

Precise Static Analysis of Binaries by Extracting Relational Information

Alexander Sepp Bogdan Mihaila Axel Simon
Lehrstuhl für Informatik 2, Technical University Munich, Germany
firstname.lastname@in.tum.de

Abstract—While the reconstruction of the control-flow graph of a binary has received wide attention, the challenge of categorizing code into defect-free and possibly incorrect remains a challenge for current static analyses. We present the intermediate language RREIL and a corresponding analysis framework that is able to infer precise numeric information on variables without resorting to an expensive analysis at the bit-level. Specifically, we propose a hierarchy of three interfaces to abstract domains, namely for inferring memory layout, bit-level information and numeric information. Our framework can be easily enriched with new abstract domains at each level. We demonstrate the extensibility of our framework by detailing a novel acceleration technique (a so-called widening) as an abstract domain that helps to find precise fixpoints of loops.

I. INTRODUCTION

Security auditing has traditionally been forced to consult the program executables (the “binary”) rather than its source code for finding vulnerabilities. The sheer size of today’s software also leads to more complex systems with more defects. Since the corresponding binaries lack any structure that make a large project manageable, tool support is mandatory to reverse engineer any real-world software. Current tools like IDAPro help with the reconstruction of the control-flow graph (CFG) but often require manual guidance when confronted with optimized or obfuscated code, malware, etc., due to the use of code patterns that can only capture a limited set of compiler idioms. Recent approaches address this shortcoming by inferring possible values of registers and memory locations which helps to resolve most indirect jumps [1], [2], [3]. However, the reconstruction of the CFG is only a first step to finding vulnerabilities. One way forward is the use of a precise and sound static analysis that is able to identify large parts of the code as correct so as to focus the security engineer on potentially vulnerable code fragments. In this paper, we address this concern by presenting a static analysis that can infer precise information on memory regions and their content, thereby helping to prove memory accesses correct. One challenge is the inference of precise numeric information to determine the possible pointer offset. It turns out that many intermediate representations in the literature express the semantics of native instructions at the bit-level [4], [5] which often requires expensive SAT solving techniques to infer numeric range information [6], [7]. We propose to re-use static analysis techniques such as numeric abstract domains (intervals [8], linear equations [9] and inequalities [10]) that have been successfully applied to high-level languages [11].

Employing numeric domains for binary analysis is challenging as assembler instructions operate on bit vectors whose numeric value depends on whether the vector is interpreted as signed or unsigned integer. In particular, certain instructions such as `add` or `shl` have the same bit-level semantics regardless of whether their arguments are signed or unsigned. In contrast, a relational test $a \leq b$ translates into different assembler instruction sequences, depending on the signedness of a, b . We present an analysis framework that maps native assembler into a small intermediate language (IL) called RREIL that is carefully crafted to allow for an easy recovery of numeric information, especially from relational tests and conversions between differently sized integers. RREIL is then translated into an IL on bit-vectors which is, in turn, translated into an IL on variables over \mathbb{Z} . An abstract domain that implements an analysis thus implements an interface in form of one of these ILs. For example, an interval domain implements the IL over \mathbb{Z} while a bit-level analysis based on SAT-solving would implement the IL over bit vectors. We argue that this hierarchical approach allows for both, efficiency and precision.

While the use of numeric domains is relatively cheap, termination of fixpoint calculations relies on widening and narrowing [12]. We argue that narrowing interacts badly with the reconstruction of the CFG and show how to refine it.

Due to over-approximation, our analysis may warn about invalid memory accesses that cannot occur in the actual program; a so-called false positive. Our framework therefore features a plug-in architecture in which auditors can integrate special-purpose domains specific to their analysis needs.

In summary, this paper contributes the following:

- the RREIL intermediate language that allows for a concise translation of complex architectures (e.g. x86-64) into precise numeric domain operations
- a domain hierarchy that separates different concerns of an analysis and which may be extended by the end user through a plug-in architecture
- a novel way to perform widening without undue loss of precision that is particularly suitable for analyses where the CFG is not known a priori

The paper is organized as follows. The next section details the grammar and motivation of our RREIL intermediate language. Section III describes the different domain types of our analysis. Section IV details our novel widening for binaries before Sect. V presents an evaluation. Finally, related work is presented in Sect. VI before Sect. VII concludes.

II. DESIGNING AN INTERMEDIATE LANGUAGE FOR RELATIONAL ANALYSIS

Mobile devices and other specialized architectures like PLCs are increasingly used in security critical contexts. Automatic analysis of these diverse systems requires tools to abstract from the instruction set of each individual architecture by using an intermediate language (IL) since simulating the semantic effect of each instruction of each platform becomes infeasible. In fact, even when targeting a single platform such as Intel x86, the semantic effects of over 600 instructions would have to be simulated. However, each x86 instruction usually assembles several distinct effects similar to a macro. Not unlike the decomposition into micro-ops within the processor, each instruction is translated into a small intermediate language (IL) with simple semantics. The next sections motivate the design of a new intermediate language called RREIL.

A. Translation of Comparisons

One goal in the design of such intermediate languages has been to minimize the number of instructions of the IL required to represent all native instruction [4], [5]. A small IL promises a simple analysis, however, small ILs often require clever bit-shifting and bit-masking operations to model a native instruction. Consider the following translation of the x86 compare instruction `cmp eax, ebx` followed by a conditional jump `j1` (jump if less) into the REIL intermediate language [5] which was the starting point for our RREIL language:

```

00: and  t0:32, 2147483648:32, eax:32
01: and  t1:32, 2147483648:32, ebx:32
02: sub  t2:64, ebx:32, eax:32
03: and  t3:32, 2147483648:32, t2:64
04: bsh SF:8, -31:32, t3:32
05: xor  t4:32, t1:32, t0:32
06: xor  t5:32, t3:32, t0:32
07: and  t6:32, t5:32, t4:32
08: bsh OF:8, -31:32, t6:32
09: and  t7:64, 4294967296:64, t2:64
0A: bsh CF:8, -32:32, t7:64
0B: and  t8:32, 4294967295:64, t2:64
0C: bisz ZF:8, t8:32
0D: xor  t9:8, OF:8, SF:8
0E: jcc t9:8, signed_less:32

```

This translation converts the compare instruction into a subtraction `sub` that stores its intermediate result in the temporary register t_2 . Various bit operations then calculate the sign flag $SF = t_2[31]$, overflow flag $OF = (eax[31] \oplus ebx[31]) \wedge t_2[31]$, carry flag $CF = t_2[32]$, and zero flag $ZF = \neg(t_2[0] \wedge \dots \wedge t_2[31])$. Here, `bsh` is a bit shift and `bisz` corresponds to the not operator `!` of C. Suppose that the intention was to test if $eax < ebx$ where eax and ebx hold signed integers. In this case the subsequent instruction is a conditional jump testing if $SF \oplus OF$ is one. This translation is at odds with the need of common static analyses that infer numeric information on program variables.

These analyses require explicit relational tests to infer precise bounds on program values that are required to find, for instance, buffer overflows. The bit-level semantics render the extraction of these relational tests difficult: recovering the semantics $eax < ebx$ of the tests requires the analysis to recover the arithmetic semantics implemented by `cmp` by tracing the bit-level calculation of the flags. The implicit assumption of using bit-level semantics in intermediate languages seems to be that SAT solving is to be used to reason about the values of variables. However, using SAT solving to perform a reachability analysis (in contrast to inferring single traces) faces scalability problems as soon as non-trivial loops are analyzed. Even when combining SAT solving with abstraction to numeric ranges [6], the resulting analyses only perform well on embedded code where registers are 8 bit wide. Thus, our aim is to obtain a scalable analysis by applying the arithmetic semantics implemented by `cmp` directly on some numeric abstract domain. One way forward would be to pattern match the REIL instruction sequences that implement `cmp r1, r2; j1`, `cmp r1, r2; jg`, etc. However, this approach is architecture-specific as each instruction set is translated differently. Furthermore, it fails if an optimizing compiler inserts instructions between the test and the jump or if malicious code is considered.

We therefore propose a new variant of REIL [5] called RREIL (Relational Reverse Engineering Intermediate Language) in which flag calculations are translated into arithmetic instructions if possible. For instance, `cmp eax, ebx` followed by `j1` translates into RREIL as follows:

```

00: sub    t0:32, eax:32, ebx:32
01: cmpltu CF:1, eax:32, ebx:32
02: cmpleu CForZF:1, eax:32, ebx:32
03: cmplts SFxorOF:1, eax:32, ebx:32
04: cmple  SFxorOFforZF:1, eax:32, ebx:32
05: cmpeq  ZF:1, eax:32, ebx:32
06: cmplts SF:1, t0:32, 0:32
07: xor    OF:1, SFxorOF:1, SF:1
08: brc    SFxorOF:1, signed_less:32

```

The comparisons in lines 1 to 6 test for, $<$, \leq of unsigned values, $<$, \leq for signed values, equality and if the result is negative, respectively. The translation creates what we call virtual flags, that is, flags that are not present in the processor but which express relational information that can be conveyed to numeric domains. For instance, $SFxorOF$ is set iff $eax < ebx$ when eax and ebx are interpreted as signed integers. The force of this approach is that each arithmetic test is associated with one flag and, once the conditional branch is reached, the test matching the flag can be applied to the numeric abstract domain. Observe that the overflow flag OF carries no meaningful arithmetic information but is still calculated to allow bit-level analyses such as SAT solving.

Note that the unsatisfactory translation of relational tests into REIL is only one reason for our new IL RREIL. Two

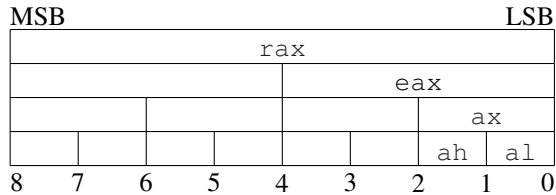


Figure 1. The possible fields of register `rax`.

other differences, namely that non-Boolean arguments to an instruction `sub`, `cmpltu`, etc., must have the same size and may be accessed at an offset, implies that our analysis cannot analyze REIL code. Thus, RREIL is not an extension of REIL; rather, it is a new IL whose features are carefully chosen to allow for a precise numeric analysis of the resulting RREIL code, as detailed in the following sections.

B. Fields in Registers

A thorough understanding of a binary requires the recovery of memory layout, that is, the location and sizes of variables on the stack, heap and global memory. Indeed, many analyses incorporate some sort of memory layout inference [13]. The idea is to identify sequences of bytes that are modelled by a single numeric variable in some abstract domain. We extend this layout inference to registers, thereby enabling the access to registers with different sizes and offsets which is common on the x86-32 and x86-64 platforms. One particular oddity of the x86-64 architecture is the implicit zero-extension when writing the lower 32 bits of a register. For instance, the instruction `dec eax` has the effect of setting the upper 32 bits of `rax` to zero and the lower 32 bits to `eax - 1`. The translation into RREIL of this instruction (without the calculation of flags) is as follows:

```
00: sub rax:32, rax:32, 1:32
01: mov rax:32/32, 0:32
```

Here, the notation `rax:s/o` is an access to `s` bits of `rax` starting at bit offset `o`. In principle, our analysis can infer fields at any bit offset and of any size. However, the translations of common register instructions will lead to byte aligned fields as shown in Fig. 1. Inferring these fields by tracking accesses has been described in the context of analyzing C programs [14], [15]. For the sake of an example, suppose that the analysis has inferred the interval $[-2, 2]$ for `eax`. The analysis of the two instructions above will infer the interval $[-3, 1]$ for the lower half of `rax` while the interval $[0, 0]$ is inferred for the upper half of `rax`. In contrast, if no fields are inferred for registers and a single numeric variable is used to model the content of `rax`, the following translation could be used:

```
00: sub t0:32, rax:32, 1:32
01: zero-extend rax:64, t0:32
```

Here, the fictitious instruction `zero-extend` calculates the effect of zero-extending the 32-bit value `t0` to `rax`.

However, the best interval for `rax` in the example would be $[0, 2^{32} - 4]$, that is, nearly the full 32-bit unsigned range. Other, more precise, numeric domains such as polyhedra [10] will incur a similar precision loss. Thus, modelling registers as a set of fields leads to an improved precision when employing numeric abstract domains. This becomes important for vector instructions which can be translated into a sequence of RREIL instructions that access the vector register at different offsets.

A challenge arises in the analysis when converting between differently sized fields as it becomes difficult to retain precise numeric information. For instance, consider a function `f` returning `unsigned int`. The call `unsigned long n = f()` followed by `n -= 3` translates into the following Intel x86-64 instructions (where the compiler has chosen to store `n` in the register `eax`):

```
00: call f
01: mov  eax,  eax
02: sub  rax,  3
```

Here, the second statement implicitly sets the upper part of `rax` to zero which gives rise to the following RREIL code:

```
00: call f
01: mov  rax:32/32, 0:32
02: mov  rax:32,  rax:32
03: sub  rax:64,  3:64
```

When evaluating the `sub` instruction, the analysis finds two 32-bit fields that constitute the 64-bits that are to be read. Depending on the values of the two fields, a new 64-bit field can be synthesized that holds a precise numeric value. For instance, given an upper field whose value is tracked by `u` and a lower field tracked by `l` and whose value lies in $[0, 2^{32} - 1]$ a new 64-bit field `c` can be created with $c = 2^{32}u + l$. In the example, since $u = 0$, this reduces to $c = l$. More propagation rules are developed in [15, Chap. 5].

Although SAT solving can infer range information, these approaches are less precise than our field-based approach [16, Ex. 4] nor do they scale [17]. Scalability was one aspect in the design of RREIL as detailed next.

C. Reducing the Size of RREIL Programs

A major disadvantage in the analysis of binaries is the output size of the translation into the IL. As illustrated in Sect. II-A, a simple comparison translates into no fewer than eight RREIL instructions. Moreover, the translation of arithmetic instructions creates similarly many assignments to flags, even though these are usually not tested thereafter. Many ILs that express the semantics at the bit-level create even more instructions; consider, for instance, the corresponding REIL translation of the test with 15 instructions. A simple liveness analysis followed by a dead code removal can often eliminate the majority of instructions as their calculated result is never used. For instance, in the introductory example 12 REIL instructions and only 2 RREIL instructions remain. Moreover, due to the use of fields in the RREIL language,

liveness analysis can be performed for each field (rather than for the whole register), thereby also removing most of the RREIL instructions generated for the x86-64 instruction set that model the implicit zero extension of 32-bit arithmetic instructions. Since liveness is a backward analysis, full dead code removal can only be done once the complete CFG of a function is known. However, a partial removal is possible by assuming that all variables are live after an unresolved branch. The intent is to allow for memory-intensive analyses on the binary that are already possible at the C level [11].

D. A Formal Definition of RREIL

We conclude the discussion of translating assembler to RREIL by a formal definition of the RREIL syntax. A precise definition is instructive not only for comparisons with other languages, but also in illustrating how RREIL instructions are broken down into simpler statements that relate to memory regions, fields of memory regions and their numeric content. We commence by presenting the RREIL instructions that concern the read and write access of a memory cell, a conditional branch, a function call and return instruction (which are jumps semantically). In all statements, information travels from right to left, following the Intel convention.

```

rreil ::= load lval, lval
        | store lval, rval
        | brc condition, rval
        | call rval
        | return rval
        | base-stmt

```

The right-hand sides of each instruction use the definition of a writable location $lval$ and an $rval$, i.e. an expression or interval where $\mathbb{Z}_\infty = \mathbb{Z} \cup \{-\infty, \infty\}$. We use the term memory region for $id \in \mathbb{M}$ which encompasses the set of registers and temporary variables that are generated during translations. Each memory region carries the size of the access which must be the same across an instruction. One of two exceptions to this rule are *flags* that are always one bit wide. Memory regions may carry an optional bit offset written */sz*.

```

condition ::= flag | c            $c \in \mathbb{N}$ 
lval ::= var
rval ::= var | [l, u] : sz     $l, u \in \mathbb{Z}_\infty$ 
var ::= id : sz (/sz)?          $id \in \mathbb{M}$ 
flag ::= id : 1                  $id \in \mathbb{M}$ 
sz ::= n                         $n \in \mathbb{N}$ 

```

These definitions are also used in the rules for arithmetic and comparisons which are summarized in the *base-stmt* production, presented next. The *base-stmt* production encompasses binary operations, comparisons, sign extension and the copying of values. The binary instructions must be applied to arguments of equal size. Only the arguments of **sign-extend** feature different sizes. The comparison statements test for

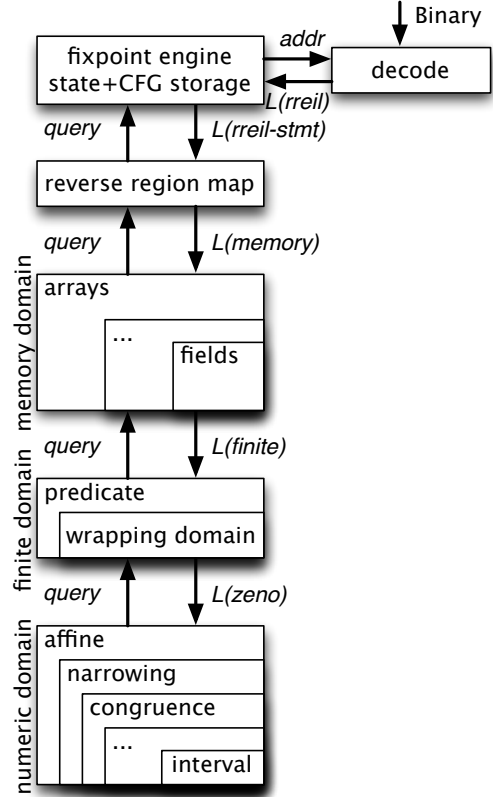


Figure 2. The analyzer's domain hierarchy

equality, for “less or equal” on unsigned and signed values and for “less than” on unsigned and signed values, respectively. Both arguments have the same size. Greater-than comparisons are translated by swapping their arguments.

```

base-stmt ::= binop lval, rval, rval
             | cmpop flag, rval, rval
             | sign-extend lval, rval
             | mov lval, rval
binop ::= add | sub | mul
         | div | divs | shl | shr | shrs
         | mod | and | or | xor
cmpop ::= cmpeq | cmpleu | cmples
         | cmpltu | cmplt

```

While the 24 RREIL instructions make for a concise language, the previous REIL language was even simpler, featuring only 17 instructions. As illustrated in the next section, the RREIL language is translated several times during the analysis, each time turning into a simpler language. Since most end-user implemented domains will use one of the simpler languages, we believe that our design is in fact easier on third-party developers than the original REIL instruction set.

III. COMBINING ABSTRACT DOMAINS

A static analysis of executables differs from the analysis of high-level languages in that the CFG is not known a priori. The CFG, in turn, is necessary to propagate states between basic blocks and to infer that a fixpoint is reached in each loop. CFG reconstruction requires the precise inference of possible jump targets for indirect jumps and calls. In the case of **switch**-statements, the generated jump tables can often be resolved by tracking a finite set of values, a so-called value-set analysis. Our aim, however, is to implement more sophisticated analyses that are required to assert, for instance, the correctness of ordinary pointer accesses such as those to stack-local arrays which are of particular interest to the security auditor. Tracking more precise information and, if required, user-defined information, requires a modular approach to static analysis with the option of adding new abstract domains in the form of a plug-in. Our approach to a modular framework follows the two-tier approach of Astrée [18] but extends it to four tiers to suit the need of the analysis of executables. The original two tiers are the *memory domain* and *numeric domain* in Fig. 2. The additional two tiers are the *reverse region map* and the *wrapping domain* that are detailed below. The interface to each tier is defined by a grammar with RREIL being the interface to the first two tiers. Since RREIL allows a calculated address in memory accesses and indirect jumps, we create memory regions $m \in \mathbb{M}$ and associate a consecutive range of addresses with m that form a semantic entity such as a stack frame, heap allocated memory, a basic block in the code segment or a constant segment. An RREIL operation on pointers is translated into an operation on a memory region which is the task of the reverse region map. In the future it will be responsible for summarizing memory regions (such as heap regions or stack frames of recursive calls). We now present the operations on memory regions.

A. Memory Domain

Let \mathbb{X} be a set of abstract variables, that map to numeric values in the current state. The purpose of the memory domain is to associate ranges of bits of $m \in \mathbb{M}$ with an abstract variable $x \in \mathbb{X}$. For instance, the register $rax \in \mathbb{M}$ may have a variable $x_l \in \mathbb{X}$ associated with the bits 0–31, thereby representing the value of eax and a variable $x_h \in \mathbb{X}$ that represents bits 32–63. Creating fields at fixed offsets is the task of the *field* domain in Fig. 2. While registers can only be accessed at constant offsets, the interface $L(\text{memory})$ also allows accesses at calculated offsets.

$$\begin{aligned}
 \text{memory} &::= \text{base-stmt} \mid \text{mem-test} \mid \text{mem-sup} \\
 &\quad \mid \text{lval} = \text{region} \xrightarrow{sz} \text{offset} \\
 &\quad \mid \text{region} \xrightarrow{sz} \text{offset} = \text{rval} \\
 \text{mem-test} &::= \text{flag} \doteq 0 \mid \text{flag} \neq 0 \\
 \text{mem-sup} &::= \mathbf{intro} \text{ region} \mid \mathbf{drop} \text{ region} \\
 \text{region} &::= \underline{m} && m \in \mathbb{M} \\
 \text{offset} &::= \underline{x} \mid sz && x \in \mathbb{X}
 \end{aligned}$$

Here, the C-like sz -bit pointer dereference $m \xrightarrow{sz} o$ accesses $m \in \mathbb{M}$ at an offset $o \in \mathbb{X}$. If o maps to a non-constant value, that is, an interval $[l, u]$, $l \neq u$, the *field* domain will remove all fields overlapping with bits $[l, u + s - 1]$ with s being the access size from $lval$ or $rval$. Other domains are conceivable such as the *array* domain in Fig. 2 which summarizes accesses to a range of offsets with a fixed stride into a single array cell. The figure shows the *array* domain as a parent domain of the *field* domain which means that it can execute any operation on the *field* domain while evaluating $m \xrightarrow{sz} o$. (Formally, the array domain A and the field domain F form a cardinal power domain F^A [12, Sect. 10.2].) In order to illustrate the interaction between *array* and *field*, consider the following loop:

```

struct { short l, h; } m[100];
for (short i=0; i<100; i++)
  { m[i].l = 0; m[i].h = i; }

```

The loop is first analyzed with $i = 0$ so that the *field* domain will create two fields as a response to $m \xrightarrow{sz} 16 = i$ and $m \xrightarrow{sz} 0 = 0$. The second iteration of the fixpoint computation will execute the loop body with $m \xrightarrow{sz} o = i$ where $o \in \{16, 48\}$, that is, $o = 16 + 32i$, $i \in [0, 1]$. The *array* domain will observe the non-constant offset and intercept the assignment before the *field* domain removes all information on the inferred fields. Instead, it copies all fields at bits 0–31 to a new summary memory region m' using an assignment operation on the *field* domain. The write operations on the i th element are then translated to an operation on m' at offset $o - 32i$ which is constant, thereby re-using the capabilities of the *field* domain, to handle e.g. overlapping fields. The ability to re-use existing domains is a major benefit in this hierarchical arrangement. For instance, an end-user might plug-in other domains such as one tracking the zero characters in a C string buffer [19] which may, in turn, re-use the *array* domain.

The remaining operations on the memory domain test flags (thereby implementing the semantics of conditional branches) and change the support set (the set of mapped variables) by introducing and removing memory regions using **intro** and **drop**, respectively.

B. Finite Numeric Domains

The memory domains associates an abstract variable $x \in \mathbb{X}$ with each field inferred by the *field* domain or with the size of an array. The underlying domains map x to possible numeric values. Each abstract variable represents a finite number of bits and the operations emitted by the memory region always access all bits in a variable. The grammar of the interface $L(\text{finite})$ between memory domains and the *wrapping* domain therefore does away with offsets and, in particular, accesses at non-constant offsets. To this end, it makes use of a production $\text{base-stmt}^{\mathbb{X}}$ that correspond to productions base-stmt with $\text{var}^{\mathbb{X}} ::= \underline{x}$ and $\text{flag}^{\mathbb{X}} ::= \underline{x}$ substituted for the corresponding var and flag rules.

$$\begin{aligned}
finite &::= base\text{-}stmr^{\mathbb{X}} \mid fin\text{-}test \mid fin\text{-}sup \\
&\quad \mid \mathbf{convert} \ var^{\mathbb{X}} \ var^{\mathbb{X}} \\
fin\text{-}test &::= flag^{\mathbb{X}} \doteq \underline{0} \mid flag^{\mathbb{X}} \neq \underline{0} \\
fin\text{-}sup &::= \mathbf{intro} \ sz \ var^{\mathbb{X}} \mid \mathbf{drop} \ var^{\mathbb{X}}
\end{aligned}$$

Note that each variable $x \in \mathbb{X}$ in the *finite* domain has a fixed bit size. This size is specified when creating a variable using **intro**. Analogous to the requirement in RREIL that every operation must use the same size for all arguments, each operation in $L(finite)$ may only use variables of equal size. Exceptions are the result of comparisons, the **sign-extend** operation and the new operation **convert**. The latter is used to strip off most significant bits or to zero-extend a variable.

In order to infer strong relational invariants, such as inequality relations between variables, we translate operations over \mathbb{Z}_{2^w} to abstract domains over \mathbb{Z} . Since some operations such as **add** and **shl** carry no signedness information, it is not possible to simply check for overflow after each assignment. Thus, we interpret each value $v \in \mathbb{Z}$ of $x \in \mathbb{X}$ in the numeric domain as $v \bmod 2^w$ in the finite domain, assuming that x was introduced as having w bits. In this interpretation, no checks for overflow are required for linear numeric transformations [20]. Linear transformations, in turn, are exactly those assembler instructions that carry no sign information. As an example, consider an unsigned 4-bit variable $x \in \mathbb{X}$ for which the numeric domain tracks an interval $[13, 15]$ corresponding to the finite interpretations $1101_b, 1110_b, 1111_b$. A left-shift by one bit is a multiplication by two and leads to the interval $[26, 30]$. Interpreting these values $\bmod 2^4$ yields the bit patterns $1010_b, 1011_b, 1100_b, 1101_b, 1110_b$ which is a sound approximation of the possible set of bit-patterns (which excludes the second and forth pattern).

The arguments of relational tests and other operations whose semantics depend on sign information (e.g. **shr** and **shrs**) are checked if their values lie in the range of the corresponding signed or unsigned w -bit integer. If not, they need to be adjusted which might involve further approximation [20]. These translations are performed by the *wrapping* domain shown in Fig. 2 which itself executes required operations on the \mathbb{Z} domains. Recently, the use of SAT solvers has become increasingly popular in the context of synthesizing transfer functions [17]. Such solutions can be implemented against the $L(finite)$ interface and may query the (wrapped) range of variables through the wrapping domain. In particular, this setup allows for expensive, SAT-based bit-level analyses to only track variables that participate in bit-level operations and leave the tracking of other variables to the numeric domains.

C. Numeric Domains over \mathbb{Z}

The interface to the numeric domains features standard operations on variables ranging over \mathbb{Z} and is given by the language $L(zeno)$, defined as follows:

$$\begin{aligned}
zeno &::= zeno\text{-}stmt \mid zeno\text{-}test \mid zeno\text{-}sup \\
zeno\text{-}stmt &::= lval = lin \ binop \ lin \\
&\quad \mid lval = zeno\text{-}test \\
&\quad \mid lval = lin / d \quad d \in \mathbb{N} \\
&\quad \mid lval = [l, u] \quad l, u \in \mathbb{Z}_{\infty} \\
zeno\text{-}test &::= lin \ relop \ \underline{0} \\
zeno\text{-}sup &::= \mathbf{intro} \ lval \mid \mathbf{drop} \ lval \\
binop &::= \times \mid \div \mid \% \mid \gg \\
relop &::= \doteq \mid \neq \mid < \mid \leq \\
lval &::= \underline{x}, \quad x \in \mathbb{X} \\
lin &::= \sum_i a_i x_i + c \quad \begin{array}{l} x_i \in \mathbb{X} \\ a_i, c \in \mathbb{Z} \end{array}
\end{aligned}$$

The language is comprised of statements, tests and instructions to modify the set of mapped variables. A statement sets a variable to the result of a binary operation, a test, a linear operation with divisor d or an interval. No bit-level binary operators are allowed as they cannot reasonably be expressed using numeric domains that model a convex state space. While common bit-level identities such as **xor** $x, x \equiv 0$, are substituted in the wrapping domain, usually only an interval approximation is substituted, e.g. **and** $x, y \equiv 0$ if $x = [0, 0]$ or $y = [0, 0]$ in the numeric domain.

A test **cmpleu** f, x_1, x_2 translates to $f = x_1 - x_2 \leq 0$ whenever the current values of x_i lie within their unsigned range. Otherwise, the result is approximated to e.g. $f = x_1 - x_2 - 256$ in the case that $x_2 \in [0, 255]$ and $x_1 \in [256, 511]$ or to $f = [0, 1]$ in the worst case.

The simplicity of the numeric interface allows for a modular combination of domains described in the literature. One interesting variation is our *affine* domain in Fig. 2 that tracks equalities in a normal form of an upper-triangular matrix using some fixed ordering on the variables \mathbb{X} . When calculating the join of two domains with n rows, the normal form makes it possible to extract only the m equalities that differ, which is possible in $O(m)$ rather than $O(|\mathbb{X}|)$ [11]. Furthermore, if an equality $x = y + z + 5$ is stored then the variable x is fully determined and thus does not have to be stored in the child domains, leading to a smaller support set of the underlying domains *narrowing ... interval*. This factoring trick also simplifies many linear operations. For instance, the assignment $x = x + 1$ is evaluated by inlining existing equalities into the right-hand side, yielding $x = y + z + 6$, and replacing the previous definition of x . Since x is only known in the *affine* domain, no operation on the child domain is necessary. Since binary programs exhibit many equality relations (e.g. between registers and fields on the stack) our implementation of the well-known affine domain [9] thereby improves the performance of the analysis.

Since abstract domains tracking intervals and congruences [21] are well-known, we turn to the *narrowing* and *predicate* domains that allow the inference of precise loop bounds.

IV. INFERENCE OF NUMERIC LOOP INVARIANTS

The driving force behind our framework is the precise numeric analysis of variable valuations in order to classify the majority of memory access as correct, thereby alleviating the need for further inspection by the security auditor. One challenge in the analysis of binaries is that conditionals in high-level programs are translated into a test setting certain flags and a conditional jump. One way to link test and jump is to perform forward substitution of the flag assignments into the conditional jump [2]. This approach fails when the arguments to the test are modified before the conditional jump, a situation that may arise as a result of a compiler optimization. One fix is to apply any modification to the tested register to the relational test that is being propagated forward. Our analyzer pursues this approach in a less ad-hoc way by tracking all observed tests in a symbolic abstract domain dubbed *predicate* in Fig. 2. In order to illustrate this process, consider the following RREIL statements:

```
01: cmpltu CF:1, eax:32, 64:32
02: add    eax:32, eax:32, 1:32
03: brc    CF:1, loop:32
```

The purpose of the predicate domain is to track assignments to flags symbolically, thus, it stores $CF = eax < 100$ after the first statement. The second statement increments a variable that is mentioned in the symbolic expression, thereby rendering this expression invalid. Rather than removing all information, any linear assignment and transformation is simply applied to the symbolic information, leading to $CF = eax < 101$ in the example. Once the conditional jump leads to test $CF \doteq 1$ on the memory domain, the predicate domain also executes the test $eax < 101$ on its child domains, thereby making the effect of the indirect test explicit.

A. The Widening/Narrowing Approach

One challenge when inferring bounds of variables is the treatment of loops. In order to ensure termination, the classic approach is to apply widening, possibly followed by narrowing [10]. We illustrate the widening/narrowing approach by extending the example above to a loop that implements the C statement `for(int i=0; i<100; i++) {/*body */}`:

```
000: mov    eax:32, 0:32
001: /* body */
n+1: cmpltu CF:1, eax:32, 64:32
n+2: add    eax:32, eax:32, 1:32
n+3: brc    CF:1, 001:32
```

The basic block starting at address 001 has two incoming edges, namely one from address 000 where $eax = 0$ and one from n+3 where $eax = 1$ after the first iteration. Joining these states as intervals leads to $eax = [0, 1]$. The idea of widening is now to enforce a fixpoint by extrapolating the change at address 001. From the states $eax = [0, 0]$ in the first and

$eax = [0, 1]$ in the second iteration, widening discards the upper bound since it is unstable such that $eax = [0, \infty]$. A further iteration with this state leads to $eax = [0, 100]$. Rather than merely joining this state to the widened state $eax = [0, \infty]$, the analyzer replaces this state with $eax = [0, 100]$, a process called narrowing. Although, this approach often leads to good (post-)fixpoints, it has the disadvantage that any statements in the loop body (stored at addresses 001 to n) are analyzed at least once with $eax = [0, \infty]$ which can lead to imprecise results [22]. Worse, at $n + 1$ the wrapping domain will check if eax is in the range of a signed integer and set eax to $[-2^{31}, 2^{31} - 1]$ since this is not the case. The latter implies that the analysis would not be able to even infer that eax is positive as it assumes that it may wrap. Further show-stoppers are indirect jumps to addresses read from an array that is indexed using eax . Analyzing such a jump with $eax \in [-2^{31}, 2^{31} - 1]$ requires the CFG reconstruction to ignore this branch which would constitute an unsound under-approximation.

B. Tracking Redundant Tests for Narrowing

In order to avoid the analysis of the loop body with a temporarily over-approximated state, we include a *narrowing* domain in the hierarchy shown in Fig. 2 that is able to restrict the numeric state immediately after widening. The key principle is to track all tests being evaluated, that is, all tests that the *predicate* domain performs on its child domain as a result of a flag being tested. In the example above, the predicate domain translates the test of the carry flag $CF \doteq 1$ to the test $eax - 101 < 0$ on the underlying *narrowing* domain. Each test is added to a set of observed tests \mathcal{O} , in the example, $\mathcal{O} = \{eax - 101 < 0\}$. For the state in which the conditional branch would not be taken, the predicate domain translates $CF \doteq 0$ to $eax \geq 101$ which is added to the set \mathcal{O} of the *narrowing* domain at $n + 4$. As in the predicate domain, linear transformations are applied to the tests in \mathcal{O} while other transformations that affect an observed test $o \in \mathcal{O}$ will discard o . In other words, the invariant of the relations in \mathcal{O} is that they hold in the current state.

In order to retain this invariant when joining states at points where the control flow merges, the join operation of the *narrowing* domain checks all tests that do not hold on both branches. Each such test $lin\ relop\ Q$ is checked if it holds in the other domain. If so, the test is carried over into the joined state. Reconsider the merge of the control-flow at address 001. Here, two test sets $\mathcal{O}_1 = \emptyset$ and $\mathcal{O}_2 = \{eax < 101\}$ are to be joined. The result $\mathcal{O}' = \{eax < 101\}$ contains the test $eax < 101$ as it holds after address 000 since $eax = 0$.

For each cycle in the CFG, the fixpoint engine applies widening instead of join in order to ensure termination. The difference to the join operation is that domains that model state that can increase indefinitely (such as intervals which may map a variable to the states $[0, 0]$, $[0, 1]$, $[0, 2]$, etc.) have to extrapolate this change in a way that infinitely increasing chains of states are no longer possible. When widening the *narrowing* domain, it will first widen its child domain and afterwards apply all tests in \mathcal{O} to the child domain. In the

example, the interval domain is asked to widen the states where eax is $[0, 0]$ and the state where it is $[0, 1]$ which results in an interval domain with eax being $[0, \infty]$. Applying the tests in $\mathcal{O}' = \{eax < 101\}$ restricts the interval domain to eax being $[0, 100]$. In the next loop iteration, a fixpoint will be found with $eax \leq 99$ within the loop body and $eax = 100$ at address $n + 4$. Thus, the novel tracking of observed tests for narrowing can avoid the problematic precision loss that is especially damaging in the analysis of binaries.

V. EXPERIMENTAL EVALUATION

We have implemented the presented analysis framework in Java with a front-end for x86-32 and x86-64 with ARM and AVR front-ends in progress. In contrast to other analyzers [3], [23] that build a single graph, we store a separate CFG for each function. A new function is generated when a **call** instruction is followed. This graph is augmented using a recursive descent strategy until a **ret** instruction is encountered. Thus, a jump into the body of a different function or into the middle of an already decoded instruction augments the graph. A jump to the address of an already decoded instruction of the current function merely creates a new edge. Our tool is able to exactly decode obfuscation examples given in [3], [24] in which the executed instructions are not those that e.g. IDA Pro decodes. The example in [24] shown as *ret-obfus* in Table I alters the return address thereby violating the assumption that execution resumes after a **call** instruction. The example in [3], dubbed *insn-alias*, decodes a **mov** instruction consisting of 5 bytes at an address a and later indirectly jumps to $a + 1$ which then decodes a **call** instruction.

While a static analysis necessarily over-approximates the set of possible jump targets, our tool is precise enough to infer exact targets for each indirect jump in the examples. Other heuristic-based approaches like IDA Pro or the machine learning approach in [24] not only over-approximate the possible targets but may also miss jump targets thereby potentially hiding the presence of malicious behavior which, in turn, hampers the analysis of malware.

The examples *msum* and *asum* in Table I perform matrix and array element summation. They evaluate our narrowing that is implemented as part of the numeric domain hierarchy. In both examples the analyzer recovered precise bounds for all index variables occurring in the (nested) loops.

All times in Table I were taken on a 2.4 GHz Core i5 machine running Linux. Although these examples are rather small, they still demand sophisticated analyses techniques and therefore demonstrate the abilities of our framework. Each double column in the table shows the analysis time, the memory usage, the number of RREIL instructions and the number of original x86 instructions. The starred columns show the effect of the liveness analysis on the RREIL translation. Dead-code removal indeed halves the number of RREIL instructions while the analysis time reduces considerably more (except for test case *mul*). Although considerable care has been taken to reduce the impact of JVM start-up time, the running times should be taken with a grain of salt.

A. Call Obfuscation Example

In order to demonstrate the utility of our tool in the context of understanding executables, we show how the static analysis of the program state helps in the reconstruction of the CFG. The following X86-32 assembler fragment demonstrates a simple **call/return**-obfuscation. The employed technique is known as return-oriented programming [25].

When analyzing the example, it turns out that the instructions at addresses 0b and 10 are actually unreachable. Providing a correct disassembly requires the inference of numeric information and the memory layout of the processors stack to determine the follow-up instruction after the call at address 06. Disassemblers that use a simple linear sweep approach and the heuristic that execution will continue after a **call** instruction will produce the following instruction sequence:

```
// main
00: 8d 35 14 00 00 00: lea esi, [14]
06: e9 06 00 00 00   : call 11
// the following two instructions are
// unreachable and could even be data
0b: b8 02 00 00 00   : mov  eax, 2
10: c3                : ret
// trampoline
11: 58                : pop  eax
12: 56                : push esi
13: c3                : ret
// callee
14: b8 01 00 00 00   : mov  eax, 1
19: c3                : ret
```

The first instruction at offset 0 saves the address of the callee to register *esi*; the second instruction calls the trampoline function that passes control-flow to the pointer in *esi*. The real call to callee is hidden in a sequence of **pop**, **push** and **ret** instructions. The following listing presents the correct disassembly, with basic blocks that form a function grouped together (as done by our analysis):

```
// main
00: 8d 35 14 00 00 00: lea esi, [14]
06: e9 06 00 00 00   : call 11
// (inlined) callee
14: b8 01 00 00 00   : mov  eax, 1
15: c3                : ret
// trampoline
11: 58                : pop  eax
12: 56                : push esi
13: c3                : ret
```

The translation to RREIL code makes every side-effect of native instructions visible and, thus, we will present the translation after the removal of dead code. RREIL addresses are encoded using the special notation $x.y$ where x denotes the underlying native instruction's address and y the RREIL sub-offset (one native instruction translates to several RREIL

benchmark	time (ms)		memory (Mb)		RREIL Instr.		x86 Instr.
	*	*	*	*	*	*	
ret-obfus	23	10	8	9	39	18	9
insn-alias	27	14	15	15	91	56	32
msum	209	68	35	27	157	48	26
asum	43	15	23	19	71	24	13
isquare	136	25	18	13	43	10	8
mul	348	232	211	180	3704	1094	527

Table I
ANALYZED EXAMPLES

instructions). Using sub-offsets allows for defining loops that are required for the translation of X86 instructions using the `repz` and `repnz` prefixes. The example translates into the following RREIL code which is discussed below:

```
// main
00.00: mov    esi:32, 14:32
06.00: sub    esp:32, esp:32, 4:32
06.01: store  esp:32, 0b:32
06.02: call   11:32
// (inlined) callee
14.00: mov    eax:32, 1:32
19.00: load   t0:32, esp:32
19.01: add    esp:32, esp:32, 4:32
19.02: return t0:32
// trampoline
11.00: load   t0:32, esp:32
11.01: add    esp:32, esp:32, 4:32
11.02: mov    eax:32, t0:32
12.00: mov    t0:32, esi:32
12.01: sub    esp:32, esp:32, 4:32
12.02: store  esp:32, t0:32
13.00: load   t0:32, esp:32
13.01: add    esp:32, esp:32, 4:32
13.02: return t0:32
```

Lines 06.00 - 06.02 show the translation of the x86 `call` instruction into RREIL. First, the address of the native instruction immediately following the `call` is saved on top of the stack, then control is transferred to the target function. From line 11.00 onwards, the saved return address is replaced inside the trampoline function by the address given in `esi`. Here, the first two triplets of RREIL instructions corresponds to the x86-32 instruction `pop eax, push esi`. The `ret` instruction then pops it's destination address from the stack which in this case is not the address of the caller `main` but the value stored earlier in the register `esi`.

In this example, the trampoline function uses stack manipulation to obfuscate what would be a direct jump/call to the passed-in function pointer. The code from the native address 11 onwards is never decoded by our analysis and thus cannot cause disassembly warnings or precision losses due to invalid opcodes or branches to arbitrary memory. An analysis

that reconstructs one function at a time would assume that `call` returns and thus evaluate `mov eax, 2; ret` thereby inferring that the function can return with `eax = 2`.

Indeed, running this example in IDA Pro 6.1.110415 [26] yields this interpretation, namely, the disassembly given in the first listing. IDA Pro is not able to infer that instructions 0b and 10 are actually unreachable (or possibly data) nor does it emit a warning or stop its stack-pointer analysis. While the two unreachable instructions seem harmless, the bytes could have used to branch to a different part of the program, possibly leading to an explosion of the function CFG.

We conclude by a discussion of related work.

VI. RELATED WORK

While many static analyses for executables exist, most of them do not raise above the abstraction level of our $L(\text{finite})$ interface which precludes the use of classic numeric domains that assume that variables range over \mathbb{Z} . The latter domains, however, are particularly attractive as they have been shown to scale up to the analysis of large-scale C programs comprising several 100kLOC [11]. When analyzing C rather than binaries, the result of each assignment can be checked for overflow since the types of the high-level language indicate if signed or unsigned arithmetic is used. This distinction is lost for many assembler instructions. For instance, `add`, `sub` and `shl` carry no signedness information. This difficulty seems to be reason enough for products such as CodeSurfer/x86 [23] to consider numeric domains ranging over \mathbb{Z}_{2^w} , that is, domains that perform operations over a modulus ring, in order to calculate the input/output behavior of a basic block [16]. Interestingly, it is possible to synthesize transfer functions for a complete basic block that very accurately model bit-level operations composed of `xor` and `shl` [17]. However, this technique requires expensive SAT solving and does not scale to 64-bit architectures [16]. Moreover, it is neither modular since every new property has to be added to the Boolean formula passed to the SAT solver nor are the results easy to interpret. Note that SAT solving methods can still be integrated in our framework using the finite domain interface in Fig 2. Indeed, by only tracking variables defined by bit-level operations and using numeric domains to query and update other variables, much smaller Boolean formulae can be obtained.

Another challenging aspect of analyzing low-level code, be it C or assembler, is the reconstruction of the memory layout.

Our approach of using field inference [14] for registers seems novel and is conceptually simpler than approaches based on program transformation [27].

The problems of applying widening in the context of binary analysis has attracted little attention so far. It has been observed that narrowing after widening may lead to a precision loss in domains whose transfer functions depend on the values of variables [22]. The reconstruction of the CFG can be thought of as such a domain: If the pointer offset into a table of jump goals is temporarily unrestricted after widening, the set of possible jump targets is meaningless and the analysis cannot continue. Our approach of gathering constraints with which the widened state is narrowed is therefore crucial when analysing malware or more sophisticated programming styles.

Future work on our framework will address the design of further abstract domains enabling, for example, the analysis of string buffers [19] and more expressive numeric domains. Another challenging aspect is to distil an interface that allows for plugging-in modules that translate from $L(\text{rreil})$ to $L(\text{base-stmt})$ which would enable different strategies to analyse the functions of a program. Currently, we only perform semantic inlining, that is, we re-analyse a function for each new call-site.

VII. CONCLUSION

The paper proposed a modular framework for a reachability analysis of executable programs. It improves upon previous work in that it provides an interface for inferring structural, bit-level and numeric information, thereby allowing the design of simple abstract domains at each level that are nevertheless able to communicate across these interfaces. New abstract domains can be added to improve the precision of the analysis; this technique was demonstrated by presenting a new form of widening. The experimental results show that the achieved precision matches that of current heuristic-based approaches.

ACKNOWLEDGEMENT

We would like to thank Thomas Dullien and his team for useful discussions and the anonymous referees for improving the presentation of this paper. This work was supported by the DFG Emmy Noether programme SI 1579/1.

REFERENCES

- [1] A. Flexeder, B. Mihaila, M. Petter, and H. Seidl, “Interprocedural Control Flow Reconstruction,” in *Asian Symposium on Program Languages and Systems*, ser. LNCS, K. Ueda, Ed., vol. 6461. Shanghai, China: Springer, November 2010, pp. 188–203.
- [2] J. Kinder, F. Zuleger, and H. Veith, “An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries,” in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2009, pp. 214–228.
- [3] A. V. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. W. Reps, “Directed proof generation for machine code,” in *Computer Aided Verification*. Springer, 2010, pp. 288–305.
- [4] C. Cifuentes and S. Sendall, “Specifying the Semantics of Machine Instructions,” in *International Workshop on Program Comprehension*, 1998, pp. 126–133.
- [5] T. Dullien and S. Porst, “REIL: A platform-independent intermediate representation of disassembled code for static code analysis,” *CanSecWest Vancouver, Canada, 2009*. [Online]. Available: <http://www.zynamics.com/downloads/csw09.pdf>
- [6] J. Brauer and A. King, “Automatic Abstraction for Intervals using Boolean Formulae,” in *Static Analysis Symposium*, ser. LNCS, R. Cousot and M. Martel, Eds. Springer, September 2010.
- [7] N. Kettle and A. King, “Bit-Precise Reasoning with Affine Functions,” in *Satisfiability Modulo Theories*. Princeton, New Jersey, USA: ACM, 2008, pp. 46–52.
- [8] W. H. Harrison, “Compiler Analysis of the Value Ranges for Variables,” *Transactions on Software Engineering*, vol. 3, no. 3, pp. 243–250, May 1977.
- [9] M. Karr, “On affine relationships among variables of a program,” *Acta Informatica*, vol. 6, no. 2, pp. 133–151, 1976.
- [10] P. Cousot and N. Halbwachs, “Automatic Discovery of Linear Constraints among Variables of a Program,” in *Principles of Programming Languages*. Tucson, Arizona, USA: ACM, Jan. 1978, pp. 84–97.
- [11] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “A Static Analyzer for Large Safety-Critical Software,” in *Programming Language Design and Implementation*. San Diego, California, USA: ACM, Jun. 2003.
- [12] P. Cousot and R. Cousot, “Systematic Design of Program Analysis Frameworks,” in *Principles of Programming Languages*. San Antonio, Texas, USA: ACM, Jan. 1979, pp. 269–282.
- [13] G. Balakrishnan and T. Reps, “Analyzing Memory Accesses in x86 Executables,” in *In Comp. Construct*, 2004, pp. 5–23. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.3669>
- [14] A. Miné, “Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics,” in *Languages, Compilers, and Tools for Embedded Systems*. Ottawa, Canada: ACM, Jun. 2006, pp. 54–63.
- [15] A. Simon, *Value-Range Analysis of C Programs*. Springer, Aug. 2008, no. ISBN 978-1-84800-016-2.
- [16] M. Elder, J. Lim, T. Sharma, T. Andersen, and T. Reps, “Abstract Domains of Affine Relations,” in *Static Analysis Symposium*, Venice, Italy, Sep. 2011, to appear.
- [17] A. King and H. Søndergaard, “Automatic Abstraction for Congruences,” in *Verification, Model Checking, and Abstract Interpretation*, ser. LNCS, G. Barthe and M. Hermenegildo, Eds., no. 5944. Madrid, Spain: Springer, Jan. 2010, pp. 197–213.
- [18] P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. Rival, “Combination of Abstractions in the ASTRÉE Static Analyzer,” in *Asian Computing Science Conference*, ser. LNCS, M. Okada and I. Satoh, Eds., vol. 4435. Tokyo, Japan: Springer, Dec. 2006, pp. 272–300.
- [19] A. Simon and A. King, “Analyzing String Buffers in C,” in *Algebraic Methodology and Software Technology*, ser. LNCS, H. Kirchner and C. Ringissen, Eds., vol. 2422. Reunion Island, France: Springer, Sep. 2002, pp. 365–379.
- [20] —, “Taming the Wrapping of Integer Arithmetic,” in *Static Analysis Symposium*, ser. LNCS, G. File and H. R. Nielson, Eds., vol. 4634. Kongens Lyngby, Denmark: Springer, Aug. 2007, pp. 121–136.
- [21] P. Granger, “Static Analyses of Congruence Properties on Rational Numbers (Extended Abstract),” in *Static Analysis Symposium*, P. Van Hentenryck, Ed. Paris, France: Springer, Sep. 1997, pp. 278–292.
- [22] A. Simon and A. King, “Widening Polyhedra with Landmarks,” in *Asian Symposium on Programming Languages and Systems*, ser. LNCS, N. Kobayashi, Ed., vol. 4279. Sydney, Australia: Springer, Nov. 2006, pp. 166–182.
- [23] G. Balakrishnan, G. Gurian, T. Reps, and T. Teitelbaum, “CodeSurfer/x86 – A Platform for Analyzing x86 Executables,” in *Compiler Construction*, ser. LNCS, vol. 3443. Edinburgh, Scotland: Springer, Apr. 2005, pp. 250–254, tool-Demonstration Paper.
- [24] N. Krishnamoorthy, S. Debray, and K. Fligg, “Static Detection of Disassembly Errors,” in *Working Conference on Reverse Engineering*, A. Zaidman, G. Antoniol, and S. Ducasse, Eds. Lille, France: IEEE Computer Society, Oct. 2009, pp. 259–268.
- [25] H. Shacham, “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86),” in *Computer and Communications Security*, S. De Capitani di Vimercati and P. Syverson, Eds. ACM Press, Oct. 2007, pp. 552–61.
- [26] Hex-Rays, “IDA Pro Disassembler,” <http://www.hex-rays.com/idapro>, 2009, retrieved on 14.04.2011. [Online]. Available: <http://www.hex-rays.com/idapro>
- [27] K. Coogan and S. Debray, “Equational Reasoning on x86 Assembly Code,” in *Source Code Analysis and Manipulation*. Williamsburg, Virginia, USA: IEEE Computer Society, Sep. 2011.