy and Sandrine Blazy. "Formal verification of a C-like memory model and its uses for verifying program transformations". In: *Journal of Automated Reasoning* 41.1 (2008), pp. 1–31.

[LD03] Cullen Linn and Saumya Debray. "Obfuscation of executable code to improve resistance to static disassembly". In: *CCS*. ACM, 2003, pp. 290–299.

[Ler06] Xavier Leroy. "Formal certification of a compiler back-end or: programming a compiler with a proof assistant". In: *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '06. New York, NY, USA: ACM, 2006, pp. 42–54.

[Les10] Stéphane Lescuyer. "Conteneurs de première classe en Coq". In: *Journées Francophones des Langages Applicatifs* (2010). URL: http://www.lri.fr/~lescuyer/Containers.en.html.

[Lim11] J. Lim. "Transformer specification language: a system for generating analyzers and its applications". PhD thesis. University of Wisconsin, 2011.

[Mad+06] M. Madou, B. Anckaert, B. De Bus, K. De Bosschere, J. Cappaert, and B. Preneel. "On the effectiveness of source code transformations for binary obfuscation". In: *SERP*. 2006, p. 527.

[Mau04] Laurent Mauborgne. "ASTRÉE: Verification of Absence of Run-Time Error". In: *Building the information Society (18th IFIP World Computer Congress)*. Kluwer Academic Publishers, Aug. 2004, pp. 384–392.

[MVD06] Matias Madou, Ludo Van Put, and Koen De Bosschere. "Loco: An Interactive Code (De)Obfuscation tool". In: *Proceedings of ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*. Charleston, South Carolina: ACM Press, 2006, pp. 140–144.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis*. Springer-Verlag New York Inc, 1999. ISBN: 3540654100.

[PDA07] Igor Popov, Saumya Debray, and Gregory Andrews. "Binary Obfuscation Using Signals". In: *Proc. Usenix Security*. 2007, pp. 275–290.

[Pic05] David Pichardie. "Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés". PhD thesis. Université de Rennes I, Dec. 2005.

[RBL06] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. "Intermediate-representation recovery from low-level code". In: *PEPM '06*. ACM, 2006.

[Rep+10] Thomas Reps, Junghee Lim, Aditya Thakur, Gogul Balakrishnan, and Akash Lal. "There's Plenty of Room at the Bottom: Analyzing and Verifying Machine Code". In: *Computer Aided Verification*. Springer, 2010, pp. 41–56.

[RM10] K.A. Roundy and B.P. Miller. "Hybrid Analysis and Control of Malware". In: *RAID*. Sept. 2010, pp. 317–338.

[SO08] M. Sozeau and N. Oury. "First-class type classes". In: *Theorem Proving in Higher Order Logics* (2008), pp. 278–293.

[Tea10] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Apr. 2010. URL: http://coq.inria.fr/.

[UDM05] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. "Deobfuscation: Reverse Engineering Obfuscated Code". In: *WCRE*. IEEE Computer Society, 2005.

[Wan+00] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. *Software Tamper Resistance: Obstructing Static Analysis of Programs*. Tech. rep. University of Virginia, 2000.

[War03] H. S. Warren Jr. *Hacker's Delight*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.