2019

Static Evaluation of Type Inference and Propagation on Global Variables with Varying Context

Ivan Frasure
Wright State University

STATIC EVALUATION OF TYPE INFERENCE AND PROPAGATION ON GLOBAL VARIABLES WITH VARYING CONTEXT

A Thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Engineering

by

IVAN FRASURE
B.S.C.E., Wright State University, 2018
B.S.C.S., Wright State University, 2018

2019
Wright State University

WRIGHT STATE UNIVERSITY
GRADUATE SCHOOL

May 3rd, 2019

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Ivan Frasure ENTITLED Static Evaluation of Type Inference and Propagation on Global Variables with Varying Context BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Computer Engineering.

_____
Michelle Cheatham, Ph.D.
Thesis Director

_____
Mateen Rizki, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on Final Examination:

_____
Michelle Cheatham, Ph.D.

_____
John Gallagher, Ph.D.

_____
Mateen Rizki, Ph.D.

_____
Barry Milligan, Ph.D.
Interim Dean of the Graduate School

ABSTRACT

Frasure, Ivan. M.S.C.E., Department of Computer Science and Engineering, Wright State University, 2019. Static Evaluation of Type Inference and Propagation on Global Variables with Varying Context

Software reverse engineering (SRE) is a broad field with motivations ranging from verifying or documenting gordian source code files to understanding and reimplementing binary object files and executables. SRE of binaries is exceptionally compelling and challenging due to large amounts of information that can be lost in the compilation progress. A central area in SRE is type inference. Type inference is built around a fundamental step in understanding the behavior of a binary, recovering the types of data in the program. Type inference has many unique techniques in both static and dynamic type inference systems that have been implemented in more than forty approaches.

The problem has been noted in literature that evaluation and testing is difficult in software reverse engineering due to various challenges like closed-source tools, commercial fees, inconstancy of data being tested; a 2016 survey noted many of these tools cannot be compared against each other, or introduce techniques that would be beneficial to evaluate in other situations. This survey noted the need for additional work to focus more on specific techniques in the hopes of generating better environments to test approaches in, or compare against, even if there is no access to the tool.

This lightweight configurable approach evaluates the well-known techniques of flow-sensitive, context-sensitive, type inference based on instructions and type propagation, however, it works to isolate these techniques and compares how they changed with additional information. With this in mind, all the indicators are configurable as means to help engineers who are interested in evaluating the effectiveness of an indicator within a configuration or technique.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

I. INTRODUCTION

1.1 REVERSE ENGINEERING

Software reverse engineering (SRE) is a broad field with motivations ranging from verifying old source to studying binaries to determining how to recreate, patch, or test a binary executable. In order to accomplish these things, engineers need to understand the binary's behavior. Type inference is built around a key component to understanding a binary -- understanding the types of its contents. There are many techniques in type inference that are implemented in over forty approaches.

1.2 BACKGROUND

Current research in the type inference field covers a broad range of approaches, which are discussed in Chapter 2. A major problem with existing research on this topic is that many publications combine multiple techniques into one approach in order to achieve the highest overall type inference results, and the evaluations are not fine-grained enough to identify which of these techniques contribute most prominently to the overall accuracy. The result is heavy approaches that may contain significant inefficiencies.

1.3 PROBLEM STATEMENT

This problem has been noted in the literature. Evaluation and testing are especially difficult in software reverse engineering due to various challenges, including closed-source tools, commercial fees, and inconstancy of datasets. The 2016 survey paper by Caballero and Lin notes the need for additional work to focus on specific techniques

in the hopes of generating a better environment in which to test approaches, iterate improvements on techniques, and compare against other tools' results. Essentially, research needs to be done to evaluate specific isolated techniques so that the valuable aspects of the technique can be improved, and any negative aspects can be limited or removed.

1.4 RESEARCH OBJECTIVES

The goal of the work presented here is to empirically analyze the performance of several base type inference techniques in order to observe the contribution of each aspect in isolation. This is accomplished through the creation of a lightweight evaluation framework that is more extensible and far more configurable than typical approaches that are focused on coarser results. This approach should result in something that is useful for understanding the underlying mechanisms of a specific technique and observing the interaction between the technique and a specific configuration of the approach.

The technique selected for evaluation is well known in literature; a flow-sensitive context-sensitive type inference based on instructions and type propagation. In order to keep the scope manageable, the flow-sensitive aspect of the approach is held constant, and the accuracy of type inference with respect to base type, size and sign of variables is analyzed in the presence of varying amounts of context. The type inference aspect of the approach is isolated by only evaluating on a single instruction without propagating types (though if the same variable is used again elsewhere it will count towards its score since it is a direct reference). The utility of type propagation aspect of the technique is then added through inclusion of a single context instruction. Finally, this is repeated with a second context instruction to see if there is a benefit to additional context.

The research here focuses on global variables, which are particularly interesting due to the difficulties they present for type inference techniques that rely heavily on flow analysis. Future work can continue the analysis to include local variables.

1.5 ASSUMPTIONS, LIMITATIONS AND SCOPE

A few aspects of the analysis are handled through a binary analysis platform called Binary Ninja (Binja), most notably disassembly the binary and generation of a control-flow graph. Thus, the approach is limited to the support the platform offers for a given architecture.

It is assumed that the binaries are not compiled with debug information, and that the disassembly of the binary is done correctly by the platform. Also, in practice, extending the range of context should mean reevaluating the feature weights and scoring to take full advantage of the additional information and achieve the highest type inference accuracy possible; however, these values are held constant in order to provide an apples-to-apples comparison among the different approaches.

1.6 OVERVIEW

Chapter 2 describes current techniques and analysis types, Chapter 3 discusses the implementation details and configuration of the tests, and Chapter 4 presents the results of the experiments. Chapter 5 summarizes the results and discusses potential future work on this topic.

II. BACKGROUND

There are a lot of choices made in type inference approaches, this chapter will review features provided and techniques used that are in other tools. This overview of techniques and features will provide a background for the choices that were made in this paper's implementation of type inference techniques. Section 2.1 PLATFORM FEATURES will cover platforms, existing programs other tools build on to do type inference. Section 2.2 TYPE INFERENCE will cover the various analysis types and techniques that needed in order to do type inference. Section 2.3 TYPES OF DATA TYPES covers the type of data that can be recovered by the various inference techniques. Section 2.4 EVALUATION AND METRICS provides an overview of some the practices used in programs.

2.1 PLATFORM FEATURES

Platforms are existing programs or tools that are used by other tools to help solve or accomplish some part of the type inference problem. Usually, only a few critical parts of a platform's capabilities are used as a basis for the new tool to be developed. There are several platforms that are used in various type inferencing tools. Most often a platform provides some sort of analysis feature, and perhaps an API or library to assist with it like IDA [1] or Binary Ninja [2]. Other times it is a separate script or tool that gathers information for additional analysis like dynStruct's utilization of DynamoRIO [3] [4].

2.1.1 DISASSEMBLY

Platforms that are used in other approaches typically provide disassemble of the binary, which is a necessity for static tools. Disassembly is a difficult problem with its own problem space [5], and so it is only mentioned as it relates to type inference. Disassembly in software reverse engineering is the process of taking a binary and attempting to translate the binary sequences into disassembly text (comprised of data and machine code, also called disassembly or assembly code) [6]. All static approaches to type inference require disassembly, whether a tool attempts it or relies on a platform. Most type inference tools that function on disassembled binaries utilize a platform [7].

2.1.2 INTERMEDIATE REPRESENTATION

Once disassembled, some platforms take additional steps to raise the disassembly code to even more human-readable text by implementing Immediate Representations (IR), sometimes called Immediate Languages (IL). Some of the IR platforms require the usage of its IR when developing tools based on that platform, like the representation LLVM (a platform and an IR by definition) [8] and the tool DDT, a type inference approach that is built on LLVM [9]. While in other platforms, using the IR provided by the program is optional, but can allow for additional features offered by a platform [10] [11].

2.1.3 STATIC PLATFORM FEATURES

Some platforms focus on providing static analysis features and requirements including disassembly, static slicing, identifying basic blocks, functions, dominator graphs, and control graphs among other abilities like Binary Ninja and SecondWrite [2] [12]. A few tools or platforms narrow in on specific applications of type inferencing and provide

capabilities such as leveraging knowledge of protocols with statically stored data to makes inferences about the data [13] [14].

2.1.4 DYNAMIC PLATFORM FEATURES

The other crucial approach supported by platforms is dynamic analysis. It includes features like setting hooks for libraries or function calls and inserting debugging operations into the binary; these features are generally known as dynamic binary instrumentation and popular with platforms such as PIN, Valgrind, or DynamoRIO [4] [15] [16]. Dynamic approaches also allow for significantly better memory tracking capabilities like making memory graphs, doing shape analysis on binaries [17], and classifying patterns in memory usage to detect malware in spite of obfuscation [18]. While shape analysis can be done statically, it has only been done on source code, and even then, it was computationally expensive. Additionally, shape analysis may still have difficulty with overlapping structures [19] [2.3.2 RECORDS & ARRAYS].

2.1.5 PLATFORMS SCOPE AND LIMITS

While tools typically utilize fewer features and capabilities than are offered by a platform, platforms can still be a limiting factor. If a tool relies too heavily on a platform that does not keep up with or support instruction sets and architectures, the tool's own support or ability to adapt to new architecture can be limited, though that can be an approach's goal [2.4.3 ARCHITECTURES CHOICES].

The table below [**Error! Reference source not found.**] is a modified version (changes in the table description) of the table from Caballero and Lin 2016 and lists platforms that have been used by type inference tools with various information about the architectures and OSs supported.

Platform	IR/IL	Static Analysis	Dynamic Analysis	X86	X86/64	ARM	MIPS	SPARC	PowerPC	Window/PE	Linux/ELF
BAP	☑	☑	☑	☑	☑	☑	☒	☒	☒	☑	☑
BitBlaze	☑	☑	☑	☑	☒	☒	☒	☒	☒	☑	☑
Boomerang	☑	☑	☒	☑	☒	☒	☑	☑	☑	☑	☑
CodeSurfer/x86	☑	☑	☒	☑	☒	☒	☒	☒	☒	☑	☑
Dyninst	☒	☑	☑	☑	☑	☒	☒	☒	☒	☒	☑
IDA	☑	☑	☒	☑	☑	☑	☑	☑	☑	☑	☑
iDNA	☑	☒	☑	☑	☑	☒	☒	☒	☒	☑	☒
LLVM	☑	☑	☒	☒	☒	☒	☒	☒	☒	☒	☒
PIN	☒	☒	☑	☑	☑	☒	☒	☒	☒	☑	☑
QEMU	☑	☒	☑	☑	☑	☑	☑	☑	☑	☑	☑
ROSE	☑	☑	☒	☑	☒	☑	☑	☑	☑	☑	☑
SecondWrite	☑	☑	☒	☑	☒	☒	☒	☒	☒	☑	☑
SmartDec	☑	☑	☒	☑	☑	☒	☒	☒	☒	☑	☑
Udisc86	☒	☑	☒	☑	☑	☒	☒	☒	☒	☒	☒
Valgrind	☑	☒	☑	☑	☑	☑	☑	☑	☑	☑	☑
DynamoRIO	☒	☒	☑	☑	☑	☑	☒	☒	☒	☑	☑
Binja Ninja	☑	☑	☒	☑	☑	☑	☑	☒	☑	☑	☑

Table 1: Platforms' IR, Architectures, and OS. Original table [7] has been modified to include Binary Ninja and DynamoRIO [4] [2], leave off release-types, in addition to other minor changes for appearance.

This is where platforms that heavily rely on an IL can be more beneficial, assuming the code that elevates an architecture type to an IL has already been created [2.1.2 INTERMEDIATE REPRESENTATION]. If this is not already the case, creating a lifter language can require significant effort, and until it has been made, tools that depend on the IL cannot analyze that instruction set. While approaches continue to meet the challenges in lifting to IR/IL [20], these difficulties can affect the ability of a tool to

extend support to different or new architectures. Referring to [Table 1] we can see various platforms' support for different architectures, but even beyond the platform's support, the survey revealed the majority of tools targeted C/C++ for two specific architectures (x86, x86/64). This makes sense considering its popularity [2.4.3 ARCHITECTURES CHOICES]. However, the survey also notes it would be interesting to see the tools expand typing into different areas (like using functional languages) to see how type inference approaches perform [7].

2.2 TYPE INFERENCE

There are a few techniques that are used in type inference that largely depend on what data a tool will inference; section [2.3 TYPES OF DATA TYPES] will cover types of data from an inference.

2.2.1 VALUE-BASED ANALYSIS

Value-based analysis (VBA) checks the contents of an executables' registers and current memory (based on dynamic analysis). Several tools use it for finding pointers and strings. It uses the strings Unix command to identify strings and checks register or memory values to determine if a value is an address in live memory to determine if it is a pointer. However, a flaw that Caballero and Lin note is that being in live memory is not enough to assume a data object is a pointer, as it could be an integer or string among other things. Monitoring the allocations and the stack carefully would improve VBA performance [7].

2.2.2 FLOW-BASED ANALYSIS

Flow-Based analysis is the dominating method in type inference in part because it can encompass both dynamic and static analysis based approaches. Dynamic approaches can intuitively use this approach through the execution as memory values are concrete, and

locations being accessed are easy to track and use in propagation [2.2.4 TYPE PROPAGATION]. However, in static approaches, something must alias or point-to a memory location so the approach can remember if the location is accessed later in the binary without relying on a concrete memory address. Value-Set-Analysis is one method that provides this ability to static approaches [APPENDIX A: ANALYSIS TYPES]. Flow-Based Analysis is comprised of two fundamental techniques: type inference from code [2.2.3 TYPE INFERENCE] and type propagation [2.2.4 TYPE PROPAGATION] through code [7].

2.2.3 TYPE INFERENCE

For type inference, there are two basic approaches, inference based on the instructions from the binary, and inference based on external functions.

Instruction based inferences can be tricky as the type being inferenced from the instruction varies per an implementation. This can be seen in how Mycroft uses XOR to denote integer, while in software that uses encryption XOR is used to encrypt many types of data [7] [21]. BITY, a binary classifier approach, also mentions errors or imprecision in typing approaches, such as a single bit variable (contained in a byte) being typed as a char or byte_t [22]. Caballero and Lin note that different tools, even using the same algorithm may have different type inferences for the same instruction. So, at the least, having a structured and formal approach to determining types an instruction should inference within a specific architecture would be valuable to complete so error can be calculated and reflect best approaches.

External function-based inference works similar to a sink in taint analysis [2.2.5 TAINT ANALYSIS]. A given function prototype has known types for parameters and return

values; these are used to type the information that passes into or from the function. Function inference can be applied to recover function prototypes [2.3.6 FUNCTION TYPES], which in turn can then be used to help type primitive data types [2.3.1 PRIMITIVE TYPES LATTICE].

2.2.4 TYPE PROPAGATION

Type propagation is using instructions that move information to determine something about the source or destination of the information. If one is unknown, the other can infer its type. BITY explains there are over 30 types of mov instructions in x86 and indicates that while movsd may be used to indicate a double, this is not always the case [22]. The opcode or mnemonic can incorrectly inference the type of a variable if the information being moved is larger than the instructions used can move in a single command [7]. This is possible for even basic data types, as long as a variable's size requires multiple instructions to move it, simple type propagation would provide an inaccurate depiction of the variable's size and would affect how its usage was perceived [22].

2.2.5 TAINT ANALYSIS

Taint analysis is a type of propagation that is used in various reverse engineering applications; it is flow-based and can be dynamic or static.

Static taint analysis (offline solving) is like static flow-based analysis in that it needs aliasing to track memory locations. It works by generating constraints by moving through the code path and solving them after it has been completed [7]. It can be used to track symbolic inputs to identify vulnerabilities or memory locations to inference types or used to complement online solving. An example of this is REWARDS, which does online solving but also includes an offline solver to build and solve constraints generated from

the log of the online part to resolve the type of any memory variables that were missed [23]. STILL is a purer static example of static taint analysis and uses it to statically find exploits in spite of obfuscation [24].

Dynamic taint analysis (online solving) types the source when the sink is reached solving constraints as instructions execute. Dispatcher is an approach that uses dynamic taint analysis and type inference to assist in reverse engineering protocols [25]. Sinks can be function calls and returns, which can provide both type information and semantic type. Even some semantic types are recovered with tainting, the tool REWARDS gathers timestamps, registry data, ports, and hostnames among other semantic data. Since REWARDS types the source as soon as a sink is located, it is classified as online solving [23].

2.2.6 UNIFICATION

Unification is a type propagation constraint technique that can be used to improve typing accuracy [21]. In cases like mov, if one type (the source or destination) is known the other is set to the same type. However, some dynamic unification approaches consider that some memory is used as temporary storage; an unknown source that is unified with a known destination may be incorrect since the destination last held information of a different type before being repurposed to temporarily hold new data. Tools have altered the approach in these cases to improve accuracy. For example, REWARDS only unifies if the destination is not a register; otherwise, it passes on the type without unification [23]. While ARTISTE goes further and points out the stack could also be used to temporarily store variables, and thus is not safer than registers [26].

2.2.7 MEMORY ACCESS ANALYSIS

Memory Access Analysis can be static or dynamic and can be used to find aggregate data types like records and arrays. Static memory access analysis is done by symbolically gathering the accesses, and solving the constraints designed for them. For a dynamic method, it allows analyzing the memory access in an execution. By observing the accesses, it becomes possible to determine structures like records and arrays in the memory. Though there are limitations to this approach if the accesses function with set bit sizes or use custom accessors, as well as concerns about scalability for dynamic memory access. Hybrid access analysis is presented to handle scalability [27], and Membrush works to reduce these limitations by discovering custom memory management functions and allowing other tools to utilize that knowledge in analysis [28].

2.3 TYPES OF DATA TYPES

Across the tools that exist in the type inference field, there are varying levels of types, structures, and other information inferenced. The information a tool infers is largely dependent on its end goal, but certain techniques used by tools for an end goal may require additional inference work either from a platform or by additional capabilities in the tool.

2.3.1 PRIMITIVE TYPES LATTICE

Primitive types are the ground level types in any program. These basic program types can include integers, doubles, floats, pointers, booleans, and characters among other types (pointer types, long integers, etc.).

A type lattice is a method to frame the type inference. Solving type constraints via the type lattice allows tools to measure their accuracy in terms of distance and

conservativeness. It is also beneficial for showing the relationship between primitive types. Figures [Figure 1] and [Figure 2] are offered as examples of primitive-type, type lattices from the tools TIE [29] and ARTISTE [26] respectively.

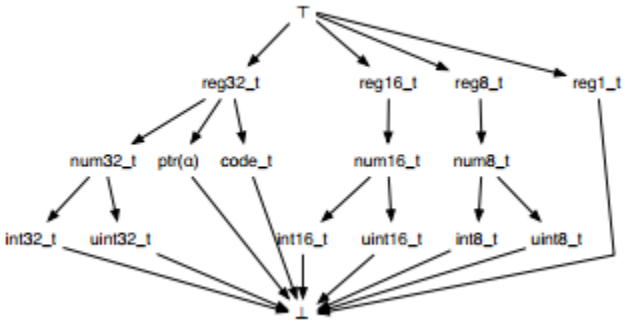


Figure 1: Primitive type lattice from TIE [29].

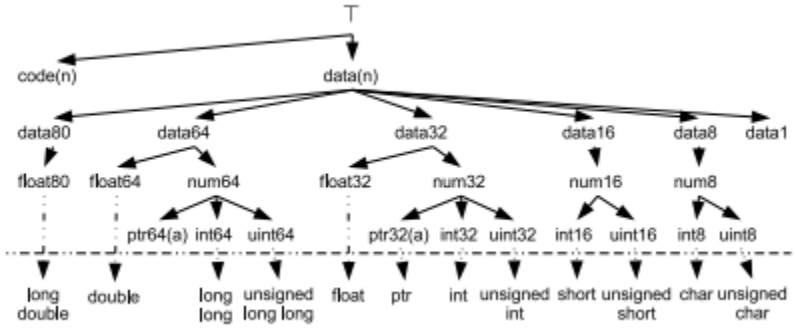


Figure 2: Primitive type lattice from ARTISTE [26].

There is a lot of thought into refining down to a primitive type, and the lattices provide valuable insight into the design and reasoning of a tool, which is why it is great when tools include a type lattice either for the benefit of the metric or the reasoning of the typing [7]. There are three types of value that can appear in a type lattice. First, design choices of what types are included can be observed in the structure of the lattice. Second, the manner in which types are refined can display some perspective on the tool. For

example, TIE uses registers as the first layer in its lattice and lists code as a subtype of a 32b register (reg32), while ARTISTE considers the separation of code and data first before refining data sizes in a similar way to TIE refining register sizes, indicating ARTISTE's more generalized perspective of a binary [29] [26]. Third, the level of refinement for the tool can clearly be seen in a lattice. Setting aside the difference in that ARTISTE supports 64b types, ARTISTE shows a higher level of refinement seen among the 32b types allowing for floats as a refinement of data32, whereas TIE does not. These examples are not to compare ARTISTE and TIE as tools or infer one has more value than the other, but rather to accentuate different values of including a primitive type lattice.

2.3.2 RECORDS & ARRAYS

Records and arrays are aggregate data types; these can have overlapping types. Records and arrays can be detected by both static and dynamic memory access analysis approaches since the accesses are usually reached by a base pointer with an offset, which is available in static instructions. Determining the size of the array or record is important to ensure data following the structure isn't incorrectly typed or excluded. This can increase the difficulty in static approaches, as loops can be used to process a structure and the loop count may not be a constant (e.g. determined dynamically) [7].

2.3.3 STRINGS

Strings are an array of characters terminated by a null. Being able to determine the difference between an actual character array and a string represented by one can be difficult. However, it can be done in several different ways. LAIKA uses a sort of value-based analysis by using the values in an unsupervised learning approach (a machine learning concept) [18]. X86/sa uses function type sources, and it works by identifying

libc functions that work with strings and then typing the data that is used by those functions [14]. REWARDS uses instruction type sources such as movs, stos, loads, and cmps to indicate strings [23]. These are few approaches to inferring a more precise and accurate typing for strings.

2.3.5 TREES AND LINKED LISTS

Dynamic shape analysis looks for links between memory locations that meet a shape constraint to find linked lists (forward pointers only), double linked lists (forward and backward pointers), and trees [7]. ARTISTE finds cycles, lists, and detects trees [26]. Mempick also employs shape analysis to determine specific types of data trees, though it does not distinguish between the purpose of specific trees of the same type (e.g. binary tree vs. binary search tree) [17].

2.3.6 FUNCTION TYPES

Function types are interesting to type because once a function type can inference the input variables and return values it can be used as a sink in taint analysis [23] [18], and type propagation [2.2.4 TYPE PROPAGATION]. Function prototypes do not yet support multivariant functions (a single function type having different possible inputs like printf()) [7].

2.4 EVALUATION AND METRICS

In the field of software reverse engineering, there is a lot of variety across tools in terms of the metrics used to evaluate the tool and the test sets on which the tool is run. Given there are many tools that are not open source comparing different tools becomes more difficult; especially considering the array of test binaries used. Even tools that work with the same benchmarks or test binaries can’t be compared very easily if different

metrics are being used. It has been suggested that tools work to use a more unified metric or also use popular benchmarks in addition to other test binaries that allow for better comparison [7].

2.4.1 BENCHMARKS AND TEST BINARIES

Test binaries used in type inference tools are largely arbitrary across the domain. According to the 2016 survey, 85% of tools chose to evaluate using benign binaries, though a few datasets stand out as being utilized by multiple tools [7]. The most common benchmark noted in the survey were versions of Coreutils, a collection of Linux command programs, and the other sets used were versions of SPEC CPU, a set of intensive computing programs. However, using benchmarks isn't without drawbacks. Some tools focus on recovering features that may not be prevalent enough for thorough testing. HOWARD notes that Coreutils' programs tend to be shorter than most software applications and thus aren't representative of most binary programs, though it tested on the Coreutils benchmark in addition to their other tests, which helps with comparisons [5].

2.4.2 EVALUATION METHODOLOGIES

Methodologies from the survey were categorized into three main groups. The first category contains approaches that didn't provide quantitative evaluations, including tools with no evaluation. These are tools that only compare against the verified types qualitatively [30], or tools that evaluate the overall result rather than the underlying typing techniques [7].

The benefit from evaluating how specific techniques contributed to the overall success of a tool would be that following effort could try to hone techniques or choose to combine

techniques in an approach based on the techniques' individual performance. This may reveal techniques that prove more useful in specific applications or worse make results more confusing, but without evaluations on specific techniques this is not possible.

The second category makes comparisons against the verified types quantitatively; however, this category is susceptible to type overlap from primitives that are within arrays, records, or unions. The survey denotes approaches that can handle primitive typing but don't identify or check for type overlap as course-grained quantitative evaluations.

The last category employs additional metrics, which the survey terms as fine-grain quantitative evaluation. Type lattices are an example of fine-grained metrics and are very useful in this regard [2.3.1 PRIMITIVE TYPES LATTICE].

Cross-Comparison between previous efforts is uncommon due to the difficulty in comparing using the different data sets and metrics noted above [**Error! Reference source not found.**] [2.4.1 BENCHMARKS AND TEST BINARIES]. The survey mentions only three of thirty-eight approaches that compare their typing results against other previous type inference approaches [29] [26] [12].

2.4.3 ARCHITECTURES CHOICES

This section will briefly cover typical scopes that appear in several approaches.

These approaches scope down to the x86 or the x86/64 architecture and the Linux or Windows operating system. Platforms are not typically a limiting factor in a tool [2.1 PLATFORM FEATURES], and this holds true for most x86 and x86/64 approaches in

the survey [7]. A scope like this can be present for a few reasons, but we will just focus on two of the more impactful motives.

The first, and perhaps the most influential motive, is the prevalence of these architectures in practice. Even without considering the use of other versions of Windows and the various popular Linux distributions, x86/64 based Windows 10 is predicted to be 75% of the professional computer market in two years [31]. This allows an approach to operate with all the binaries that are associated with that architecture.

Second, and more specific to the field, is that the more type inference approaches that support a specific architecture, the better comparisons of techniques or approaches will be. As mentioned above, this will promote the type inference field and assist in refining various techniques if the metrics are also present [2.4.2 EVALUATION METHODOLOGIES].

However, there are significant reasons why an approach may not implement in x86 or x86/64, such as if the tool is developed for a specific application, or if their motivation focuses specifically on one architecture. Additionally, while it is nice to be able to easily have high-resolution analyses on techniques and approaches, a tool having the ability to port architecture is still different from a tool being implemented in an architecture. IR and IL can help with some of these difficulties by providing a level field for comparison across techniques, but generalization has its costs [2.1.2 INTERMEDIATE REPRESENTATION].

III. METHODOLOGY AND IMPLEMENTATION

This chapter covers the decisions and methods that were implemented in the approach to statically type inference global variables.

3.1 STATIC FLOW-SENSITIVE

There are several considerations that go into choosing whether to implement a dynamic or a static approach, and in analyzing isolated techniques of type inference the choice largely depends on what techniques are going to be reviewed. Here we are interested in a static flow-sensitive context-sensitive type inference and propagation technique for global variables. Since dynamic approaches are intuitively flow-sensitive it is more interesting to isolate and observe the results of static flow-sensitive work. Additionally, while type inference and type propagation are equally applicable in static and dynamic approaches, a flow-sensitive static approach provides a better baseline evaluation of these techniques than a dynamic approach as explained below.

Static approaches often have better code coverage than dynamic methods, but if the static analysis is flow-sensitive the type inferences for a given variable can be based on the same (minimal) instruction information available to a dynamic approach. Aside from code coverage, a dynamic approach has more information available than a static flow-sensitive approach in that it has access to concrete memory addresses and values in the registers. Also, in static approaches loops can be confusing, while a dynamic approach

has known loop termination points. Overall, a static flow-sensitive approach it should perform more accurately than in a dynamic approach since more information is available.

3.2 PLATFORM: BINARY NINJA

The Binary Ninja (Binja) platform was chosen as the base for the static flow-sensitive type inference approach.

Binja is already dedicated to static analysis, and provides disassembly for several types of instruction sets, architectures, and operating systems variations. Support for x86/64 binaries was a necessity since the goal is to determine the effectiveness of type inference and propagation on typing global variables. The purpose of using x86/64 is to help the results be useful, since it is a popular architecture that many tools have implemented their approaches in, the results can better be communicated.

Additionally, Binja has a few useful techniques that enable better static analysis, including Single Static Assignment (SSA), which allows rejoining uniquely identified versions of variables after the variables split due to branch. Binja has varying levels of IR to provide options on how lifted an approach wants its binary; this approach uses the assembly version provided by Bina, which is essentially the source assembly instruction with some labelling for variables.

Binja applies its labelling to both local variables and global variables. The labelling of variables is unique within a function, or within the program if the variable is identified as a global variable. This can help with aliasing problems for local variables and enables cleaner type inference analysis overall. Binja also tags instructions with the label via propagation, however, this tracing of the label through registers is not used by this

approach, and future work plans to move away from even slight IR and toward pure assembly instructions to enable more control for extensibility and portability.

Another feature useful for this implementation is the easy inference and API that Binja offers to build plugins. IDAPro was considered for the same reasons, however, Binja was much more affordable for prototyping an approach. The implementation was developed as plugin for Binja and is launched from within its interface.

This approach, while focused on type inferencing for the sake of evaluating isolated techniques, also sought be mindful of extensibility and flexibility concerns so it could easily be expanded. Specific aspects of the implementation can be configured to highlight different information depending on the settings. The fact that Binja's API was written in Python was also desirable because Python is widely available on both Windows and Linux. Since many systems are supported by Python and it is popular, using Python should help with extensibility and flexibility. The following sections go into more detail on the implementation and the dataflow used to gather results.

3.3 IMPLEMENTING TYPE INFERENCE AND PROPAGATION

The critical component of the process is inferencing a type from an instruction, after this is done the type can be propagated, and additional instructions can be analyzed to contribute certainty to the typing. Three versions of inference were done to determine value. A simple instruction inference version, a context instruction inference version, and a secondary context instruction inference, which is interesting to consider in global type inference. Explanations of how the different methods are implemented are shown through sample instructions. The implementation details covered are important as there are ways type inference could be applied further without using a different technique. Note all

assembly instructions are in Intel assembly x86/64 format, so in mov instructions the destination is the first operand, and source is the second operand.

3.3.1 SINGLE INSTRUCTION

Inferencing on a single instruction, especially for globals, can provide bad inference information and obviously provides limited inference information.

```
mov     rax, qword [rel g_19]
```

Figure 3: A single instruction; in source code it is global unsigned 64-bit integer

With a single instruction the mnemonic (mov), the first operand (rax), and the second operand (qword [rel g_19]), can be used to determine information about the global variable in question (g_19). Allowing *mov* to infer type *int*, and *rax* and *qword* inferring 64-bits is the size, we can assume we have a 64-bit *int*. Additionally, *unsigned* can be included as an inference of *mov* as other variations exist of *mov* to preserve leading sign bits. These decisions are made configurable in this paper's approach to allow for an easy change depending on what an engineer wants to infer.

```
mov     rbx, qword [rel g_1362]
```

Figure 4: A single instruction; in source code g_1362 is a global pointer

However, g_19 could've also been a pointer, like g_1362 above. Without using additional information like compiler preferences, the unsigned 64-bit integer and the pointer look the same.

When two items are not the same size (unlike *qword* and *rax*) then sometimes additional information can be inferred.

```
movzx   eax, byte [rel g_1595]
```

Figure 5: A single instruction; in source code g_1595 is a global signed 64-bit integer

Here a 8-bit sized variable g_1595 is moved into a 32-bit register *eax*. The interesting thing is that *movzx* is called, which pads the extra room with zeros [32]. This may allow inferring *signed int*, however, the size inferred by this single instruction will be both 32-bit (from register size of *eax*) and 8-bit (from *byte*).

These Figures 1-3 above show a need for more information in an inference; however, single-instruction inference is still tested in the results section to measure the improvement in inference when using a context instruction.

3.3.2 CONTEXT INSTRUCTION

The context instruction is a second instruction that provides more context about the usage of an instruction. The context instruction is found for an access instruction; the access instruction is the first instruction in the pair. These two instructions provide more information about an instruction. Locating context instructions is usually straight forward, as shown below.

```
)1bd3  lea     rax, [rel g_44]
)1bda  mov     qword [rbp-0x48], rax
```

Figure 6: Access and Context Instruction pair; in source g_44 is a global 32bit integer (notice it is providing a 64bit address, more on that with secondary contexts)

For the above example, instruction *0x1bd3* is the access instruction, and *0x1bda* is the context instruction. In the access instruction, the address of *g_44* is loaded into *rax*. Now, to find a context the algorithm looks for the next instruction that uses *rax* (64bit register) or a sub-register associated with *rax* (e.g. *eax* and *al* among others). Many times, the context will be the register (or a sub-register) that was used as the destination in the access, simply propagated to the source of the context instruction like in Figure 4; *rax* is the destination of the access and the source in the context instruction. The context

instruction passes the data on to the local base pointer at offset *-0x48*, which is a local variable in the source code, note it's still a 64bit address thanks to the mnemonic *lea* despite being a 32bit integer in source, more on that later [3.3.3 ADDITIONAL CONTEXT].

If the reader is not as familiar with reverse engineering and with the relationship between registers like *rax* and *eax* a concise statement that might provide clarity is that *rax* controls 64bits, and *eax* controls 32bits, specifically the lower 32bits of *rax*.

Note in Figure 4 above, the context instruction happens to be the very next instruction, it is important to note that sometimes instructions are working with other registers between an access (first line, *edx* as destination) and context (last line, *edx* as source) like in Figure 5 below.

```
movzx   edx, al
movzx   eax, byte [rel g_41]
movzx   eax, al
mov     esi, edx
```

Figure 7: Access and Context instructions are separated by other instructions, working with other registers (eax)

Context works in both directions: if an access instruction's source is more interesting than its destination like in Figure 6, then the algorithm just goes the opposite way (this interest in terms of type inferencing of global variables can be determined based on what operand the global is in for a given instruction) .

```
000030cc  movzx   eax, al
000030cf  mov     dword [rel g_319], eax
```

Figure 8: Access and context where context is at a lower address than the access.

In Figure 4, it was discussed that usually an access instruction and a context instruction entirely or partially share the same bits within a register as the destination and the source (or vice versa) of the instruction pair.

However, Figure 7 shows a case in which the register used in an access and used in the following instruction are connected by the destination, rather than the source-destination or destination-source.

```
1e23  mov     rax, qword [rel g_787]
1e2a  mov     dword [rax], 0x1
```

*Figure 9: Pointers and types of pointers; g_787 is a global pointer, a int32_t**

The reason is that g_787 is a pointer, and while the address is passed to *rax*, the next instruction, rather than propagating the address as a source, stores a value at the address that is being pointed to by g_787. Whether 0xle2a is the context or the search for a context should end is a design choice. 0xle2a provides an opportunity to type what kind of pointer g_787 is, or to label it a generic pointer. The instruction *mov* and *dword* indicate a 32bit integer is located at the address that *rax* points to, and the source code confirms this and adds it is a signed integer. This approach works to identify generic pointers, but in the future, it will expand to include information about the usage, like *mov* and *dword* mentioned above, to determine the type of pointer.

3.3.3 ADDITIONAL CONTEXT

A second context instruction provides a third instruction for type inference. This is particularly interesting with globals in some instances. Take Figure 8 (the same image as Figure 4) for instance. In the source code g_44 is a 32bit integer, however, we can see at address 0x1db3 and the first context at address 0x1bda that it behaves like a pointer, even using *lea* to load the address of the variable rather than the contents at the address.

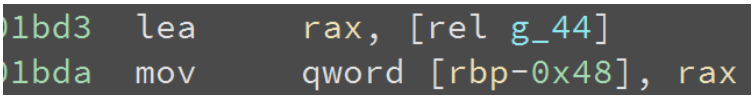


Figure 10: g_44 looks like a pointer; though in source it's a signed 32bit integer

Only using two instructions (the access and the first context), the approach would incorrectly infer this is a pointer. However, if varying levels of contexts are applied, things become more interesting; note the offset of the base pointer *rbp* -0x48. A second context is just a context for the first context instruction. The algorithm looks for a second occurrence of *rbp-0x48*. This happens at instruction 0x*1d0e*, where it is still an address.

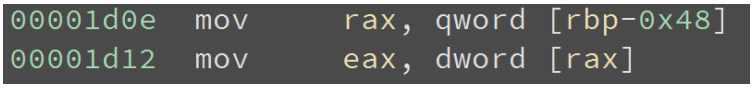


Figure 11: Many instructions after Fig. 8, a second context appears

However, note that the instruction address *0x1d12* finally accesses the value stored at the address in *rax*, and *dword* denotes the value is 32bits, which is correct. While more information is usually desirable, the work presented here will only test using one to three instructions to inform an inference to evaluate the effect.

3.4 INFERRED TYPES AND SCORING

When inferencing a primitive type from an instruction, three pieces of information are considered: the size of the variable, the core (or base type) of a variable, and the sign if appropriate. This process is from the tool DC, and it's approach to type inference [ref_2008_paper], which uses these three pieces of information to determine a variable's primitive type.

The goal of this approach is to establish a basis to enable the evaluation of many different techniques in the future. Because of this, the tool enables configuration so a 'none' category for components can be retrieved. Essentially, it allows engineers to decide if

they do not want to rely on a piece of information to provide a size, core, or sign. These configurations are done modularly, so an instruction, keyword, or register can provide any combination of the three components (core, size, sign) of a type as desired. Additionally, type components for a variable are scored so the more components for a type that appear the higher the score will be. The algorithm has a means to extract the highest scoring components and construct and fulfill a type constraint to generate a type for the variable. The fractional or soft scoring method provided is very useful to determine what components are impacting the scoring processing and allows for informed fine-tuning of a technique or inference constraints.

The next chapter, four, evaluates these techniques as described on the global variables. Since most programs do not use globals very heavily, CSmith a compiler testing system was used to generate binaries with many globals used as pointers, floats, and signed/unsigned integers [33].

IV. RESULTS

4.1 OVERVIEW

This chapter reviews the data and results collected from analysis. Section 4.2 discusses results from single instruction inference, 4.3 covers context, and 4.4 evaluates using a second context. The tests used for these comparisons are covered briefly in 4.1.1 and 4.1.2 before the other sections.

4.1.1 STATISTIC TESTS: CHI SQUARED TEST

Two tests are used through this chapter to determine statistical significance of a change in the accuracy between two techniques (SI vs SC etc.). The first test is the Chi-Squared test. For this test, two hypotheses are proposed; a null-hypothesis labeled (H0), and an alternative hypothesis (HA). Typically, H0 states that there is no interesting relation in the data being run through Chi-squared test; HA usually states there is some relationship between some of the data in the test.

The test takes a range of observed values and determines a distribution for those values according to probability those values could have occurred by chance. This is then expressed as a percentage known as a P-value. The P-value is then compared against the significance level, a value set before running the test that determines the threshold where H0 is rejected in favor for HA. For this chapter, the significance level is set to 0.05, a value commonly used for these tests. If the values in the test are 5% (or less) those values are unlikely to have occurred by chance, thus why H0 is rejected for HA. Chi-Squared

statistic test is commonly used for binary and categorical data and fit well with the data here. Another reason the Chi-squared test was selected is that the test doesn't require the use of normalized datasets (like t-tests would).

4.1.2 STATISTIC TESTS: FISHER'S (TWO-TAILED) EXACT

In some locations, Chi-squared is not appropriate to use with our data due to the frequency of an observation. In these cases, Fisher's Exact Test (two-tailed) is used to produce the P-value that is used in the same manner as the Chi-Squared test. The significance level for the Fisher's Exact Test (two-tailed) is also set to 0.05 throughout this chapter. The Fisher test is also very useful for binary and categorical data. Where either Fisher or Chi-squared test can be used, Chi-square will take precedence, though using either method can achieve reasonable similar P-values.

4.2 SINGLE INSTRUCTION INFERENCE

This section makes an inference with a single instruction (abbreviated SI for Chapter 4), instead of using two instructions or more. The flexibility of the configuration allows for us to analyze how effective specific instructions are by adjusting the weight contributed to the type inference score of the global variable by the instruction's features. As mentioned in Chapter 3, the core, type, and size can be adjusted individually for a given piece of information.

The single inference instruction results have three main categories to review: the core, the size, and the sign, as seen in Table 1. Within those three categorizes are interesting bits of relationships as to what type of errors appear, and how that can inform a reverse engineer when adjusting an inference constraint.

Table 2: (SI) Coarse-Grained Eval. of Categorical Correctness

	Core (%)	Size (%)	Sign (%)
All Components	27.46	83.45	52.11
Type Sources:	7	81	3

This table represents the accuracy across 284 global variables in 20 programs. The number of sources (type inference constraints) set for different instructions and pieces of information was varied to display the relationship between informed sources and results. As seen above, core was informed by only seven instructions (of some two-thousand in the x86/64 instruction set). Size was informed by nearly all the registers (about eighty). Sign was informed by three instructions for moving information.

The accuracy is bad, as expected, but it does have a lot of value relating to determining effective weights for instructions within the approach's configuration. Chi-squared test denoted there was some statistical relationship between components core, size, and sign. However, this relationship is better displayed with two components, so this will be explored further in the following sections. This is also expected for SI inferences because the results are more clearly reflective of the weights; when context additional constraints are involved, it is more difficult to evaluate the usefulness of particular configurations. It is also important to note that percentages include results of None instructions. Chapter 3 notes this feature to include "None" types to provide a measure of how information that is not being evaluated is affecting the results, though the option to take any typed information in preference to a "None" for a given variable exists to allow for more accurate comparisons against the verified types from the source code. This approach has not considered a means to assert base assumptions such as assigning all unknown values

of a certain size as integers of that size, though *mov*, used in these results to infer integer, can remove a lot of 'None' types.

4.2.1 SI CORES

There are three core types in the test set of globals (pointer, integer, float), and four possible values because 'None' is an option (with the aforementioned preference).

Table 3: SI Cores Type

	Pointers	Integers	Floats
Recovery:	7.53%	50%	42.86%
Population:	146	110	28
Type Sources:	1	4	2

Integers are detected 50% of the time, floats are detected 42% of the time, but pointers are only detected 7.5% with this configuration. Many types of global variables sometimes look like pointers, as seen in Chapter 3, when a global integer used a *lea* instruction. With only one instruction to determine the variable type it can be difficult in situations like this.

4.2.2 SI SIZES

There are four sizes considered in addition to 'None'; each one has about sixteen type sources.

Table 4: SI Sizes

	Size_8b	Size_16b	Size_32b	Size_64b
Percentage (%)	100	100	79.05	84.28
Population:	13	7	105	159

Looking at the size typing gives plenty of feedback for the configuration. The eight bit and sixteen bit sizes have particularly high detection rates. Some of this is likely influenced by how few globals there are in those categorizes, and how values with smaller sizes are used compared to larger sizes that can be pointers. This approach also built support for 128-bit (*xmm* registers), but the dataset did not use those.

4.2.3 SI SIGN

This category has three types: signed, unsigned, and this approach considered pointers to be a non-sign or None.

Table 5:SI Signed

	Unsigned	Non-Sign	Signed
Percentage (%)	26.67	95.80	0
Population	30	146	108

This category had the fewest sources and thus provides a large amount of information on the weight and changes that should be made to them, despite the actual accuracy being poor. The nature of global variables contributes to the ambiguity of the sign as well, since global variables are passed by address.

Starting with Unsigned, which were identified with 26.67% accuracy, there was an interesting observation that appeared in the system: all the cores and sizes associated with the unsigned types were correct (even though the accuracy was low). Statistical significance between the components will be discussed after this section.

For now, looking at Signed and Non-sign values in SI inference there are indications that there is a relation between being able to identify the core type for pointers, and that it can be observed in the poor accuracy in ‘Signed’ score evaluation.

Using the Chi-squared test some relationship between pointer cores and signed values were identified. The null-hypothesis (H0) was that there is no significance between the poor score of the ‘pointer’ detection and the poor detection of the ‘signed’ sign values. The alternative (HA) was that there is some significance to these scores. The test provided a 0.35% P-value, the probability there could be a relationship between the values by chance. The significance value chosen for the test was 0.05 (5%), which means the value 0.0035 was enough to reject the null-hypothesis. Thus, there is a statistically significant relation between the poor performances.

Looking at Non-sign instead of Signed reveals an even stronger statistical relationship between the high success of the Non-sign and low inference rate of pointer core types. H0 was there was no relation between the success of Non-sign and the poor performance of pointer inference. HA was that there was some relation. The significance level remained at 0.05, and the P-value returned was approximately 0%, a strong rejection of H0.

Looking at the errors at a high level, we see in the figure below most of the incorrect ‘Sign’ components were incorrectly noted as non-sign, and a few were incorrectly marked as unsigned.

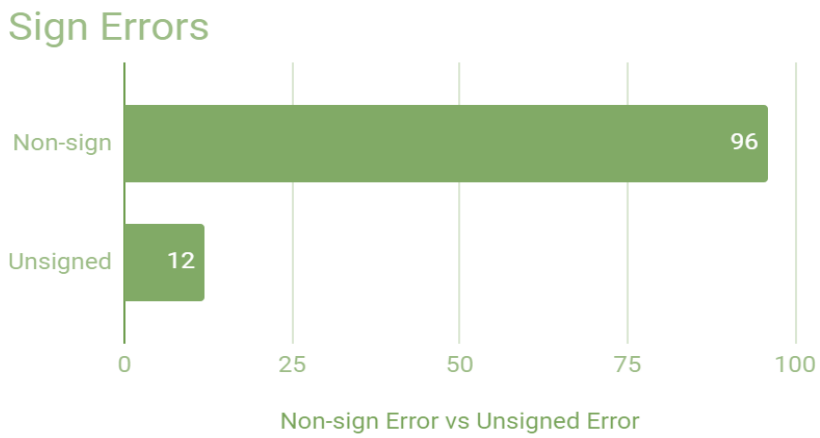


Figure 12: Why so many 'Non-sign' in Signed?

The next Figure shows the ‘Core’ component for each ‘Sign’ component that was incorrectly marked as a non-sign.

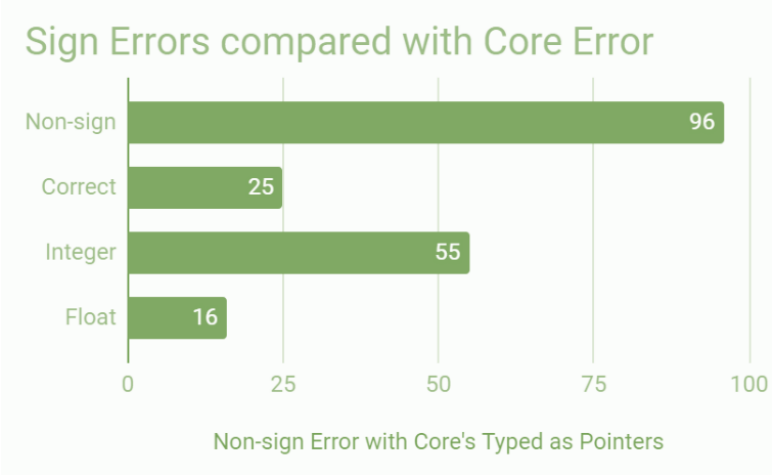


Figure 13: How many Non-sign values had correct cores?

Given that non-sign is a designed type for a pointer, it is little surprising that the previous errors in identifying pointers appear here. Both the integer and float columns were frequently incorrectly marked as pointers. There were 25 cases where the core was correct, but the ‘Sign’ component was still wrong. This is a great example of the

configuration system; it can be improved by changing the weights and reevaluating the results.

4.2.4 SI FLOATS, POINTERS, 32BIT GLOBAL INTEGERS

Table 6: (SI) Floats, Pointers, and 32bit Integers

	Core (%)	Size (%)	Sign (%)
int_32	12.7	92.06	0
Pointer	7.53	82.88	95.89
Float	42.85	39.29	0

The pointers produced interesting results in SI, given its size and sign information were so accurate to the detriment of other cores' signs. The signed 32bit integer also was interesting for its low core score, and the float seemed to manage a poor average, still with no sign accuracy. The configuration could benefit from additional core integer sources and redoing the weights on the sign system as is very clear at this point. Since the frequency for types with signed values was so low, Fisher's exact test was used to keep with best practice regarding frequency of occurrence. Fisher's exact test was set up, and the H0 was that there was no relationship between these types' poor overall categorization. HA was that there was some relationship or dependency between type categorization. The level of significance was the same as the Chi-squared test, 0.05. However, the result of the Fisher exact two-tailed test was 69.56%, there was very little to indicate statistical significance about the performance of three overall types listed.

4.2.5 SI 8BIT AND 16BIT GLOBAL INTEGERS

There was a subset of global variables that the configuration of this approach excelled in; 8bit and 16bit integers (numbering twenty from across the forty binaries). Seen in the table below, all the 8bit or 16bit size, integer cores that were unsigned were correctly identified.

Table 7: SI Performance on 8bit and 16bit Integers

	Core (%)	Size (%)	Sign (%)
uint_8	100	100	40.0
uint_16	100	100	40.0
Type Sources:	4	33	3

The ‘Sign’ component was not as accurate as the core and size, showing the benefit in a configurable frame as a means to improving technique. It highlighted a change that should be made in the configuration; specifically, it showed all the types going out were unsigned. This configuration’s bias towards unsigned types indicates that the current information source configuration needs to add weights and sources toward signed types and/or alter unsigned sources.

When examining for statistical significance, the threshold was set at 5%. H0 was defined as no relation between integer core detection and unsigned sign detection. HA was defined as a relation between integer and unsigned detection. Running both Fishers exact two-tailed test (2.45%), and Chi-squared test (2.28%) yielded a probability of chance that allowed rejecting the null-hypothesis (H0) and taking the alternative. The accuracy detecting integers was related to the accuracy in detecting unsigned values. Rerunning Fisher’s test provides stronger results that size is related to integers and unsigned with P-

value of about 0.000%. This shows that separately with integer vs sizes, and unsigned vs sizes the relations significant, allowing for the rejection of both null-hypotheses for integer vs size and unsigned vs size.

4.3 CONTEXT INSTRUCTION INFERENCE

The next step from single instructions is using access and context instruction(s), which is what is done in context-sensitive approaches. This section makes inferences on an instruction and a single context instruction (abbreviated SC). While the additional instruction makes analyzing the instructions and their weights or contributions to the type inference score less clear, with only one additional instruction it is still feasible.

Table 8: (SC) Category Accuracy

	Core (%)	Size (%)	Sign (%)
All Components	37.1024735	79.15194346	48.76325088
Type Sources: + Context Instruction	7	81	3

This data set is the same 284 globals variables from the same 20 programs used in the (SI) approach. There are a few interesting points compared to the SI section that appear in the Figure below.

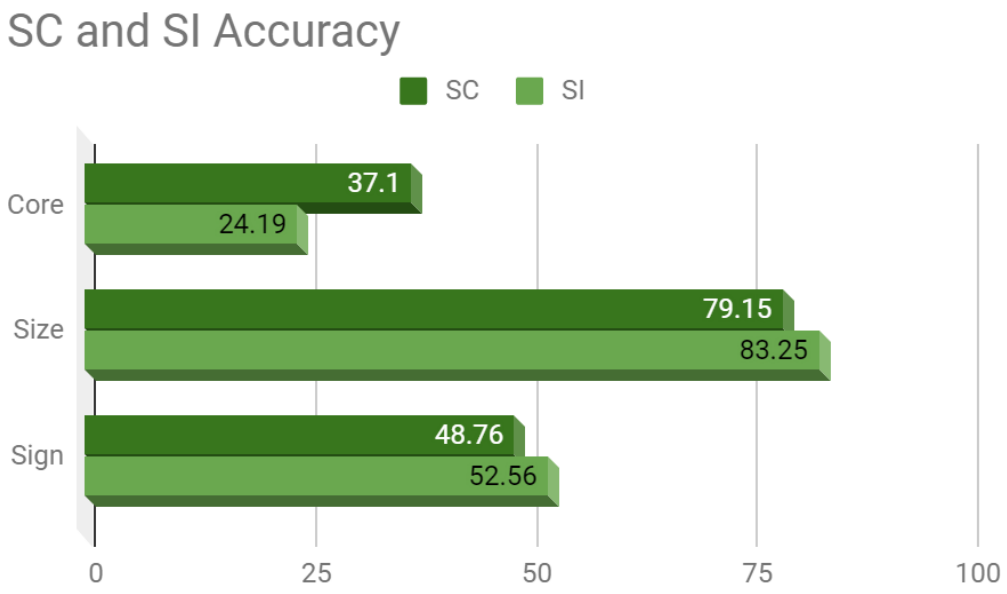


Figure 14: SC in dark green improves 13% in Core accuracy over SI in light green

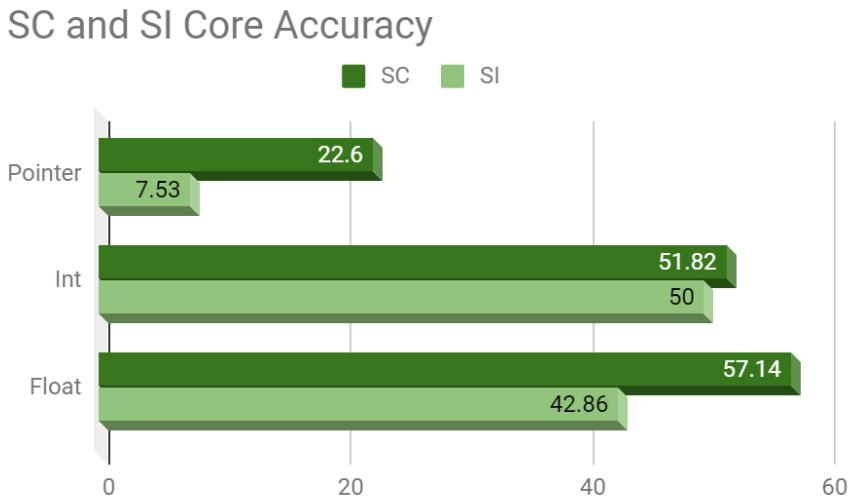
First, comparison shows the core accuracy improves by 13%. That is especially important for these globals, as globals of different cores are treated as pointers sometimes. This improvement is checked with the Chi-squared test for statistical significance with a 0.05 significance level. The result was a 1.2% probability the improvement was by chance; enough to reject the H0 that stated the improvement was by chanced given the significance level of 5%. The alternative was the improvement from using SC was statistically significant.

However, with that increase in core there were small decreases in both the size and sign accuracy. Size accuracy was reduced by 4% and sign decreased 3.8%. These were also run through the Chi-squared test. H0 was that size had no significant improvement in SI from SC. HA was that size was significantly improved. While the change overall was small internally there was a shift in the size that motivated this check. The result of Chi-squared test was 19.62%, no where near enough to reject H0. Thus, the size category did

not have a statistically significant chance from SI to SC. When applying the same test to Sign, the probability increased to 45%, not enough to reject H0 for sign either. So, the only statistically significant change of the categories overall was core.

4.3.1 SC CORE

With the three core types the increase denoted in the overall category graph is seen clearly below; though the configuration's accuracy is poor overall, adding an extra



instruction greatly boosts accuracy, especially in pointers.

Figure 15: SC improvement in Core accuracy

All three cores see considerable improvement, though the overall accuracy is still poor. Floats increase 14%. Pointers are nearly three times more accurate (15%) and still with only one indicator. Global 64-bit integers can be particularly difficult to tell part from pointers, so the pointer detection improvement is good to see. Though, integers only increased 1.8%, which may simply be near a limit of using such few indicators unless a lot more context statements were collected. The Chi-squared tests were run across all three core types separately, all with the significance level at 0.05, this showed that

integers were not significantly changed with probability of 78.74%, and floats though closer were also not statistically significantly changed. Pointers stood out with a probability of 0.032% (0.00032) that allowed for rejecting H0, and taking HA for SI vs SC pointers, which stated that pointers were significantly improved by the change in method.

4.3.2 SC SIZE

Size overall decreased 4%, and while there was serious decline in 32bit size accuracy (15%) it was not nearly as drastic as the changes within the Core component. In fact, 8-bit and 16-bit did not realize any change, so those were left out of the graph and added to a table summarizing the SC size accuracy.

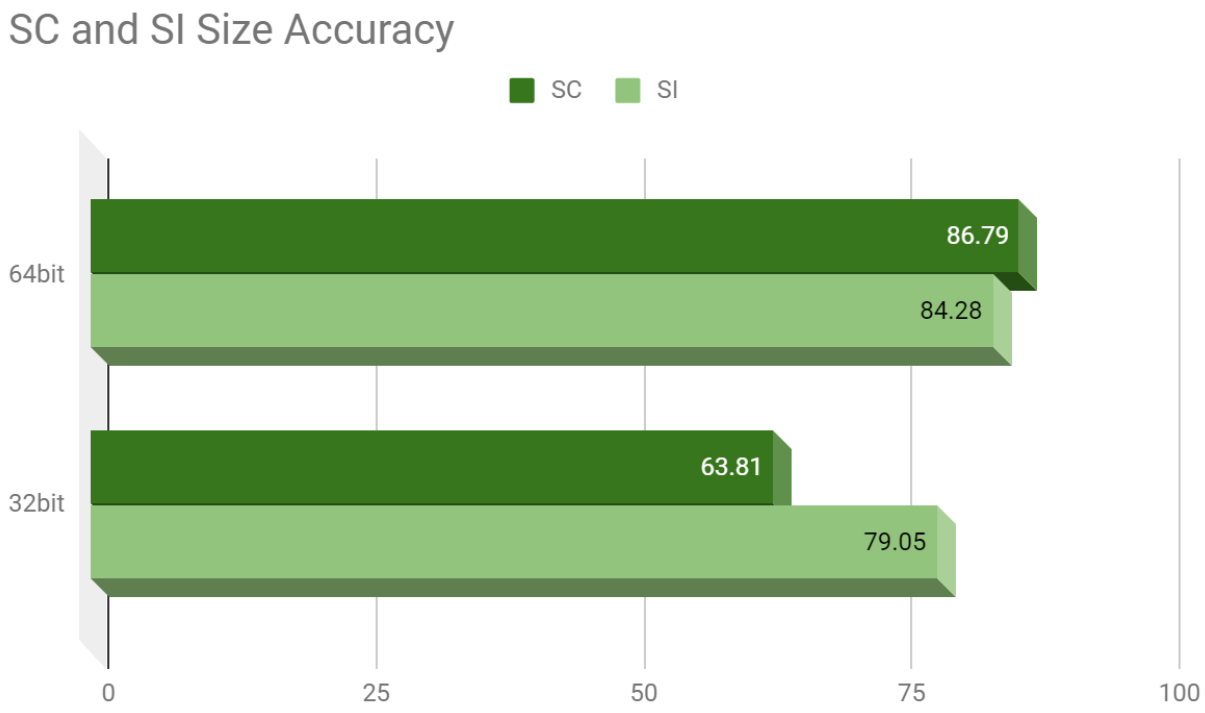


Figure 16: SC and SI Size Accuracy, significant 15% diffence in 32-bit size detection

Thus far, SC has been statistically significant in core, specifically pointer-core. All the other component changes were not significant, but even those changes usually were better for SC than SI. In this case, 32-bit size decreased a lot, so it was checked for significance. The significance level remained at 0.05 (5%). Chi-squared test gave P-value of 1.45% for comparing SI and SC for 32bit values, enough to reject the H0 there was nothing significant about the change in method. This is the first value that SI outperforms SC with statistically significance. The same test for 64bit resulted with a 52.3675% P-value, meaning while SC changed for the better, it wasn't a statistically significant improvement. SI is significantly better with 32bit sized values, than SC.

Table 9: SC Sizes

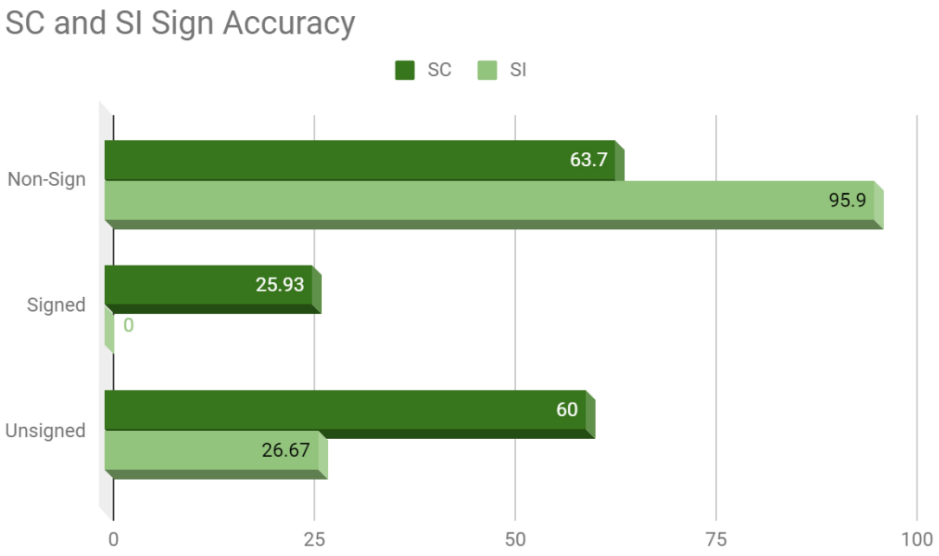
	Size_8b	Size_16b	Size_32b	Size_64b
Percentage (%)	100	100	63.81	86.79
Population:	13	7	105	159

Internally, SC has a statistically significant advantage in identifying 64bit values as compared to 32bit values. The Chi-squared test resulted with P-value less than 0.001, rejecting H0. While SI didn't have a significant difference between 32bit and 64bit that was due to the fact the detection rate was roughly equal in accuracy (the P-value was 27.7%).

This pattern across both SI and SC indicates that the configuration needs a greater diversity in inference sources, specifically the results show there is worth in focusing on improving 32bit values in SC mode.

4.3.3 SC SIGN

The Sign component for such a minor categorical change (only dropped 3.8% in the composite graph), had very a large flux internally as seen below.



Despite the overall category not undergoing a statistically significant change between methods, all three specific types had significant change. Fisher's exact test (two-tailed) was used in place of Chi-squared again to meet usage requirements. A large portion of the non-signed accuracy was reduced and was coupled with a lower accuracy on signed values; both signed and non-signed rejected the null-hypotheses that there was no significant change between methods, with P-values of less than 0.001.The unsigned accuracy also increased with statistical significance, the result of the Fisher exact test was a P-value of 1.82% under the maximum value 5% to reject H0.

4.3.4 SC FLOATS, POINTERS, 32BIT GLOBAL INTEGERS

Reevaluating the same variable types as SI allows another comparison of the components that benefit the most (pointers) and lost the most (32 bit signed integers). Even though integer accuracy increased some, the 32 bit size accuracy decreased, and signed accuracy increased less than unsigned. Table 9 shows the average score of global variables of the appropriate type recovered from the dataset. This means core, size, and sign are considered in context of only the variable type indicated, not all variables.

Table 10: (SC) Floats, Pointers, and 32bit Integers

	Core (%)	Size (%)	Sign (%)	Total (%)
int_32	15.87	82.54	41.27	46.56
Pointer	22.6	85.61	63.7	57.3
Float	57.14	3.57	0	20.24

Despite the increase in the core accuracy, the 32bit size loss of accuracy really did a trick of recovering the size of a float. All floats with incorrect cores (43%) also had incorrect size scores of 64bits, but as seen in the table above, only 3.5% of floats were correctly sized. Almost every float received an incorrect size.

Statistically there is a lot of interesting things taking place. The statistical test applied was the Fisher's exact test two-tailed, with a significance level of 0.05. Floats between SI and

SC were run first and the result as unable to reject H0, so there was no statically significant change. 32bit signed integers were tested next, and achieved a P-value of 5.76%, just shy of rejecting H0, but unable to make it. Pointers overall, just like when evaluating the pointer core alone, was able to reject H0 with a P-value of less than 0.1%. So, the improvement in the pointer type overall from using SC instead of SI was statistically significant.

4.3.5 SC 8BIT AND 16BIT GLOBAL INTEGERS

There were also improved results for unsigned types, so it will be worth reevaluating the table.

Table 11: SC Unsigned 8bit and 16bit Integers

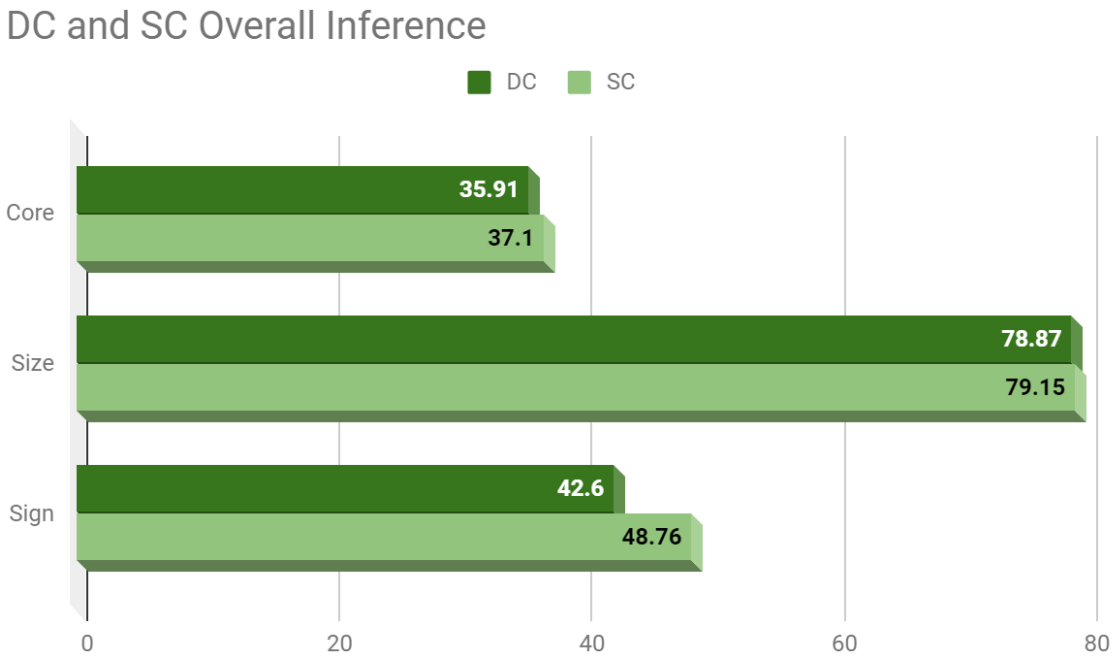
	Core (%)	Size (%)	Sign (%)
uint_8	100	100	100
uint_16	100	100	100

The large increase in unsigned accuracy impacted the uint_8 and u_int16 types. It closed the gap in the scoring department. Applying the Chi-square test to all the components reveals only unsigned was able to reject its null hypothesis and demonstrate statistical significance, with a P-value of roughly 1% (0.0092). Unsigned component was key in the improved the overall result.

This indicates a context-sensitive, flow-sensitive configuration, even with only a few informative type instructions sources, was able to retrieve those types with high accuracy.

4.4 SECONDARY CONTEXT

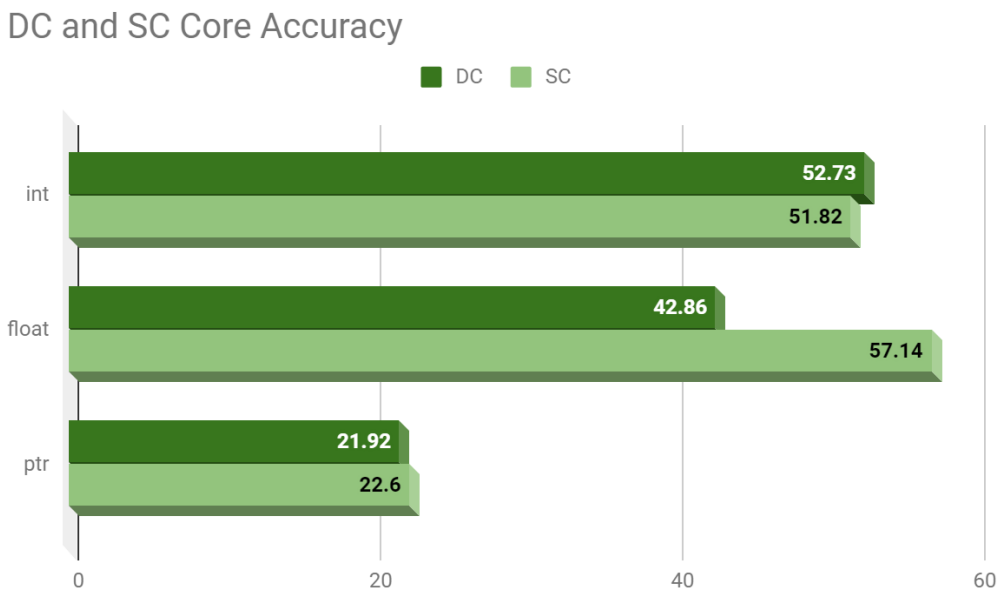
After running with one context, a secondary context or a double context (abbreviated DC for this chapter) was used to evaluate the impact of additional context on the results. If applied to locate variables the additional information would have to be collected carefully since there would be a chance of rescoring variables multiple times; doubling the weight of the instruction without increasing the information that informs an inference. Globals will tend to provide new information when retrieving a secondary context, because only instructions where the globals are accessed directly are used as the first context. A lot of other instructions exist that are not used immediately. The extra information could provide an opportunity to type specific pointers with the technique. However, a serious problem is determining what instructions to weight and how much. This difficulty affects the overall results; without adjustment additional information can confuse the system.



There is an initial decrease in accuracy, about an 1% for the Core and Size components. Sign accuracy decreased a bit more, falling 6%. Testing the change in the Core category for a significant difference between the DC and SC methods at the top level, the chi-squared test is used with a significance level of 0.05. The P-value for DC vs. SC core was 72.76% so it was not possible to reject a null-hypothesis that there was no significant statistical difference between SC and DC for core. Even less significant, the change in size between DC and SC had a P-value of 91.79%. Sign noted a larger simple change in value and running that against the chi-squared test shows that while not statistically significant change the P-value was much closer to significance, at 12.95%. This said, there was no statistical significance in the main three component categories high-level between DC and SC.

4.4.1 DC CORES

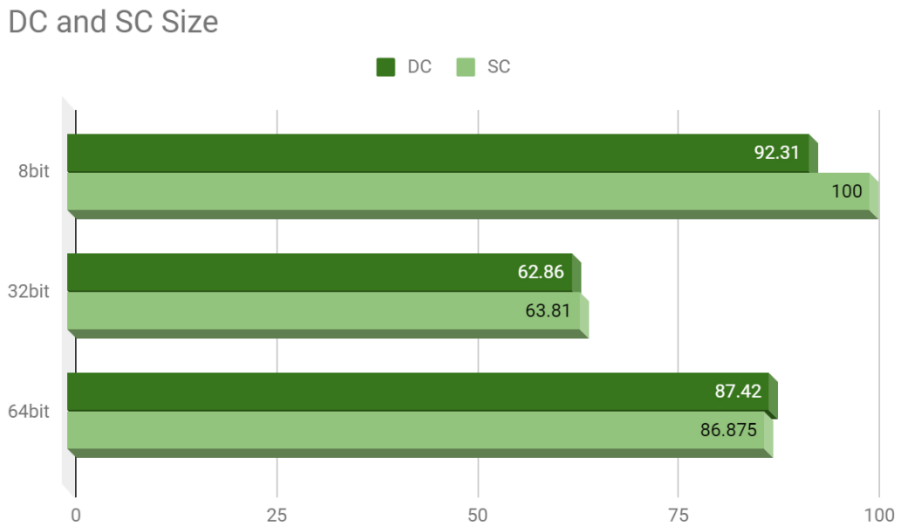
The value of all the cores decreases, float accuracy most noticeably, with a 14% loss. However, integers see about a 1% increase in accuracy. This tracks the change in integers from *SI* to *SC*. At this point, changes to the configuration would be done except for the need for consistent data; it seems the configuration requires additional indicators as increase in detection slows down.



Additionally, with the floats the drop in accuracy indicates that the configuration weights have a multilevel trigger-like functionality. The float core drop seemed significant, however, running the chi-squared test gave a 28.50% probability the change was chance, not enough to reject H0. H0 confirmed, states there was no a significant difference. These are float cores, however, something interesting happens when we try to combine the core with the appropriate size and sign, as will be discussed later.

4.4.2 DC SIZE

The category accuracy, like Core, only decreased by a percent. Previously, *SC* gave up a small percentage in category size, but decreased the accuracy of 32bit sizes by 15%. As seen below, this does not follow suit with large internal shifts. Though it is interesting to



note that an 8bit size was incorrectly typed after so long; 16b size remains unchanged so it is listed in a Table following the Figure instead.

The change in 32bit values were very unlikely to be significant with a chi squared test P-value of about 88.6%. Despite the decrease in 8bit value accuracy the decrease was not significant either as the P-value for the Fisher's exact two-tailed test was very high, and thus probable to be the result of chance.

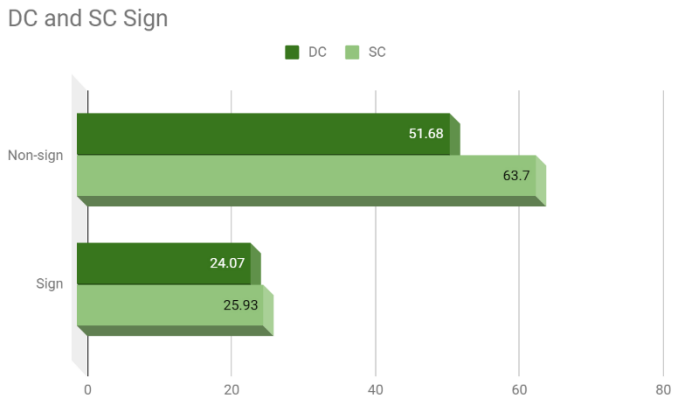
Table 12: DC Sizes

	8bit	16bit	32bit	64bit
Percentage (%)	92.31	100	62.86	87.42
Population:	13	7	105	159

It should be noted that 64bit sign accuracy increased a bit, these mild changes, highlighting the lack of significant Chi-squared and Fisher's tests. However, without additional indicators as mentioned with the drop in 32bit and slight increase in 64bit when going from SI to SC, additional information provided by DC may be less impactful.

4.4.3 DC SIGN

The non-sign value that appears alongside pointer core types dropped nearly 12%. The null-hypothesis is that there is no difference in using SC and to detect variables. The level remains set at 0.05. Applying the chi-squared test results with a 5.76% probability the

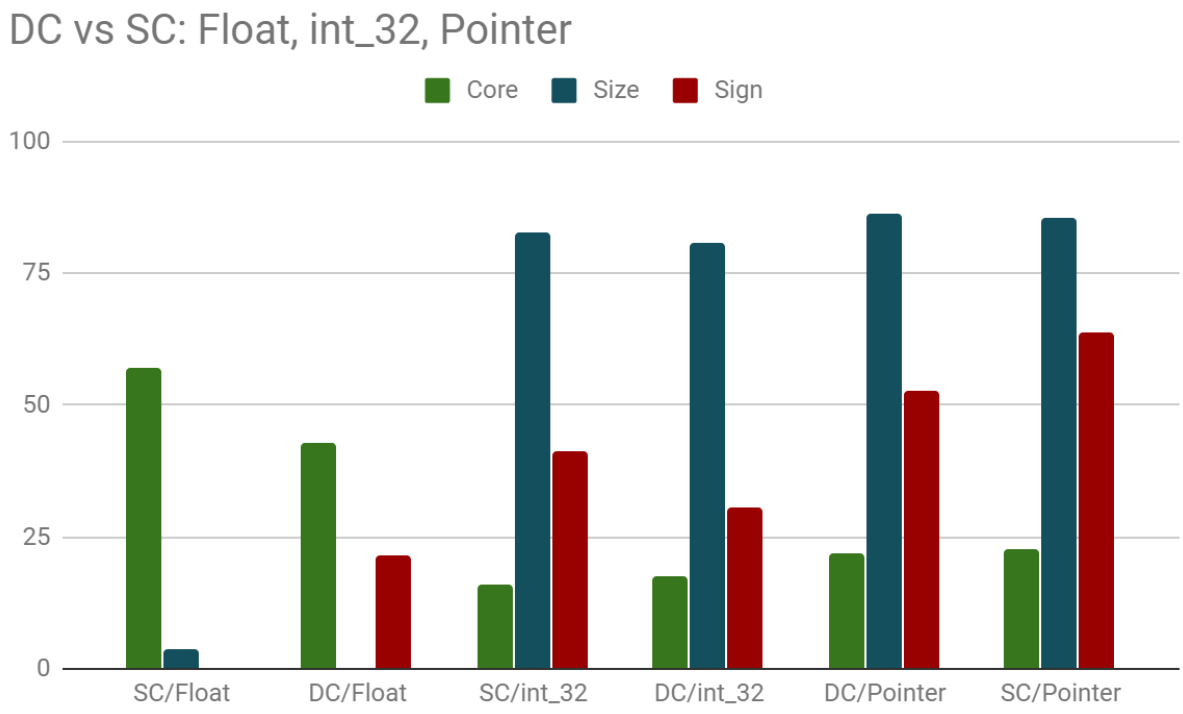


drop of accuracy when switching from SC to DC was due to chance. This was almost enough to reject the null-hypothesis and take the alternative that SC was statistically significant in better classifying Non-signs, however, it fell 0.76% short of the significance level (0.05).

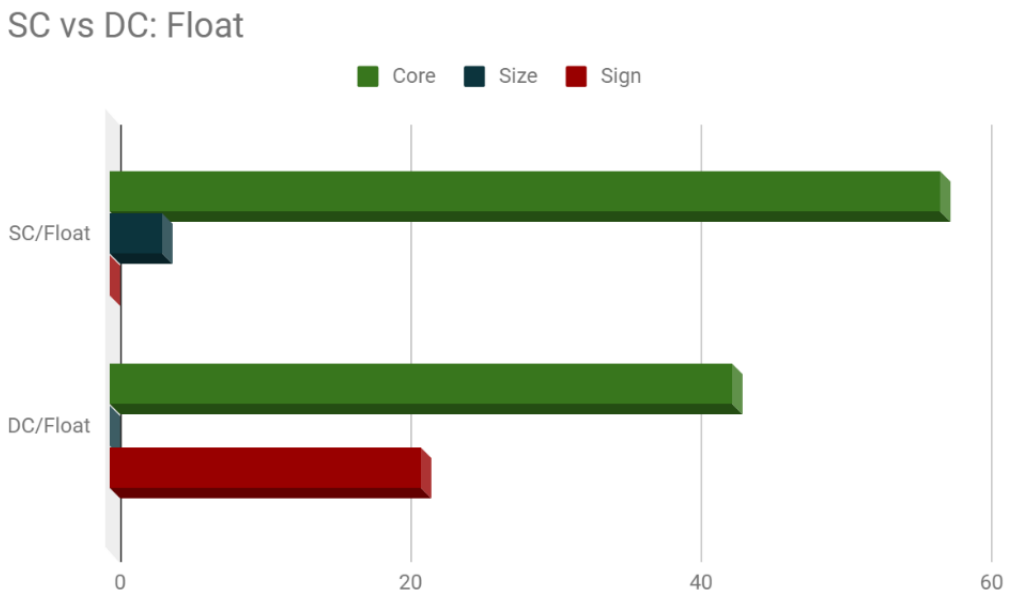
This indicates that either expanding the configuration or adjusting the weights might be helpful, though that would affect the value of these comparisons across the different methods, which is why there were no changes made in this instance.

4.4.4 DC FLOATS, POINTERS, 32BIT GLOBAL INTEGERS

With the trend of decreased values in DC it is interesting to see the differences between SC and DC, typing for int_32, pointer, and floats. Grouping the types of variables via the sublabels in the graph, a pattern emerges for most the variables except the floats.



As seen above, and a bit clearer in the Figure below the Float is a unique situation in that DC has a sign but no size.



The totals for the image above are calculated in the following table; DC, in a simplistic comparison, is a bit over 1% better at identifying floats than SC in this configuration. However, running this through Chi-squared test it is not significant.

Table 13: (DC) Floats, Pointers, and 32bit Integers

	Core (%)	Size (%)	Sign (%)	Total (%)
int_32	17.45	80.87	30.54	42.95
Pointer	21.92	86.3	52.74	53.65
Float	42.86	0	21.42	21.43

The overall detection of the float is the only improved item, the other two are lower by a few percent, though all changes were determined not to be significant. Pointers as a type overall, was closest to significant change with a 30% probability the change could be due to chance.

4.4.5 DC 8BIT AND 16BIT GLOBAL INTEGERS

The noted decline in unsigned accuracy affected the score here.

Table 14: DC Unsigned 8bit and 16bit Integers

	Core (%)	Size (%)	Sign (%)	Total (%)
uint_8	100	80	100	93
uint_16	100	100	100	100

While the DC uint_8 score decreased, the method maintained good detection rates even with all the additional information from the second context. The drops in accuracy were not statistically significant as covered earlier.

4.4.6 DC EFFECTIVENESS

The takeaway from DC lies in the tradeoff in significance. SC had many significant improvements because of the technique used. DC while accuracy decreased in many places, the decreases were not significant as the SC increases had been when switching from SI.

However, there were a few limitations discussed that will be covered more in Chapter 5 that could improve the DC effectiveness. Across all three methods it seemed that indicators could be increased

V. CONCLUSION

The problem has been noted in literature that evaluation and testing is difficult in type inferencing approaches due to various challenges like closed-source tools, commercial fees, inconstancy of data being tested as 85% of approaches noted in a survey from 2016 used begin binaries to test their program [7]. This survey noted the need for additional work to focus more on specific techniques in the hopes of generating better environments to test approaches in, or compare against, even if there is no access to the tool.

5.1 SUMMARY OF TECHNIQUES AND USE

This lightweight configurable approach evaluates the well-known techniques of flow-sensitive context-sensitive type inference based on instructions and type propagation. With the techniques isolated the various aspects of this approach and determines individual impact of the base techniques on the overall accuracy. Assessing the core, size, and sign of derived types independently as done in another approach [34], allows a detailed exploration of the pros and cons of each configuration. A ‘None’ value used when scoring the core, size, and sign is included as a means to assess when an indicator does not apply to a particular aspect of type inferencing. Indicators, scored values (weights), and sizes, among other information sources, are all configurable. If applied in lesser-used architectures, this approach could assist in learning valuable inference sources and testing techniques in different environments.

5.2 SUMMARY OF RESULTS

First, single instruction information tested type inference from an instruction in isolation. This revealed several significant relations between different pieces of data. Pointer cores poor accuracy rates were statistically significant in the relation to the good success of the Non-sign component (approximately 0% probability of achieving the same number by chance), and the poor rates of the signed component (0.3% probability of chance). Standing out from the poor overall typing abilities of single instruction inference, unsigned integers were detected with statistically significant success (~2% probability of achieving the same success rate by chance). Showing that certain types can be very accurately inferenced even with only a single instruction.

Then, a single context instruction was considered in addition to the primary instruction. This change greatly improved the inference of the core type and internally shifted the size and sign accuracy for certain types. This was all done with the overall size and sign accuracy only losing a couple percent. This is mostly due to the effect of the additional information, using only a limited number of indicators means if making a decision on a single instruction an indicator must be present otherwise no information can be provided. The internal redistribution of accuracy that was observed with an additional instruction occurred alongside a massive boost in pointer detection, up from 7% to 22% for this configuration using only a single indicator for pointer types. The overall difference in detection rates for core components was statistically significant; SC performed much better than SI in this regard, with only a 1.2% chance the results could have achieved this level of performance randomly, well below the level of significance at 5%. However,

core is the only overall component type to hold significance; sign and size failed to reject the typical H0 that there was no significance between the data in the test.

Despite the fact sign components as a category weren't statistically significant the internal reordering was very significant, so much signed, unsigned, and non-sign all had significant changes. The Non-signed values statistically significantly dropped a ton from SI to SC, while the signed (less than 0.1%) and unsigned (1.82%) both increased the accuracy when transitioning from SI to SC. Overall, 32bit signed integers almost made a significant increase in detection from SI to SC (5.76%), but was just a little too much for the 0.05 significance level required to reject the null-hypothesis. Pointers as a core improved the accuracy when switching from SI to SC, in a statistically significant amount (less than 0.01%).

Double context, a third instruction, was added to see what would result from the additional information. While initially it was believed more information would be better, the results generally decreased a little, though no decrease was statistically significant, which was interesting. Double context managed to improve the typing of the float's sign, which gave an overall boost in detecting floats over single context. However, the increase was not statistically significant. Sign component for double context came the closest to being statistically worse than the single context method, categorically Sign had a P-value of 13%, so there was 13% chance the decrease in Sign accuracy was random. Within Sign, Non-sign nearly managed to decrease with statistical significance with a P-value about 6%. Though the accuracy is limited by using so few indicators, the effect of the techniques are much more apparent these limitations and future work are discussed next in 5.3.

5.3 LIMITATION, SCOPE, AND FUTURE WORK

The approach presented here uses the labeling and disassembly capabilities from the Binary Ninja (Binja) reverse engineering platform. It would be desirable to shift towards more open source tools. Work has already been done in this direction, focusing on using gcc, and objdump for disassembly.

If a version of the tool were continued on for Binja, then there are other features that could be implemented, specially using the built-in graph support. The control-flow graph API from Binja supports dominator graph theory that could be used to increase the weight of instructions that are dominators for a block of code and improve the accuracy. A test function for this was designed with the API but wasn't implemented to better isolate the techniques evaluated. Future work could evaluate if there was additional benefit to it.

In either case alternative configurations with different indicators could be used. It should be made clear that the configuration has remained constant through all steps for best evaluation value of the techniques. In practice, extending the range of context should mean reevaluating weights and scoring to ensure full-advantage with the additional reach. Aside from the code itself, the dataset itself could be improved. Both in surveys and personal conversations, it is frequently mentioned that established databases with known types and verification methods would assist reverse engineers in evaluating type inference approaches. While the approach presented here verifies inferred types against source code it can be tricky to do completely statically. Future work, especially the effort already invested in moving the approach from its Binja platform to a free open source

platform, would take time to make datasets useful to other engineers for verification purposes.

REFERENCES

[1] Hex-rays, "IDA," [Online]. Available: https://www.hex-rays.com/products/ida/index.shtml. [Accessed 2019].

[2] Vector35, "Binary Ninja," Vector35, [Online]. Available: https://binary.ninja/. [Accessed March 2019].

[3] D. Mercier, "dynStruct: An automatic reverse engineering tool for structure recoery and memory use analysis," School of Computing, University of Kent, 2016.

[4] D. Bruening, T. Garnett, V. Kiriansky, R. Kleckner, Q. Zhao, B. Chandramohan and S. Larsen, "The DynamoRIO: Dynamic Instrumentation Tool Platform," [Online]. Available: http://dynamorio.org/. [Accessed 2019].

[5] A. Slowinska, T. Stancescu and H. Bos, "Howard: A Dynamic Excavator for Reverse Engineering Data Structures," *NDSS*, 2011.

[6] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries.," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016.

[7] J. Caballro and L. Zhigiang, "Type Inference on Executables," *ACM*, vol. 48, no. 4, 2016.

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, IEEE Computer Society, 2004.

[9] C. Jung and N. Clark, "DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage.," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2009.

[10] M. Zhang, A. Prakash, X. Li, Z. Liang and H. Yin, "Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis.," 2012.

[11] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis.," in *International Conference on Information Systems Security*, Springer, Berlin, Heidelberg, 2008.

[12] K. ElWazeer, K. Anand, A. Kotha, M. Smithson and R. Barua, "Scalable variable and data type detection in a binary rewriter.," *ACM SIGPLAN Notices,* vol. 48, no. 6, pp. 51-60, 2013.

[13] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables.," in *International conference on compiler construction*, Springer, Berlin, Heidelberg, 2004.

[14] M. Christodorescu, N. Kidd and W.-H. Goh, "String analysis for x86 binaries.," *ACM SIGSOFT software engineering notes,* vol. 31, no. 1, pp. 88-95, 2005.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation.," *Acm sigplan notices,* vol. 40, no. 6, pp. 190-200, 2005.

[16] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation.," *ACM Sigplan notices,* vol. 42, no. 6, pp. 89-100, 2007.

[17] I. Haller, A. Slowinska and H. Bos, "Mempick: High-level data structure detection in c/c++ binaries.," in *20th Working Conference on Reverse Engineering (WCRE),* IEEE, 2013.

[18] A. Cozzie, F. Stratton, H. Xue and S. T. King, "Digging for Data Structures," *OSDI*, vol. 8, pp. 255-266, 2008.

[19] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'hearn, T. Wies and H. Yang, "Shape analysis for composite data structures.," in *International Conference on Computer Aided Verification*, Springer, Berlin, Heidelberg, 2007.

[20] N. Hasabnis and R. Sekar, "Lifting assembly to intermediate representation: A novel approach leveraging compilers.," *ACM SIGARCH Computer Architecture News,* vol. 44, no. 2, pp. 311-324, 2016.

[21] A. Mycroft, "Type-based decompilation (or program reconstruction via type reconstruction).," in *European Symposium on Programming*, Springer, Berlin, Heidelberg, 1999.

[22] Z. Xu, C. Wen and S. Qin, "Learning types for binaries.," in *International Conference on Formal Engineering Methods*, Springer, Cham, 2017.

[23] Z. Lin, X. Zhang and D. Xu, "Automatic reverse engineering of data structures from binary execution.," in *Proceedings of the 11th Annual Information Security Symposium*, CERIAS-Purdue University, 2010.

[24] X. Wang, Y.-C. Jhi, S. Zhu and P. Liu, "Still: Exploit code detection via static taint and initialization analyses.," in *2008 Annual Computer Security Applications Conference (ACSAC)*, IEEE, 2008.

[25] J. Caballero, P. Poosankam, C. Kreibich and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering.," in *Proceedings of the 16th ACM conference on Computer and communications security*, ACM, 2009.

[26] J. Caballero, G. Grieco, M. Marron, Z. Lin and D. Urbina, "ARTISTE: Automatic generation of hybrid data structure signatures from binary code executions," IMDEA Software Institute, Tech. Rep. TR-IMDEA-SW-2012-001, 2012.

[27] S. Rus, L. Rauchwerger and J. Hoeflinger, "Hybrid analysis: static & dynamic memory reference analysis," *International Journal of Parallel Programming*, vol. 31, no. 4, pp. 251-283, 2003.

[28] X. Chen, A. Slowinska and H. Bos, "Who allocated my memory? Detecting custom memory allocators in C binaries," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, IEEE, 2013.

[29] J. Lee, T. Avgerinos and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," 2011.

[30] E. Robbins, J. M. Howe and A. King, "Theory propagation and rational-trees," in *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, ACM, 2013.

[31] Gartner, "Gartner Says Global Device Shipments Will Be Flat in 2019," Gartner, 2019. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2019-04-08-gartner-says-global-device-shipments-will-be-flat-in-. [Accessed 2019].

[32] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual," Intel, 2016.

[33] X. Yang, Y. Chen, E. Eide and J. Regehr, "Csmith," [Online]. Available: https://embed.cs.utah.edu/csmith/. [Accessed 2019].

[34] K. Dolgova and A. Chernov, "Automatic type reconstruction in disassembled c programs.," in *2008 15th Working Conference on Reverse Engineering*, IEEE, 2008.

[35] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[36] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385-394, 1976.

[37] C. P. G. S. K. C. S. P. Cadar, K. Sen, N. Tillmann and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *2011 33rd International Conference on Software Engineering (ICSE)*, IEEE, 2011.

[38] K. Sen, D. Marinov and G. Agha, "CUTE: a concolic unit testing engine for C.," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263-272, 2005.

[39] P. Godefroid, N. Klarlund and K. Sen, "DART: directed automated random testing," *ACM Sigplan Notices*, vol. 40, no. 6, pp. 213-223, 2005.

[40] S. Debray, R. Muth and M. Weippert, "Alias analysis of executable code," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, 1998.

APPENDIX A: ANALYSIS TYPES

Reverse engineering has two core analysis approaches outline above, dynamic and static. However, there are other types that stem from these such as concolic, concrete, and symbolic analysis, as well as techniques that are applied to a base analysis approach. Analysis approaches have specific advantages and disadvantages. Sometimes, depending on the specific application of a tool, a disadvantage in an analysis approach is not relevant. Thus types of analysis listed in this section are considered with respect to the impact on type inferencing primitive types.

DYNAMIC ANALYSIS

Dynamic analysis (similar to concolic execution in that concrete input values are used) is done with the use of input values and gathering information about a binary as it executes. However, this means the information gathered is restricted to the control flow path taken by the current inputs of the run. So multiple runs with varying inputs must be executed to explore the program, a consideration labeled code coverage. By definition, to successfully use dynamic analysis the binary being analyzed must execute, which raises an issue. There are times when executing a binary is not appropriate, such as environments that cannot run the binary, binaries that are uninteresting due to significant reliance on missing libraries, or it is a partial binary. While some issues can be addressed, like using QEMU to emulate an environment to enable execution [35], incomplete or non-executable binaries require a different approach.

STATIC ANALYSIS

Static analysis is done by gathering information about binary without executing it. This means analyzing either the assembly instructions or an intermediate representation (IR) if the binary was lifted. An advantage of static analysis is that all the code of the binary is visible; unlike in a dynamic approach, there is no need for using multiple inputs runs to explore branches or generally get better code coverage. Additionally, this means that even partial binaries can be analyzed. Unfortunately, static analysis is far more susceptible to obfuscation than dynamic approaches, the cost of having the benefit of full code visibility. While this paper does not explore the effects of obfuscation in detail, it is an important drawback to mention when considering static analysis of any binary, complete or partial.

SYMBOLIC, CONCOLIC, AND VALUE-SET ANALYSIS

Symbolic execution can be considered a type of static analysis; it uses symbols in place of inputs and generates an expression as the symbols propagate through the binary [36]. The analysis keeps a 'path condition' on branches that creates a set of constraints. This feature is often applied in test generation tools. A weakness in symbolic analysis is the scalability of constraint generation; for large programs, there are many paths and branches to be analyzed generating several constraints [37].

Concolic execution is hybrid of symbolic (static) and concrete (dynamic) execution; defined as *co*operative concrete and symbolic execution [38]. DART was an approach to software testing that used this by running a program with concrete inputs and running it symbolically to generate constraints for the concrete inputs at certain points in the program [39].

Value-Set analysis is a static analysis algorithm that is used to identify and track usage of memory addresses and offsets in addition to hardware registers. It maintains a list of addresses a data object could have at a given point in the program and can use this to provide lists of used, killed, and possibly-killed sets for a given program instruction [13].

SINGLE STATIC ASSIGNMENT

Single static assignment (SSA) is an IR technique that can allow static approaches to resolve the problem of unification. SSA works to deconflict multiple writes to the same location by creating a new label on each assignment, so each location is only assigned to once. When code branches and a location is updated in both branches if the branches later rejoin, a new assignment is made denoting the resulting variable could have been assigned to by either branch, thus even with branching each location is only assigned to once. Locations are commonly register locations, but TIE also used this on memory locations in addition to registers to improve the typing results [29].

ALIASING ANALYSIS

Aliasing is a technique that works to track registers and determine a set of values the registers could contain at a point in the program [40]. VSA is further progress in aliasing that has a valuable difference in that it tracks the set of possible values for memory locations, in addition to registers [13] .