Stony Brook University

Using Type Inference and Abstract

Interpretation for Static Binary Analysis

A Thesis Presented

by

Alireza Saberi

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

December 2012

Stony Brook University

The Graduate School

Alireza Saberi

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Prof. R. Sekar
Thesis Advisor
Computer Science Department
Stony Brook University

This thesis is accepted by the Graduate School

Charles Taber
Interim Dean of the Graduate School

Abstract of the Thesis

Using Type Inference and Abstract

Interpretation for Static Binary Analysis

by

Alireza Saberi

Master of Science

in

Computer Science

Stony Brook University

2012

In recent years, many research efforts have been dedicated to detect vulnerabilities in software. Most of these techniques are based on source code analysis. However, source code-based analysis methods are ineffective when the program source code is not available. In such a case, binary analysis is the only option. Yet, all binary analysis methods have to address serious challenges such as indirect memory access, missing functions and data abstraction. Historically, these problems have been addressed using rather ad hoc techniques. However, recent research has begun to reverse this trend. In this thesis, we cover Value-Set Analysis (VSA) and Abstract Stack Analysis (ASA) that use abstract interpretation to address aforementioned challenges in a principled way. We then move on to binary analysis methods that try to recover the missing type information in binaries. We describe TIE, Howard and REWARD as three binary type analysis methods and compare their effectiveness.

Contents

List of Figures

1 Introduction

In recent years, many research efforts have been dedicated to detect bugs and vulnerabilities in software. Most of these techniques are based on source code analysis. Source code based analysis techniques need source code or source code derived information such as intermediate representation and symbol table. Although, source code analysis methods are usually effective, there are situations that source code analysis methods cannot be applied or they cannot provide required results.

For instance, Commercial Off-The Shelf (COTS) software is only available in binary format. There is a great need to ensure COTS software does not perform malicious activity or cannot be subverted by exploiting a vulnerability.

Most malware is only available in binary format. There is a great need to understand their behavior and be able to detect and mitigate them based on their signatures and behaviors. In both cases, we need analysis methods to reason about binary without having extra information, such as symbol table, that is not required for binary execution .

There are also situations where binary analysis is used to complement source code analysis. For instance, Automatic Exploit Generation (AEG) [2] uses binary analysis to assist source code analysis result. AEG performs source code analysis to find vulnerabilities, but it needs low level information about activation record of vulnerable function to automatically produce exploit. AEG uses binary analysis to reveal required low level information such as activation record size and positions of variables in activation record.

Situations where binary analysis is used, can be categorized as:

- **Hidden Details.** Guided fuzz testing is an example where source code analysis is not helpful [16]. In general there are lots of additional details in binary which are not available in source code. Compiler makes many decisions during code optimizations and code generation which are hidden from source code analysis methods. Guided fuzz testing uses binary analysis to find variable adjacency in activation records and heap structure. This information helps fuzzer to mark potentially vulnerable variables.

- **Unified View.** Source code analysis techniques ran into trouble when they face a program that is coded in different programming languages. It is a challenging task to develop a source code analysis method that can operate on multiple languages simultaneously. Binary analysis can provide a unified view of the whole program. For instance, if there is an inline assembly snippet in a source code, it is usually omitted by source analysis methods. Binary analysis methods are useful in this case because they analyze inline assembly code consistently.

- **Library.** Dealing with libraries is another challenging issue for code analysis approaches, especially if the source code of library is not available. Most of the code analysis techniques simulate library calls by using function summaries or making stubs that simulate library calls. Since these

stubs are usually developed by hand, they may contain errors which lead to incorrect results. On the other hand, binary analysis approaches can handle library code similar to program code and there is no need to rely on potentially incomplete call stubs.

- **Global View.** Binary based analysis approaches can provide unified global view of programs that source centric analysis usually cannot provide. For instance, many source code analysis techniques are limited to analyzing only one module at a time, similar to the way how a compiler processes source code.

Research Interest. Our research is focused on securing COTS software using Binary Rewriting and Transformation. We can use binary rewriting and transformation to retrofit security defenses in binaries such as "stack canary" or function-pointer protection. It can also be used to implement security-enhancing transformations that maintain and check metadata associated with a program's data such as taint-tracking.

we are interested in binary rewriting and transformation with low overhead. Although dynamic binary rewriting techniques are less complicated than static techniques, they usually have high overhead. Thus, we have to use static binary rewriting approach that can provide us low overhead.

Challenges. To understand challenges in binary analysis, we consider a typical data flow analysis such as liveness analysis that can be used in binary rewriting and transformation. Liveness analysis involves distinguishing variables in the first step. Unfortunately, distinguishing between variables and identifying their boundaries is a challenging task in binaries. Furthermore, liveness analysis needs to keep track of read and write to variables. It is not an easy task if read or write operations use indirect memory access addressing.

The main source of challenges in binary analysis is missing abstractions which are available at source code and can be easily accessed by source code analyzers. For instance, variable types and sizes are readily available to source code analysis methods while there is no notion of variables or types in binaries. Binary analysis approaches usually have to perform expensive analysis to discover variables and infer about their types.

Function abstraction is another missing point at binary level. Even though there is a "call" instruction in binary, it has a very different notion than "call" in high level languages. In practice, there are calls to the middle of functions in binary and a high level function may be compiled to non-contiguous code snippets. Missing function abstraction can have negative effect on scalability of binary analysis approaches.

Indirect memory access is another serious challenge in binary analysis, especially static approaches. A simple access to an array or struct member is translated to a sequence of address arithmetic operations followed by indirect memory accesses. Having many address arithmetic operations makes reasoning about indirect memory access operations a difficult task. The lack of variable type and size infomation limits us to conservatively reason about the entire memory space for each operation which involves indirect memory access.

The lack of symbol table information is another challenge. Binary analysis methods cannot rely on debugging or symbol table information because COTS software or malware do not embed this information.

Approaches. In this report, we focus on Type Inference and Abstract Interpretation [10] as two fundamental techniques which are frequently used to address binary analysis challenges.

Type inference refers to the automatic deduction of the type of an expression in a programming language. Since high-level type information is discarded during compilation, binary analysis methods have to reverse engineer data abstraction. It is a challenging task. Section 2 describes how type inference basics are used to recover variables and their types in binaries.

Abstract interpretation is a non-standard program execution technique which attempts to obtain information about program run-time properties by using abstract values in place of actual values. Abstract-interpretation techniques try to reason about all program's behaviors for all possible inputs and all reachable program states. Section 3 is devoted to abstract interpretation on binaries.

2 Type Analysis for executable binaries

Compilers perform type checking on source code to prevent potential errors in programs. Type checking ensures operand with appropriate types are used by each operation in programs.

Since type information is not required for binary execution, many COTS software is shipped without any type information.

Type analysis for binaries is a process of associating each piece of data in binary with appropriate higher level type. Readers familiar with concept of type theory may have observed that this description is very similar to definition of type inference. Although some binary analysis techniques are based on type inference, all of them rely on heuristic methods to some extend; hence type inference results on binaries are not sound or complete all the time. Note that in some contexts like fuzzing and binary image forensics, these incomplete results are acceptable, because we try to infer as much as possible about binaries.

2.1 Motivation

Type information is required to address many challenges in binary analysis. For instance, binary analysis methods cannot disassemble binaries completely because of indirect control-flow transfer instructions. Type analysis can improve binary disassembly coverage by differentiating between integers and code pointer values. Disassembler considers code pointer values as valid starting addresses and disassembles instruction located after these addresses.

Vulnerability detection methods are another example that uses type information to reveal potential vulnerable variables. In this case, binary type analysis is used to detect variables that are adjacent to arrays. These variables are potential candidates to be overwritten by a write to array which fails to check array boundary. Guided fuzz testing is a vulnerability detection method that uses adjacency information [26].

Memory image forensics heavily rely on type analysis as well. For example, the first step to analyze a memory dump is to type all reachable memory. It can be very important to find out whether a four-byte value in memory dump represents an integer or an IP address [26].

Decompilation is the process of translating a binary to source code. Since each and every variable in source code should have a proper type, decompilation techniques [25] depend on binary type analysis. Binary rewriting techniques also benefit from type analysis. Binary rewriting techniques that use fine-grained type information get better performance when compared to binary rewriting methods that utilize only coarse-grained type information. *DisIRer* [13] is a binary rewriter that lifts up a binary to GCC IR (Intermediate Representation) by using coarse-grained type information. Since *DisIRer* does not recognize all the variables, it must simulate binary access to memory at the IR level. This prevents the compiler from performing many optimizations when the GCC IR is recompiled.

On the other hand, *secondwrite* [18] is a binary rewriter that uses fine-grained type information. Since *secondwrite* recognizes most of variables, it can generate IR which includes distinct variables. Compiler can produce optimized code using *secondwrite* IR.

2.2 Background

In this section we briefly go through background material in typing, subtyping and lattice theory [15]. Interested readers are referred to programming languages book such as Pierce [20] for more details.

Inference Rules. Typing rules are defined as Inference rules in the following form:

$$\frac{P_1 \; P_2 \; \ldots \; P_n}{C}$$

Premises $P_1, \ldots, P_n$ are on the top of the inference rule bar. We can conclude the statements below the bar C, if all the premises are satisfied. A rule is called axiom when there is no premise. Inference rules provide a formal compact notation for single-step inference. We can specify an inference algorithm by recursively applying inference rules on premises until an axiom is reached.

Typing. Let us use term t to refer to variable, value or expression. A term t is typable or well typed if there is some type T such that $t : T$ (term t has type T). The types of terms are specified as conclusion of inference rules while the sub-terms type are specified as the premise. We use typing context (also called type environment) Γ to make sure variables are typed consistently. Context Γ is a sequence of variables and their types. The type of term t is denoted by $\Gamma \vdash t : T$ which means term t has type T under context Γ.

For instance, if variable x has type T, we update Γ to include a new variable $x : T$. Later on, we can find type of x by looking it up in Γ. It is shown as:

$$\frac{x \; : \; \text{int} \; \in \Gamma}{\Gamma \vdash x \; : \; \text{int}}$$

We can type an expression by recursively typing each sub-expression. For example, the type of expression $x+y$ is integer when $x : \texttt{int} \in \Gamma$ and $y : \texttt{int} \in \Gamma$. The same expression can be inferred to be pointer if any of its operands has type pointer, since the plus operator accepts pointers and integers as arguments.

Subtyping. Type S is a subtype of T, $S <: T$, if any term of type S can be safely used where a term of type T is expected. The subtype relation is reflexive and transitive.

The bridge between the typing relation and subtype relation is provided by the subsumption rule:

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t \; : \; T}$$

This rule says if t has type S and $S <: T$, then every element t of S has also type T.

The depth subtyping rule expresses the subtype relation for records (e.g. C structures) where each field has a l_i label:

$$\frac{\text{for each } i \; S_i <: T_i}{\left\{l_i : S_i^{i \in 1 \ldots n}\right\} <: \left\{l_i : T_i^{i \in 1 \ldots n}\right\}}$$

This rule says if each record filed l_i has type S_i and $S_i <: T_i$, then we infer record S is a subtype of record T.

The subtype relation is also defined for function types as follows:

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

A function type is defined $S_1 \to S_2$ where S_1 is the argument and S_2 is the result type. Notice that in a function subtype relation, the subtype relation is reversed (contravariant) for the argument type, while it remains in the same direction (covariant) for the result type.

2.3 Type Information Categories

So far, several applications of binary type analysis are provided. Although all mentioned applications try to infer higher level type information from binaries, they have different definitions for higher level type. Considering different binary analysis techniques, we categorize them in two groups:

- The first group is interested in primitive types such as integer, pointer, char and array. DIVINE [5] and TIE [15] are two well-known members of the first group.
- Members of second group are more interested in semantic types. For example, REWARDS [26], tries to find out if a 4 byte variable represents an IP address or a file descriptor.

2.4 Source of Information

The main sources of type information in binary are library calls, system calls and machine opcodes. Generally, signatures of system calls and well-known library functions are known. In other words, each system call or library call reveals type information about input arguments and return value.

Each machine instruction has a semantic that imposes restrictions on its operands. These restrictions are very helpful in type inference. Almost all instructions have fixed size operands. The operand size is a good start but is not sufficient for finding exact type. Move, load and store instructions give us more hints such as the source type is a subtype of its destination type. A load instruction with 4-byte operand does not clarify whether the operand is an

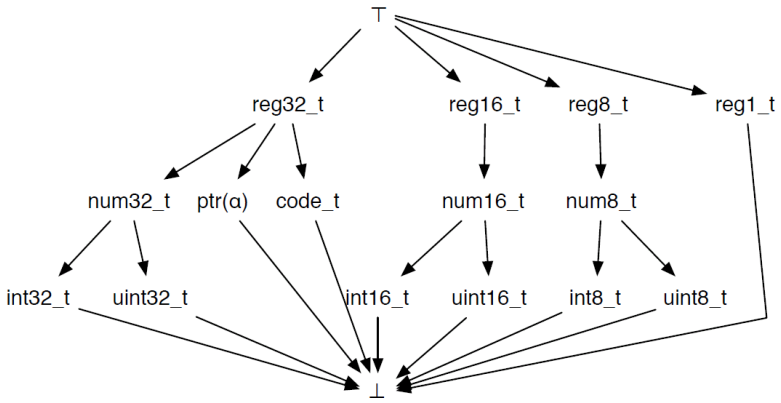


Figure 1: TIE type lattice showing the hierarchy of type [15]

integer, a float or a pointer. As a result, all binary type inference techniques have an expressive internal type representation with special types whose only information is the size, i.e. size32 for any 32-bit piece of data. Binary type analysis methods use lattice to represent their internal type hierarchy. Figure 1 shows hierarchy of types used in TIE.

Limited type information comes from the values of some constants. In many operating systems or architectures, we can rule out some values as pointers. For example, knowing Linux reserves `0xC0000000` to `0xFFFFFFFF` for kernel, we can assume that a constant value grater than `0xC0000000` is not a function pointer.

Debugging information, as a part of a binary, contains some type information. Since debugging information is optional and is not available in all binaries, we do not focus on it as a main source of type information. In general, we are interested in analyzing stripped binaries, i.e., binaries without debugging or symbol table information.

2.5 Type Analysis Steps

Unlike source code analysis methods which have the list of all variables, binary analysis methods have to perform some analysis to discover variables in binaries. Thus, most of binary analysis methods have a variable recovery step.

All binary type inference methods, static or dynamic, consist of two main steps:

1. Variable Recovery
2. Type Reconstruction

Variable recovery phase uses memory access pattern to uncover variables. In dynamic methods [24, 26] every access to memory reveals a new basic variable. Accessing 4-byte values inside a loop is an access pattern corresponds to an array.

Static methods [5, 15] analyze the code to find memory access patters like `ebp+0x0c` that represents local variable. Emmerik [25] categorizes several memory access patterns to find variables. For example, accessing memory using a constant address represents a global variable.

Type reconstruction is concerned with finding types for variables that are recovered in the previous phase. It consists of constraint generation and constraint solving. In the former phase, each instruction semantic is considered to produce constrains accordingly. For instance, moving a constant value zero to register r1 produces two constraints: a) r1 is an int, b) r1 is pointer and initialized to null. Constraint solving phase is concerned with finding answers (types) for all variables while satisfying all of the constraints.

2.6 Constraint Generation

Mycroft [17] was the first one who used type analysis for decompilation. Figure 2 shows a sample RTL (Register Transfer Language) code and the corresponding constraints. The sample code is presented in SSA (Static Single Assignment) form.

Although constraint generation may seem easy, it requires careful considerations in order to produce sound and complete constraints. For example, an arithmetic add instruction is used to add two integers, or an integer with a pointer. Thus three different constraint sets are generated for add instruction. Figure 2 shows the constraints genereted for the arithmetic add instruction.

All the constraints generated in Mycroft's paper [17], REWARD [26] and Howard [24] are in equality form. TIE [15] generates constraint with subtype relation. Figure 1 shows TIE type hierarchy and figure 3 gives some example rules for TIE inequality constraint generation. As illustrated in figure 1, TIE does not recover float and double variables.

2.7 Constraint Solving

In the constraint solving phase, we try to assign types to all variables in such a way that all the constraints are satisfied. Equality constraints are solved by unification. Unification is the process of substitution, where having $A = B$, we substitute all occurrences of A with B.

In equality constrains solving, occurs-check [20] should be performed to detect equalities in the form $a = ptr(a)$, where a variable is in both side of equation. TIE raises an error and drop such a constraint. Mycroft [17] tries to cope with occurs-check constraint failure by using structure. Solving constraint in figure 2 leads to an occur-check failure as $t0c$ appears on both sides of the equation:

$$t0c = t0b = ptr(mem(4 : t0c)) = ptr(mem(0 : t2a))$$

This is fixed by using $struct\, G\, \{t2a\, m0;\ t0c\, m4; ...\}$ i.e.,

$$t0c = ptr(mem(0 : t2a,\ 4 : t0c)) = ptr(struct\, G)$$

```
int f(struct A *x)
{  int r=0;
   for (; x !=0; x = x->t1)
      r += x->hd;
   return x;
}
```

(a) Original C program

Destination is last operand
r0a, r0b are SSA versions of register r0
t1a = type of r1a
mem(n:t) means a pointer offset n bytes from pointer t

(b) Legend

```
f:
      mov  r0,r0a
      mov  #0,r1a
      cmp  #0,r0a
      beq  L4F2
L3F2: mov  φ(r0a,r0c),r0b
      mov  φ(r1a,r1c),r1b
      ld.w 0[r0b],r2a
      add  r2a,r1b,r1c

      ld.w 4[r0b],r0c
      cmp  #0,r0c
      bne  L3F2
L4F2: mov  φ(r1a,r1c),r1d
      mov  r1d,r0d
      ret
```

(c) SSA machine code

$tf = t0 \rightarrow t99$
$t0 = t0a$
$t1a = int \vee t1a = ptr(\alpha_1)$
$t0a = int \vee t0a = ptr(\alpha_2)$

$t0b = t0a,\ t0b = t0c$
$t1b = t1a,\ t1b = t1c$
$t0b = ptr(mem(0 : t2a)$
$t2a = ptr(\alpha_3),\ t1b = int,\ t1c = ptr(\alpha_3) \vee$
$t2a = int,\ t1a = ptr(\alpha_4),\ t1c = ptr(\alpha_4) \vee$
$t2a = int,\ t1b = int,\ t1c = int$
$t0b = ptr(mem(4 : t0c))$
$t0c = int \vee t0c = ptr(\alpha_5)$

$t1d = t1a,\ t1d = t1c$
$t0d = t1d$
$t99 = t0d$

(d) Constraints

Figure 2: A sample RTL code, corresponding constraints [17]

Inequality constraints are solved by constraint propagation. Backtracking, local search and constraint propagations are the three most used techniques to solve Constraint Satisfaction Problem [1]. TIE is the only approach which has inequality constraints and propagates them via transitive subtype relations. For example, if we have a subtype relation $S = \{a <: b\}$ and we append the constraint $\{b <: c\}$, the constraint propagation algorithm, called closure by TIE, makes new subtype relation set three constraints $S = \{a <: b,\ b <: c,\ a <: c\}$.

The main challenge in constraint solving is conflicts. There are situations that conflicting constraints are generated in constraint generation phase. The most common way to deal with conflicts is to drop one of them and try to solve the remaining constraints. If constraint solver cannot find a solution, it drops the other constraint and tries again.

Statement	Generated constrains
$x := e$	$\tau_x = \tau_e$
$\texttt{goto}\ e$	$\tau_e = \texttt{ptr(code\_t)}$
$\texttt{if}\ e\ \texttt{then goto}\ e_t\ \texttt{else goto}\ e_f$	$\tau_e = \texttt{reg1\_t} \wedge \tau_t = \texttt{ptr(code\_t)} \wedge \tau_f = \texttt{ptr(code\_t)}$
$\texttt{call}\ f\ \texttt{with}\ m\ v^*\ \texttt{ret}\ r$	$\tau'_m = \tau_{mf} \wedge \bigwedge_v (\tau_v = \tau_{vf}.[v]) \wedge \tau_r = \tau_{rf}$ (where $F \vdash f : \tau_{mf} \to \tau_{vf} \to \tau_{rf}, \tau'_m = \#\texttt{update}(\tau_m)$)

Expression	Generated constraint for term with type variable τ
x (variable)	τ_x
v (integer)	τ_v
$-_n e$ (unary neg)	$\tau_e <: \texttt{int}n\texttt{\_t} \wedge \tau :> \texttt{int}n\texttt{\_t}$
$e_1 +_{32} e_2$	$(\tau_{e_1} <: T_\gamma \wedge \tau_{e_2} <: T_\gamma \wedge \tau :> T_\gamma \wedge T_\gamma <: \texttt{num32\_t})$ $\vee(\tau_{e_1} <: \texttt{ptr}(T_\alpha) \wedge \tau_{e_2} <: \texttt{num32\_t} \wedge \tau :> \texttt{ptr}(T_\beta))$ $\vee(\tau_{e_1} <: \texttt{num32\_t} \wedge \tau_{e_2} <: \texttt{ptr}(T_\alpha) \wedge \tau :> \texttt{ptr}(T_\beta))$
$e_1 +_{n \neq 32} e_2$	$\tau_{e_1} <: T_\gamma \wedge \tau_{e_2} <: T_\gamma \wedge \tau :> T_\gamma \wedge T_\gamma <: \texttt{num}n\texttt{\_t}$
$\sim_n e$	$\tau_e <: \texttt{uint}n\texttt{\_t} \wedge \tau :> \texttt{uint}n\texttt{\_t}$
$e_1 <_{S_n} e_2$	$\tau_{e_1} <: \texttt{int}n\texttt{\_t} \wedge \tau_{e_2} <: \texttt{int}n\texttt{\_t} \wedge \tau :> \texttt{reg1\_t}$
$\texttt{load}(m, i, d, \texttt{reg}n\texttt{\_t})$	$\tau_i = \texttt{ptr}(\tau_m.[i]) \wedge \tau = \tau_m.[i] \wedge \tau <: \texttt{reg}n\texttt{\_t}$
$\texttt{store}(m, i, v, d, \texttt{reg}n\texttt{\_t})$	$\tau_i = \texttt{ptr}(\tau_v) \wedge \tau = \tau_m\{i : \tau_v\} \wedge t.[i] <: \texttt{reg}n\texttt{\_t}$
$\Phi(e_1, \cdots, e_n)$	$\tau' <: \tau_{e_1} \cap \cdots \cap \tau_{e_n}$

Figure 3: Example rules for type constrain generation in TIE [15]

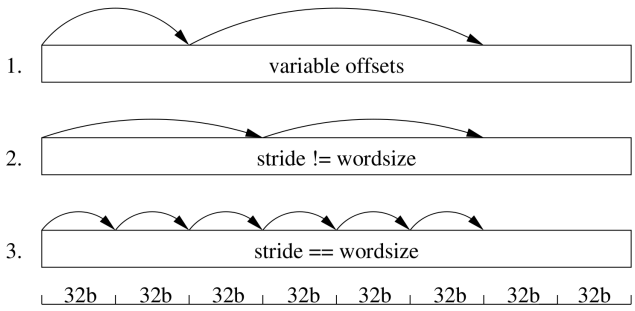


Figure 4: Three different access patterns[24]

2.8 Type Analysis Challenges

There are several challenges in variable recovery and type reconstruction phase. Indirect memory access is one of the main challenge in variable recovery that has great effect on final result. Indirect memory accesses also make it difficult for static analysis methods to reason about array boundary and structure size.

Dynamic analysis methods rely on different input samples to find array boundary and structure size; hence if they are not fed by appropriate inputs, they are not capable to produce the correct result. Static methods usually use heuristic to find the result.

There are situations that we face challenges in constraint generation. Figure 4 shows different memory access patterns to a memory region. Since we use memory access patterns to produce appropriate constraints, we should sort out memory access conflicts. For example pattern 1 and 2 help to generate appropriate constraints based on access to structure's element, but pattern 3 is generated by using a general function like `memcpy`. These general functions are used to copy or manipulate structures without having any idea about internal structure. In 32 bit machines, general functions like `memcpy` and `memset` access memory in 32 bit strides that is misleading and generate conflicting constraints.

2.9 Static Type Analysis vs Dynamic Type Analysis

Dynamic type analysis approaches such as REWARD [26] and Howard [24] are light-weighted techniques when compared to static approaches such as TIE [15] and VSA [4]. Dynamic approaches have runtime information about memory access pattern and this information helps them to improve the final results.

Main challenge for dynamic approaches is code coverage. If dynamic approaches cannot provide appropriate test to cover all the code then some variables are left out. Experimental results in REWARD and Howard shows 60% variable coverage. It means if there is no error in type inference analysis, REWARD and Howard can only discover and type 60% of variables in their test cases.

Static type analysis approaches such as TIE and VSA are considered to be heavy-weighted analysis. TIE and VSA does not have runtime information such as location accessed by indirect memory access; hence they have to make over approximation in order to produce sound results. In some cases, this over approximation affects the accuracy of the results.

2.10 Arrays and Structures

Identifying arrays and their boundaries in binaries is a challenging task. Although there are distinct differences between arrays and structures in source code, they are very similar in binary. Some type analysis techniques such as TIE [15] does not distinguish between array and strucures and type them in the same way.

REWARD [26] and Howard [24] rely on memory access pattern and heuristic to find arrays and array boundaries. For example, REWARD marks a region as an array when adjacent elements in memory are accessed within a loop. REWARD makes use of well-known library function signatures to find well-known structures. For example, when there is a call to the function `send(sock *s,...)`, REWARD considers the first element as a pointer to socket structure and type memory according to the socket structure definition in the library source code. If the source code is not available, REWARD cannot find structures.

Howard has a more specialized array detection component. It can detect different pattern of accessing elements of an array. For instance, an element may be accessed using base address by `base+index*stride` pattern or it may be accessed by incrementing to a pointer pointing to previous element in `*(prev_pointer++)` pattern. Howard array detection component has some heuristics to detect boundary elements. However, loop unrolling changes the access pattern of array, Howard has introduced some heuristics to deal with it.

DIVINE [5] (an improved VSA algorithm) is the only static approach that tries to reveal internal structure of aggregates in binaries. DIVINE uses VSA and Aggregate Structure Identification (ASI) [21] algorithm. ASI is a unification-based flow-insensitive algorithm to identify the structures of aggregates in a program (such as array and C structs). ASI basically generates some equality constraints based on access patterns and uses a unification flavor algorithm to solve the constraints.

2.11 Summary

In this chapter, we focused on type analysis as an essential analysis that helps us to find out more about binary internal and behavior. However, type inference on source code is flow-insensitive; type analysis in binaries is a bidirectional flow-sensitive analysis.

We introduced two different categories of type in binary and showed sources of type information in binary. Variable recovery and type reconstruction are

discussed as the two main steps of binary type analysis. We showed how constraint generation and constraint solving can be used in type analysis. Finally, we discussed different approaches for identifying the structure of aggregates.

3 Abstract Interpretation

Abstract Interpretation [10] is a theory of sound approximation that is used in many areas such as proofs, static analysis, model-checking, counter-example-based refinement, program transformation, watermarking, information hiding, code obfuscation, malware detection and verification [9]. Abstract Interpretation has been applied successfully in static analysis domain to automatically infer about run-time properties of programs. In this section we focus on using abstract interpretation in two static binary analysis methods called Abstract Stack Analysis (ASA) [23] and Value Set Analysis (VSA) [4].

ASA is applied on function granularity and use abstract interpretation to keep track on local variables and register values. VSA [4] is a combined pointer-analysis and numeric-analysis algorithm based on abstract interpretation. VSA computes an over-approximation for registers and variable values.

3.1 Motivation

Abstract Interpretation can assists static analysis. For instance, consider the case that we use static binary rewriting to perform taint tracking. If a static binary analysis method analyzes a function `f` and shows that a local variable `x` is only accessed within `f` then the binary rewriting can use a local variable as taint metadata which is more efficient than using global variable as taint metadata. ASA [23] technique shows how an abstract interpretation analysis result assists escape analysis.

Although indirect memory access is very common in X86 binary, static analysis methods have hard time to reason about them. In general, a register used as indirect memory access operand may be initialized by a read from memory. In this case, it is necessary to know the value of that memory location to determine the value of the register. This is a very challenging task for static analysis methods, thus many of them ignore memory operations [11] or treat memory operations in a unsound manner [8]. VSA uses abstract interpretation to computes an over-approximation for registers and variable values. Static analysis methods can use result of VSA to reason about indirect memory access operations.

3.2 Definition

Abstract Interpretation [1] is described as a non-standard execution which tries to reason about all possible program executions by using "abstract values" in place of actual computed values. Abstract Interpretation associates with each program point the set of all memory stores (C) that can occur when program control reach that point. In static analysis, these stores represent variables located in memory. The set $\wp(Store)$ of all sets of stores form a complete lattice that is called concrete domain. Concrete domain is represented by $\wp(Store)$ or $\wp(C)$. Abstract interpretation uses simpler lattice (Abs) that is connected to $\wp(Store)$ by an abstraction function $\alpha : \wp(Store) \rightarrow Abs$ where Abs is a

[1] We description abstract interpretation in the context of static analysis.

lattice of abstract stores that is called abstract domain. Section 3.6 describes two abstract domains based on interval and strided intervals that is used by static analysis methods.

We assume standard abstract interpretation where *concrete* and *abstract domains*, L given by $\wp(C)$ and A, are complete lattice $\langle L, \subseteq, \cap, \cup, C, \emptyset \rangle$ and $\langle A, \sqsubseteq, \sqcap, \sqcup, \top_A, \bot_A \rangle$, respectively. The two lattices are related by abstraction and concretization maps α and γ forming a *Galois connection* $\forall c \in L : \forall a \in A : \alpha(c) \sqsubseteq a \Leftrightarrow c \subseteq \gamma(a)$ [19]. We write this fact as: $\langle L, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq \rangle$.

Galois connection represents the notion of abstract soundness. We say that an abstraction is sound (or correct) if the abstract semantics covers all possible cases of the concrete semantics. In context of ASA and VSA, it means for every program point p in the program, an abstract value a, associated with variable v, covers all the possible values for variable v at point p in the normal execution. In other words, if value v is not in abstract representation of a variable, it must be impossible for the variable to have value v in any run of the program.

3.3 The Design of Abstract Interpretation

An Abstract Interpretation for a program can be designed in the following steps [7]:

1. Identify the interesting property
2. Choose an appropriate abstract domain that captures above property precisely while balancing computational and representational constraints
3. Establish a precise connection between the concrete and abstract domain
4. Define the required operations in abstract domain

In this section we focus on two abstract interpretation-based static analysis methods called VSA [6] and ASA [23]. We describe these two methods by going through steps of Abstract Interpretation design.

3.4 Example

In this section, we illustrate the idea of abstract interpretation using Collatz problem as example. We try to find out whether variable n is even or odd at each point of program. Figure 5 shows Collatz problem and corresponding flowchart. Collatz problem is about determining whether this program terminates for all possible initial n. To our knowledge it is still unsolved.

The property of interest is whether n is odd or even in different points of program.

Accumulating semantics associate for each point in program (A, B, ..,G) all the possible values of variable n at that point. Then we choose abstract values as $\{\bot, even, odd, \top\}$. The abstract values form a complete lattice as abstract domain. Figure 6 shows the abstract domain.

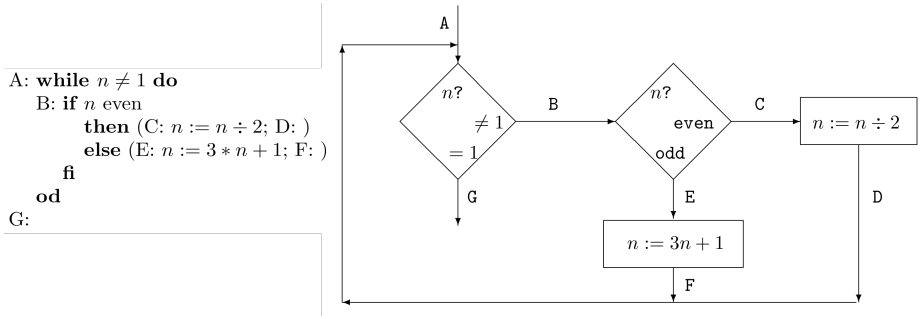


Figure 5: Collatz Problem [14]

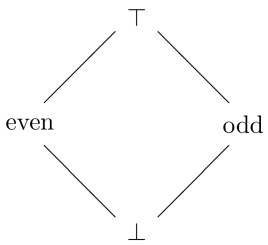


Figure 6: Abstract Domain for finding even or odd numbers

For this sample the connection between concrete values and abstract values is obvious. The abstraction function map each number to abstract value (*even* or *odd*). We define two operations for $n \div 2$ and $3n + 1$ over abstract domain.

$$f_{n \div 2}(abs) = \begin{cases} \bot & if\ abs = \bot \\ \top & else \end{cases}$$

$$f_{3m+1}(abs) = \begin{cases} \bot & if\ abs = \bot,\ else \\ even & if\ abs = even,\ else \\ odd & if\ abs = odd,\ else \\ \top & if\ abs = \top \end{cases}$$

Now we define a set of approximate data-flow equations that describe program's behavior.

$$
\begin{aligned}
abs_A &= \alpha(S_0) \\
abs_B &= (abs_A \sqcup abs_D \sqcup abs_F) \sqcap \top \\
abs_C &= abs_B \sqcap even \\
abs_D &= f_{n \div 2}(abs_C) \\
abs_E &= abs_B \sqcap odd \\
abs_F &= f_{3n+1}(abs_E) \\
abs_G &= (abs_A \sqcup abs_D \sqcup abs_F) \sqcap odd
\end{aligned}
$$

Final step is to compute the fixpoint for a simple input. Following shows fixpoint calculation for $S_0 = \{5\}$.

abs_A	abs_B	abs_C	abs_D	abs_E	abs_F	abs_G	iteration
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	0
odd	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	1
odd	*odd*	$\bot$	$\bot$	$\bot$	$\bot$	*odd*	2
odd	*odd*	$\bot$	$\bot$	*odd*	$\bot$	*odd*	3
odd	*odd*	$\bot$	$\bot$	*odd*	*even*	*odd*	4
odd	$\top$	$\bot$	$\bot$	*odd*	*even*	*odd*	5
odd	$\top$	*even*	$\bot$	*odd*	*even*	*odd*	6
odd	$\top$	*even*	$\top$	*odd*	*even*	*odd*	7,8,...

The conclusion is that n is always *even* at points C and F, and always odd at E and G.

3.5 Property of Interest

Although both ASA [23] and VSA [4] are designed to assist binary analysis, they address this goal with different point of views and ideas, thus property of interest is not the same for both of them.

ASA is more interested in analysis methods that can assist binary rewriting. For instance, in the context of taint-tracking, we want to maintain and update metadata (taint information) for each variable and register using static binary rewriting. There are several optimizations that can reduce the taint-tracking overhead. Tag sharing is an optimization that uses ASA results. Using ASA result, tag sharing optimization can statically find all the variables that can share a single copy of metadata tag. These variables usually keep the same value or slightly different value (in the form of $i = j + 5$) that produce the same metadata value.

As property of interest, ASA keeps track of register values, variable values and the simple linear relation between them. This information is useful for optimization methods like tag sharing. Although the register or memory values are not always available for static analysis, we can still apply the optimization if we keep the linear relation between variable and registers. For example, if ASA

result shows register *esp* and a local variable inside function `f` keep the same value, we can apply the tag sharing, without knowing the exact value of *esp* or local memory.

Recovering information about indirect memory-access operations is a challenge in X86 binary analysis. Most of data flow analysis methods cannot work on binaries without having information about indirect memory access. VSA addresses this issue by combining a pointer-analysis and a numeric-analysis algorithm. As property of interest, VSA computes an over-approximation of all possible concrete values at each program point. The result of VSA can be used in "dependence analysis" and "resolving indirect jumps and calls".

3.6 Abstract Domain

Both VSA [4] and ASA [23] use accumulating semantics similar to [10]. Accumulating semantics associates with each program point, the set of all memory stores that can occur when program control reaches that point. In other words, VSA and ASA try to compute all the possible variables and register values for each line of assembly code in binary. This kind of analysis is very expensive, thus we define an abstract domain and continue our analysis on abstract domain.

ASA. Points in ASA abstract domain are represented as $Base + [l, h]$, where $Base$ and h are optional. $Base$ denotes the value of a register or local memory at the entry point of a function. A missing $Base$ is treated as equivalent to zero, and a missing h is treated as equal to l. Both l and h can be negative integers. $Base$ is usually represented by a symbolic value X. $Base + [l, h]$ represents any concrete value in range of $X + l$ to $X + h$ (concretization).

ASA is interested in analysis on function granularity and the initial value of *esp*, denoted by $BaseSP$, has an important role in defining the notion of local variable. Figure 7 shows the ASA abstract domain that is in the form of complete lattice . The abstract value *const* denotes an unknown concrete value that does not point to local memory addresses. Abstract value *const* capture the assumption that address of local variable i is created inside its function f and cannot exist before activation of f.

VSA. VSA shows every memory address is a pair of memory region and offset $(memory\,region, offset)$. VSA uses a set of tuples of the form $(region_i \mapsto \{offest_1^i, offset_2^i, ..., offset_n^i\})$, called `absEnv,` to represent its abstract domain. VSA uses a k-bit strided-interval (SI) [22] to represent the set of offsets in each memory region. A k-bit strided interval $s[l, u]$ represents a set of integers $\{i \in [-2^{k-1}, 2^{k-1} - 1] \mid l \leq i \leq u,\ i \equiv [l]_s\}$ where

- congruence class of l mod m, defined as $[l]_s = \{l + i \times s \mid i \in \mathbb{Z}\}$
- s is called the stride.
- $[l, u]$ is the interval.
- $0[l; l]$ represents the singleton $\{l\}$.
- we consider$\perp$ as strided interval with empty set of offsets.

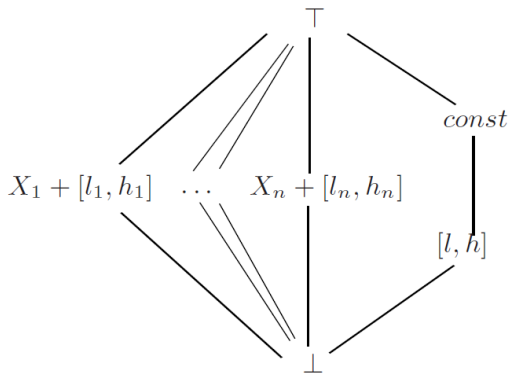


Figure 7: The abstract domain for ASA [23]

Consider the set of addresses

$$S = \{(Global \mapsto \{2, 6, 14, 18\}),\ (ARmain \mapsto \{-20, -16\})\}$$

The VSA abstract store, called value-set, for S is the set

$$\{(Global \mapsto 4[2, 18],\ ARmain \mapsto 4[-20, -16])\}$$

Note that the value-set for S is an over-approximation; the value-set includes the global address 10, which is not an element of S.

Value-set forms a complete lattice. Relation $(vs_1 \sqsubseteq^{vs} vs_2)$ means value-set vs_1 is subset of vs_2. Meet (intersection) and join (union) are shown by $(vs_1 \sqcap^{vs} vs_2)$ and $(vs_1 \sqcup^{vs} vs_2)$ respectively.

Comparison. Figure 8 shows a sample assembly code that VSA cannot infer about the values but ASA produces useful result. In this sample, the value of *ebp* is moved to *eax*, then increased by `5`. If we use VSA to infer about *eax* just before instruction `4`,we get $\top$ as a result that is not useful. Register *ebp* can have any value ($\top$) at the beginning and adding `5` to it does not change the result. ASA cannot infer the exact value of *eax* before instruction `4`,but it can infer *eax* has value $ebp + [5, 5]$ which is useful.

Another advantage of ASA is being able to capture the equality between registers and variables. For instance, consider the case that *eax* has two potential values `{1000, 1004}` before the instruction $mov\, eax, ebx$. Then ASA can easily capture the equality between *eax* and *ebx*, but VSA cannot. VSA can infer that *eax* and *ebx* have SI $4[1000, 1004]$, but it does not mean *eax* and *ebx* have the same value.

VSA gains advantage by using SIs instead of simple interval, because SIs are more accurate and can keep alignment information. SI helps VSA to achieve more accurate results on indirect addressing operations such as field-access operations in an array of structs or dereferencing a pointer. For instance, let $*t$ denotes a 4-bytes fetch from address t . Suppose t contains the set `{2000,`

```
1 push ebp
2 mov esp, ebp
3 mov [esp], eax
4 add 5, eax
5 mov eax, [esp-12]
```

Figure 8: Sample assembly code using ebp value

2004} that is represented as SI 4[2000, 2004]. Since VSA uses SI representation, it fetches exactly from two memory addresses 2000 and 2004. On the other hand, the interval [2000, 2004] contains addresses 2001, 2002, 2003; hence $*[2000, 2004]$ could result in forged addresses.

Although each abstract store in VSA is associated with only one abstract value, ASA associates a set of abstract values to each abstract store. This set can have up to k members where k is a small constant value. If the number of abstract values in the set exceeds k, an appropriate generalization is used to reduce the number of values to less than k.

3.7 Operations in Abstract Domain

Both ASA and VSA address basic operations like assignment, dereferencing, and addition, but they have different abstract domain; hence different ways for applying these operations on abstract domain.

ASA. Figure 9 describes ASA operations in a simplified RISC format. In this figure, Instruction I is applied on abstract store A and abstract store A' is produced. R denotes the registers and $A[l]$ denotes abstract store at location l. If the location being updated is precisely known then ASA uses "strong update" to replace the abstract value. If it is not known precisely, then ASA cannot use "strong update"; instead, we use the notion of "weak update" by adding the value of the right-hand side of the assignment to each of these potential locations.

As we have mentioned before, ASA is only interested on function granularity and only distinguishes between local memory locations. For instance, it differentiate between $BaseSP + a$ and $BaseSP + b$ when $a \neq b$.

Static operation "$\oplus$" denotes arithmetic addition "$+$" in abstract domain. ASA simply interpret $X + [l_1, h_1] \oplus [l_2, h_2]$ as $X + [l_1 + l_2, h_1 + h_2]$. The result of $X + [l_1, h_1] \oplus Y + [l_2, h_2]$ is *const* if it cannot represent address of local variables; in other words neither X nor Y should not be $BaseSP$. If any of X or Y is equal to $BaseSP$ the result would be $\top$.

ASA captures a single summery for each function `f`. This summary includes change in `esp`, maximum size of `f` activation record, input parameters to `f` and change to registers and parameters of `f`. Having function summary, abstract interpretation uses a function `applysum` to update the abstract store to reflect local memory and register changes specified in the summary.

VSA. As we have mentioned before, tuple `AbsEnv` is used to store abstract values for registers, global variables, local variables and X86 flags at each point of program. Figure 10 shows main VSA operations in abstract domain. In this

Instruction (I)	Abstract store ($\mathcal{A}'$)
$R := c$	$Upd(\mathcal{A}, [R \mapsto [c, c]])$
$R := R'$	$Upd(\mathcal{A}, [R \mapsto \mathcal{A}[R']])$
$R := *(R')$	$Upd(\mathcal{A}, [R \mapsto \bigcup_{x \in \mathcal{A}[R']} \mathcal{A}[x]])$
$*(R) := R'$	$Upd(\mathcal{A}, [x \mapsto \mathcal{A}[R']])$, if $\mathcal{A}[R] = \{x\}$ $Upd(\cdots(Upd(\mathcal{A}, [x_1 \mapsto \mathcal{A}[x_1] \cup \mathcal{A}[R']]) \cdots$ $\cdots), [x_n \mapsto \mathcal{A}[x_n] \cup \mathcal{A}[R']])$ if $\mathcal{A}[R] = \{x_1, ..., x_n\}, n > 1$
$R := R_1 + R_2$	$Upd(\mathcal{A}, [R \mapsto \bigcup_{r_1 \in \mathcal{A}[R_1], r_2 \in \mathcal{A}[R_2]} r_1 \oplus r_2])$
$call(f)$	$Upd(\cdots(Upd(\mathcal{A}, [x_1 \mapsto applysum(\mathcal{A}[x_1], f, x_1)]) \cdots$ $), [x_n \mapsto applysum(\mathcal{A}[x_n], f, x_n)])$ where $ModifiedNonLocals(f) = \{x_1, ..., x_n\}$

Figure 9: ASA operations in abstract domain [23]

figure instruction I is applied on input abstract store, `in:AbsEnv`, and produces `AbsEnv` tuple, called `out`, as output abstract store. In this figure $in[R]$ denotes value-set of register R in abstract store in. The abstract domain for X86 flags, called **Bool3,** is defined as: **Bool3** $= \{False, Maybe, True\}$. The value $Maybe$ means "may be $False$, maybe $True$.

Three operators and couple of interesting point about figure 10 are described as:

- $(vs +^{vs} c)$: Returns the value-set obtained by adjusting all values in vs by the constant `c`, e.g., if $vs = (4, 4[4, 12])$ and $c = 12$, then$(vs +^{vs} c) = (16, 4[16, 24])$.

- $*(vs, s)$:Returns a pair of sets (F, P). `F` represents the set of "fully accessed" memory location: it consists of the memory locations that are of size s and whose starting addresses are in vs. `P` represents the set of "partially accessed" memory location: it consists of memory location whose starting addresses are in vs but are not of size s.

- $RemoveLowerBounds(vs)$: Returns the value-set obtained by setting the lower bound of each component SI to -2^{31}. For example, if $vs = ([0, 100], [100, 200])$, then $RemoveLowerBounds(vs) = ([-2^{31}, 100], [-2^{31}, 200])$.

- $RemoveUpperBounds(vs)$: Similar to $RemoveLowerBounds$, but sets the upper bound of each component to $2^{31} - 1$.

- Each AR region of a recursive procedure , potentially represents more than one concrete data object, assignments to their local variables must be modeled by weak updates, i.e., the new value-set must be joined with the existing one, rather than replacing it (case two of figure 10).

Instruction	*AbstractTransformer*(*in*: AbsEnv): AbsEnv
$\texttt{R1} = \texttt{R2} + c$	Let *out* := *in* and $vs_{\texttt{R2}}$:= *in*[R2] *out*[R1] := $vs_{\texttt{R2}} +^{\text{vs}} c$ **return** *out*

$*(\texttt{R1} + c_1) = \texttt{R2} + c_2$	Let $vs_{\texttt{R1}}$:= *in*[R1], $vs_{\texttt{R2}}$:= *in*[R2], $(F, P) = *(vs_{\texttt{R1}} +^{\text{vs}} c_1, s)$, and *out* := *in*
	Let *Proc* be the procedure containing the instruction
	if ($\|F\| = 1 \wedge \|P\| = 0 \wedge$ (F has no heap a-locs or a-locs of recursive procedures)) **then**
	  out[v] := $vs_{\texttt{R2}} +^{\text{vs}} c_2$, where $v \in F$ // Strong update
	else
	  for each $v \in F$ **do**
	  out[v] := *out*[v] $\sqcup^{\text{vs}}$ ($vs_{\texttt{R2}} +^{\text{vs}} c_2$) // Weak update
	  end for
	end if
	for each $v \in P$ **do** // Set partially accessed a-locs to $\top^{vs}$
	  out[v] := $\top^{vs}$
	end for
	return *out*

$\texttt{R1} = *(\texttt{R2} + c_1) + c_2$	Let $vs_{\texttt{R2}}$:= *in*[R2], $(F, P) = *(vs_{\texttt{R2}} +^{\text{vs}} c_1, s)$ and *out* := *in*
	if ($\|P\| = 0$) **then**
	  Let $vs_{rhs} := \bigsqcup^{\text{vs}}\{in[v] \mid v \in F\}$
	  out[R1] := $vs_{rhs} +^{\text{vs}} c_2$
	else
	  out[R1] := $\top^{vs}$
	end if
	return *out*

$\texttt{R1} \leq c$	Let $vs_c := ([-2^{31}, c], \top^{si}, \ldots, \top^{si})$ and *out* := *in* *out*[R1] := *in*[R1] $\sqcap^{\text{vs}}$ vs_c **return** *out*
$\texttt{R1} \geq \texttt{R2}$	Let $vs_{\texttt{R1}}$:= *in*[R1] and $vs_{\texttt{R2}}$:= *in*[R2] Let vs_{lb} := RemoveUpperBounds($vs_{\texttt{R2}}$) and vs_{ub} := RemoveLowerBounds($vs_{\texttt{R1}}$) *out* := *in* *out*[$R1$] := $vs_{R1} \sqcap^{\text{vs}} vs_{lb}$ *out*[$R2$] := $vs_{R2} \sqcap^{\text{vs}} vs_{ub}$ **return** *out*

Figure 10: VSA Operations in Abstract Domain[3]

$$
\begin{array}{lrl}
(1) & a+c<-2^{31}, b+d<-2^{31} & \Rightarrow \mathbf{a+c \leq x+y \leq b+d} \\
(2) & a+c<-2^{31}, b+d\geq -2^{31} & \Rightarrow \mathbf{-2^{31} \leq x+y \leq 2^{31}-1} \\
(3) & -2^{31}\leq a+c<2^{31}, b+d<2^{31} & \Rightarrow \mathbf{a+c \leq x+y \leq b+d} \\
(4) & -2^{31}\leq a+c<2^{31}, b+d\geq 2^{31} & \Rightarrow \mathbf{-2^{31} \leq x+y \leq 2^{31}-1} \\
(5) & a+c\geq 2^{31}, b+d\geq 2^{31} & \Rightarrow \mathbf{a+c \leq x+y \leq b+d}
\end{array}
$$

Figure 11: Different cases to consider for adding two two's-complement numbers [12]

- Unaligned writes can modify parts of various variables that could create forged addresses. In case 2 of figure 10, such writes are treated safely by setting the values of all partially modified variables to $\top^{vs}$. Similarly, case 3 treats a load of a potentially forged address as a load of $\top^{vs}$.

Although arithmetic addition is the only arithmetic operation presented in figure 10, VSA support subtraction, bitwise And($\&^{vs}$), Or($|^{vs}$) and Xor($\hat{}^{vs}$). In this section, we present arithmetic addition on SIs and how VSA use that to define value-set addition. Interested readers are referred to [3] for more details on remaining supported arithmetic operations.

VSA uses H. Warren's [12] algorithms for performing arithmetic and bit-level operations on intervals (i.e., strided interval with stride 1). Figure 11 shows stride addition for two two's-complement values x and y where $a \leq x \leq b$ and $c \leq y \leq d$. The result of $x+y$ is not always in the interval of $[a+c, b+d]$ because of overflow in either positive or negative direction.

Using the definition for arithmetic operation on SIs, VSA defines the value-set arithmetic. VSA classifies value-sets in four categories and the result of value-set arithmetic is defined for each value-set type. Figure 12 shows different value-set groups and figure 13 shows the value-set type produced by value-set addition.

The valus-set operation $+^{vs}$ is defined as follow:

- $VS_{glob} +^{vs} VS_{glob}$ **:** Let $vs_1 = (si_0^1, \bot, ...)$ and $vs_2 = (si_0^2, \bot, ...)$.
  Then $vs_1 +^{vs} vs_2 = (si_0^1 +^{si} si_0^2, \bot, ...)$.

- $VS_{glob} +^{vs} VS_{single}$ **:** Let $vs_1 = (si_0^1, \bot, ...)$ and $vs_2 = (\bot, ..., si_l^2, \bot, ...)$.
  Then $vs_1 +^{vs} vs_2 = (\bot, ..., si_0^1 +^{si} si_l^2, \bot, ...)$.

- $VS_{single} +^{vs} VS_{glob}$ **:** Let $vs_1 = (\bot, ..., si_l^1, \bot, ...)$ and $vs_2 = (si_0^2, \bot, ...)$.
  Then $vs_1 +^{vs} vs_2 = (\bot, ..., si_l^1 +^{si} si_0^2, \bot, ...)$.

- $VS_{arb} +^{vs} VS_{glob}$ **:** Let $vs_1 = (si_0^1, ..., si_k^1, ...)$ and $vs_2 = (si_0^2, \bot, ...)$.
  Then $vs_1 +^{vs} vs_2 = (si_0^1 +^{si} si_0^2, ..., si_k^1 +^{si} si_0^2, ...)$.

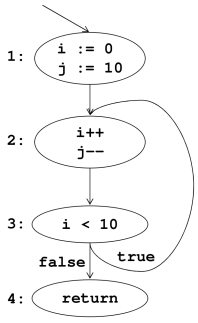
- $VS_{glob} + VS_{arb}$ **:** Let $vs_1 = (si_0^1, \bot, ...)$ and $vs_2 = (si_0^2, ..., si_k^2, ...)$.
  Then $vs_1 +^{vs} vs_2 = (si_0^1 +^{si} si_0^2, ..., si_0^1 +^{si} si_k^2, ...)$.

Kind	Form of value-set	
VS_{glob}	$(si_0, \bot, \ldots)$	si_0 is a set of offsets in the Global memory-region
VS_{single}	$(\bot, \ldots, si_l, \bot, \ldots)$	si_l is a set of offsets in the l-th memory-region ($l \neq$ Global)
VS_{arb}	$(si_0, \ldots, si_k, \ldots)$	si_k is a set of offsets in the k-th memory-region
$\top^{vs}$	$(\top^{si}, \top^{si}, \ldots)$	all addresses and numeric values

Figure 12: Four value-set types [3]

$+^{\text{VS}}$	VS_{glob}	VS_{single}	VS_{arb}	$\top^{vs}$
VS_{glob}	VS_{glob}	VS_{single}	VS_{arb}	$\top^{vs}$
VS_{single}	VS_{single}	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$
VS_{arb}	VS_{arb}	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$
$\top^{vs}$	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$	$\top^{vs}$

Figure 13: Value-set type produced by addition[3]



(a)

i	j
$[1,1]$	$[9,9]$
$[1,2]$	$[8,9]$
$[1,3]$	$[7,9]$
$\vdots$	$\vdots$
$[1,9]$	$[1,9]$
$[1,9]$	$[0,9]$
$[1,9]$	$[-1,9]$
$\vdots$	$\vdots$

(b)

i			
$old_{\mathtt{i}}$	$new_{\mathtt{i}}$	$old_{\mathtt{i}} \nabla new_{\mathtt{i}}$	$\mathtt{i} < 10$
$\perp$	$[1,1]$	$[1,1]$	$[1,1]$
$[1,1]$	$[1,2]$	$[1,\infty]$	$[1,9]$
j			
$old_{\mathtt{j}}$	$new_{\mathtt{j}}$	$old_{\mathtt{j}} \nabla new_{\mathtt{j}}$	$\mathtt{i} < 10$
$\perp$	$[9,9]$	$[9,9]$	$[9,9]$
$[9,9]$	$[8,9]$	$[-\infty,9]$	$[-\infty,9]$

(c)

Figure 14: (a) Example program; (b) ranges for i and j at back-edge $3 \rightarrow 2$ without widening; (c) ranges for i and j at back-edge $3 \rightarrow 2$ with widening [3]

3.8 Improvements

There are different techniques to improve abstract interpretation result. Widening is a technique that is used to improve the abstract interpretation performance. Affine relation analysis is a technique used by VSA to improve the accuracy. In this section we show these two techniques are used by ASA and VSA.

3.8.1 Widening

Widening is an extrapolation technique used to ensure the termination of abstract-interpretation algorithms with lattices of infinite, or very large, height. Both ASA and VSA use widening to reduce the analysis time for loops. ASA apply a simple widening by expanding integer intervals to $[-\infty, 0]$, $[0, \infty]$ or $[-\infty, \infty]$ after inspecting the abstract values of the first two iterations of the loop.

VSA defines the widening operator (∇) for intervals as follows:

$$[l_1, u_1] \nabla [l_2, u_2] = [l, u], \textit{where}, l = \begin{cases} l_1 & l_1 \leq l_2 \\ -\infty & \textit{otherwise} \end{cases}, u = \begin{cases} u_1 & u_1 \geq u_2 \\ \infty & \textit{otherwise} \end{cases}$$

VSA only apply widening to the back-edge of loops. For example, in figure 14(a) the widening operator is applied to edge from node 3 to node 2. The ranges computed with widening for variables at each iteration is shown in figure 14(c). By using widening, interval analysis terminates faster when compared to figure 14(b) where the widening is not applied.

3.8.2 Affine Relation Analysis

One of the main drawback of VSA is the loss of precision due to non-relational nature of VSA abstract domain. ASA can maintain basic relation between

```
typedef struct {
    int x,y;
} Point;

int a = 1, b = 2;

int main(){
    int i, *py;
    Point pts[5], *p;
    py = &pts[0].y;
    p  = &pts[0];
    for(i = 0; i < 5; ++i) {
        p->x = a;
        p->y = b;
        p += 8;
    }
    return *py;
}
```

```
        proc main         ;
   1    sub esp, 44       ;Allocate locals
   2    lea eax, [esp+8]  ;t1 = &pts[0].y
   3    mov [esp+0], eax  ;py = t1
   4    mov ebx, [4]      ;ebx = a
   5    mov ecx, [8]      ;ecx = b
   6    mov edx, 0        ;i = 0
   7    lea eax,[esp+4]   ;p = &pts[0]
 L1:    mov [eax], ebx    ;p->x = a
   8    mov [eax+4],ecx   ;p->y = b
   9    add eax, 8        ;p += 8
  10    inc edx           ;i++
  11    cmp edx, 5        ;
  12    jl L1             ;(i < 5)?L1:exit loop
  13    mov edi, [esp+0]  ;t2 = py
  14    mov eax, [edi]    ;set return value (*t2)
  15    add esp, 44       ;Deallocate locals
  16    retn              ;
```

Figure 15: A sample code to traverse an array of struct [3]

abstract values if they are equal ($v_1 = v_2$) or have a difference of constant value ($v_1 = v_2 + c$). To recover some of the losses in precision, VSA uses an auxiliary analysis, ARA, to track relations between registers.

An integer affine relation among variables $r_i (i = 1...n)$ is a relationship of the form $a_0 + \sum_{i=1}^{n} a_i r_i = 0$, where the a_i represents integer constants. VSA uses Affine Relation Analysis (ARA) to improve the interpretation of conditional-branches and widening operation. Figure 15 shows a sample program and corresponding assembly code. By running VSA algorithm for this code, we get abstract values for registers and memory locations at each point of program. Figure 16 shows the VSA result in Instruction `L1, 8` and `14`. VSA cannot make a precise estimation for upper bound of eax at instruction `L1`. Looking carefully figure , we can infer there is an affine relation between `eax, esp` and `edx` in the form of $eax = (esp + 8 \times edx) + 4$. When the true branch of condition `jl L1` is interpreted, `edx` is bounded on the upper end by 4; hence it is represented by value-set $([0, 4], \perp)$ at instruction `L1`. The value of `esp` at the same place is $(\perp, -44)$. Putting all the these value-sets and solving the affine relation yields

$$eax = (\perp, -44) + 8 \times ([0, 4], \perp) + 4 = (\perp, 8[-40, -8])$$

In this way, a more precise value for upper bound of `eax` at `L1` is obtained that is impossible to infer by simple VSA without ARA.

3.9 Summary

In this chapter, we focus on using abstract interpretation for static analysis of binaries. This chapter introduces two abstract interpretation techniques, called ASA and VSA. Although ASA and VSA use abstract interpretation to assist

Instruction L1 and 8	Instruction 14
esp $\mapsto (\bot, -\mathbf{44})$	esp $\mapsto (\bot, -\mathbf{44})$
mem_4 $\mapsto (\mathbf{1}, \bot)$	mem_4 $\mapsto (\mathbf{1}, \bot)$
mem_8 $\mapsto (\mathbf{2}, \bot)$	mem_8 $\mapsto (\mathbf{2}, \bot)$
eax $\mapsto (\bot, \mathbf{8}[-\mathbf{40}, \mathbf{2^{31}} - \mathbf{7}])$	eax $\mapsto (\bot, \mathbf{8}[-\mathbf{40}, \mathbf{2^{31}} - \mathbf{7}])$
ebx $\mapsto (\mathbf{1}, \bot)$	ebx $\mapsto (\mathbf{1}, \bot)$
ecx $\mapsto (\mathbf{2}, \bot)$	ecx $\mapsto (\mathbf{2}, \bot)$
edx $\mapsto (\mathbf{1}[\mathbf{0}, \mathbf{4}], \bot)$	edx $\mapsto (\mathbf{5}, \bot)$
edi $\mapsto \top^{vs}$	edi $\mapsto (\bot, -\mathbf{36})$
var_44 $\mapsto (\bot, -\mathbf{36})$	var_44 $\mapsto (\bot, -\mathbf{36})$

Figure 16: VSA result on sample code [3]

static binary analysis, they try to address different challenges. Thus, they have different abstract domains and different abstract operations. ASA is designed to make an abstract summery for functions and it only cares about local variables. ASA is more interested in capturing simple linear relation between abstract points.

On the other hand, VSA is more interested in global analysis. VSA tries to address the indirect memory access challenge, thus it uses SI that is more accurate than simple interval. VSA loses some precision because of using SI as non-relation abstract representation. Affine relation analysis is used to recover some of these losses.

We describe property of interest, abstract domain and operations of abstract domain for ASA and VSA. Finally, widening and affine relation analysis are discussed as two improvement methods for ASA and VSA.

References

[1] Krzysztof R. Apt. *Principles of constraint programming.* Cambridge University Press, 2003.

[2] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. In *Network and Distributed System Security Symposium*, February 2011.

[3] Gogul Balakrishnan. *Thesis: WYSINWYX: What You See Is Not What You Execute.* PhD thesis, University of Wisconsin Madison, USA, 2007.

[4] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. *Proc. Int. Conf. on Compiler Construction, Springer-Verlag, New York, NY*, 2004.

[5] Gogul Balakrishnan and Thomas Reps. Divine: Discovering variables in executables. In *Verification, Model Checking, and Abstract Interpretation*, pages 1–28. Springer, 2007.

[6] Gogul Balakrishnan and Thomas Reps. WYSINWYX : What You See Is Not What You eXecute. *ACM Trans. on Program. Lang. and Syst.*, 2009.

[7] Patricia Mary Benoy. *Polyhedral Domains for Abstract Interpretation in Logic Programming.* PhD thesis, University of Kent, Canterbury, UK, 2002.

[8] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural static slicing of binary executables. In *Proceedings of the International Conference on Software Maintenance*, pages 188–, Washington, DC, USA, 1997. IEEE Computer Society.

[9] Patrick Cousot. Abstract interpretation. Technical report, http://www.di.ens.fr/ cousot/AI/, 2008.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[11] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 12–24, New York, NY, USA, 1998. ACM.

[12] Jr H.S. Warren. *Hacker's Delight.* Addison-Wesley, 2003.

[13] Yuan-Shin Hwang, Tzong-Yen Lin, and Rong-Guey Chang. Disirer: Converting a retargetable compiler into a multiplatform binary translator. *ACM Trans. Archit. Code Optim.*, 7:18:1–18:36, December 2010.

[14] N.D. Jones and F. Nielson. *Abstract interpretation: a semantics-based tool for program analysis.* Oxford University Press, 1995.

[15] Jonghyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[16] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Network and Distributed System Security Symposium*, February 2010.

[17] Alan Mycroft. *Type-Based Decompilation.* 8th European Symposium on Programming, ESOP99, 1999.

[18] Padraig OSullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos Keromytis. Retrofitting security in cots software with binary rewriting. In *Future Challenges in Security and Privacy for Academia and Industry*, IFIP Advances in Information and Communication Technology, pages 154–172. Springer Boston, 2011.

[19] Jean-Francois Rask Patrick Cousot, Pierre Ganty. Fixpoint-guided abstraction refinements. In *14th Int. Symp. on Static Analysis*, 2007.

[20] B. C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[21] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 119–132, New York, NY, USA, 1999. ACM.

[22] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code. *ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation*, page 100, 2006.

[23] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 74–83, New York, NY, USA, 2008. ACM.

[24] Asia Slowinska, T. Stancescu, and Herbert Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium*, 2011.

[25] M.J. Van Emmerik. *Static Single Assignment for Decompilation.* PhD thesis, The University of Queensland, Australia, 2007.

[26] Z.L.X.Z.D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. *In Proceedings of the 17th Network and Distributed System Security Symposium*, 2010.