November 2017

Strong-DISM: A First Attempt to a Dynamically Typed Assembly Language (D-TAL)

Ivory Hernandez
University of South Florida, binaryworldnexus@gmail.com

Strong-DISM: A First Attempt to a Dynamically Typed Assembly Language (D-TAL)

by

Ivory Hernandez

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Jarred Ligatti, Ph.D.
Yao Liu, Ph.D.
Xinming (Simon) Ou, Ph.D.

Date of Approval:
October 13, 2017

Keywords: Computer Security, Proof-Carrying Code, Typing Theory,
Static and Dynamic Languages, Compilers and Interpreters

DEDICATION

Many are the people I have to be graceful for this achievement. Although all of them contributed to me in more than one way, here are very narrow highlights of their contributions. My grandpa, on my mother's side, always made me curious of things and told me that knowledge does not occupy space and could be always taken anywhere. My grandma, on the mother's side, always taught me respect, decency, and organization. My grandma, on the father's side, taught me respect, humility, and toughness. My mom set the bar high as I always saw her solving problems and reading her engineering books; she made us both, my brother and me, aware of the necessity of higher education and sent us, every Saturday (sometimes forcibly), to the public library. My dad who got his high school diploma as an adult and always kept trying to learn new things, told us to study since the early moments of school. My fifth grade teacher, Xiomara, who taught me not to be sloppy and showed me I could be an achiever. All my beloved art school teachers who led me as a teenager to become into a mature individual and showed me a hidden and marvelous way to see the world. All my friends who have been all along helping me and demonstrating me that I am not alone. My younger brother for being always there by my side. My dear mother-in-law, Mamota, and my second mother Luisita for always wishing me the best. My life companion Lileana for being far above and far beyond golden. All of them tied by common denominators: love and respect.

ACKNOWLEDGMENTS

Especial thanks to my professor, Dr. Ligatti, for his patience and guidance all along the development of this thesis; and to my friend Mario Lopez, M.C.S., who provided me with beneficial comments on the final paper.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Dynamically Typed Assembly Language (D-TAL) is not only a lightweight and effective solution to the gap generated by the drop in security produced by the translation of high-level language instructions to low-level language instructions, but it considerably eases up the burden generated by the level of complexity required to implement typed assembly languages statically. Although there are tradeoffs between the static and dynamic approaches, focusing on a dynamic approach leads to simpler, easier to reason about, and more feasible ways to understand deployment of types over monomorphically-typed or untyped intermediate languages. On this occasion, DISM, a simple but powerful and mature untyped assembly language, is extended by the addition of type annotations (on memory and registers) to produce an instance of D-TAL. *Strong*-DISM, the resulting language, statically, lends itself to simpler analysis about type access and security as the correlation between datatypes and instructions with their respective memory and registers becomes simpler to observe; while dynamically, it disallows operations and further eliminates conditions that from high level languages could be used to violate/circumvent security.

CHAPTER 1: INTRODUCTION

1.1 Introduction

The proper implementation of a sound computing system requires a clear understanding of what security properties must be preserved by all intervening computing languages across all levels of computation.

Typing, for high level languages, is commonly used as the preferred technique to establish and preserve soundness [1,12,15,16,17,18], as types and typing rules can be mathematically abstracted, calculated, and formally proven. This is the main reason why typing mechanisms are seen as powerful security tools and have become the standard safety validator for most high-level languages.

The addition of types can also be used in low level languages to prove soundness [2,18]. However, as the application of types has not been generalized to existing low-level models of computation, namely the Von Neumann and Harvard models [72,73,74,75,76,77], it is possible to ***improperly eliminate types*** during the translation process of programs from high to low-level languages. This elimination of types, as a consequence, creates an important gap in security for computations executed at low levels. Thus, if safety properties throughout the different layers of computation are not preserved and enforced, characteristics and properties that rendered a high level language as sound are by consequence lost, leaving the system vulnerable to inappropriate accesses to memory (i.e. buffer-overflows, format string attacks, integer overflows, etc.) [19]. As a matter of fact, many of the known exploits that take effect at low levels directly relate to

improperly translated high level programs allowed to execute at low levels. During the translation, some of the programs' security properties are inadvertently eliminated, and the programs, assumed to be sound, are erroneously granted a trusted status when they are actually unsound and vulnerable.

1.1.1 The Static Solution

Several ideas to cope with this rift in security have been proposed. One of them, Proof Carrying Code (PCC) [33,34], is based on the general idea of a mechanism that allows an untrusted program to access system's memory after the program has provided proof that it conforms to the system's security requirements. Another proposition, descendant of PCC, is Typed Assembly Language (TAL) [2,8,13,14]. TAL is considered as an instance of PCC, and it proposes the use of the properties of types as a security mechanism to prove the soundness of a program's execution. TAL requires that types are generalized over low level untyped intermediate languages, and once this generalization of types takes over, the safety properties of high level languages programs can be preserved during translation to equivalent lower level programs [24,25]. Successful implementations of TAL and PCC produced by researchers have been able to provide proofs that confirm it is possible to close the gap in security.

The use of a static typing discipline for TAL implementations has been prevalent [2,8,13,14,16,18], and the resulting implementations, especially the initial ones, have been inherently convoluted. The reason behind this level of complexity has been that in order to derive formal proofs to demonstrate the correctness of programs after their transformation from a high level source language to a target typed assembly language, there has been needed to perform static analyses to deduce if the desired properties were still present after the transformation; hence, requiring the use of complex compilers, formal systems, proof checkers [3,4,5,6,7,23], and, in general, an extraordinary level of theoretical sophistication and expertise.

1.1.2 The Dynamic Option

In contrast, the use of dynamic types for TAL implementations, which did not seem to have received the same attention by the research community, shows a great potential to alleviate much of the burden associated with the static approach to typing. This thesis presents, to the best of our knowledge, what it is believed to be the first implementation of a dynamically typed assembly language, especially one where types are preserved throughout all the relevant computational stages. Also, this thesis will not be concerned nor will include any compilation proof or analysis coming from a high level typed source language, but will simply focus on showing the different aspects required to achieve a dynamic TAL implementation.

1.1.3 Thesis

The exposition of the detailed steps of implementation and particulars on how to convert DISM [92], an untyped language, into *Strong*-DISM [93,94], a dynamically type checked assembly language, will be used to uncover beneficial aspects of a dynamic approach not previously investigated and to help realize why *a dynamic typing discipline has important advantages over its static counterpart as it simplifies and makes visible important aspects concerning the implementation of typed assembly languages*.

1.2 Motivation

The evaluation of the feasibility of a dynamically checked TAL, the lack of reliability of static typing for .NET and Java virtual machines [96], the need of a clearly understanding of the mechanics behind a dynamically typed low level system, and curiosity are the main motivations for this thesis. At a time, there was knowledge of the existence of typed assembly languages, but awareness grew as more details were gathered after previous TAL research was reviewed. This research in conjunction with attempting a D-TAL implementation, surprisingly revealed that the

approach taken on this thesis led to a simpler and sound alternative to previous statically typed implementations and clarified ways to extend type safety to other untyped assembly instances.

1.3 Approach

As the mechanisms used to implement D-TAL are crucial to show the benefits of a dynamic typing approach, a well-studied technique that surely prevents inappropriate access memory attacks by extending type safety to lower layers of computation has been selected for implementation.

1.4 Contributions

The enunciation of a clear and simple methodology of how to implement dynamically type checked assembly languages and, to the best of our knowledge, the first ever produced dynamically typed assembly language implementation (with type checking inclusive of all memory) are claimed as contributions of this thesis.

1.5 Overview of Related Work

Previous work and research considered as directly related to this thesis are Proof-Carrying Code [33,34] and TAL [2,8,13,14,19,22,23,24,25,26,27] researches, as well as research made on gradual typing [29,68] and dynamic typing [66,67,69] areas. PCC proposes a way to prove correctness on untrusted code by using different certification mechanisms, while TAL, an instance of PCC, proposes the use the properties of types to establish safety. Many foundational aspects of this thesis directly relate to theoretical concepts and practical mechanisms of gradual and dynamic typing.

1.6 Static and Dynamic Typing Tradeoffs

Static and dynamic typing disciplines complement each other [10,11]. However, some of their characteristics make them suitable (or desired) for specific types of implementation.

Desirable characteristics of statically type checking:

- All possible paths of execution are type checked.
- Earlier error checking.
- Lends itself to more structured programming styles.
- Object code can be generated more efficiently.
- Programs can handle a higher level of robustness.
- Finer-grained transformations from high level languages can be achieved.
- Compiled code is faster to execute
- Type checkers will catch type safety violations before code is executed.

Undesirable characteristics of statically type checking:

- Valid programs can be ruled-out from executing.
- Language becomes less flexible with respect to the evaluation of expressions.
- Formal methods and theorem provers may be needed to verify programs.

Other undesirable characteristics directly observed and/or inferred during research:

- The level of knowledge required to reason about compilation's correctness can be cumbersome, imposing in many cases a high intellectual toll.
- Code generated tends to be complex and bloated.
- Type-preserving compilers, formal systems and proof-checking devices are needed to show code correctness in some instances.
- If an error is made in one of the components, the error could be hard to locate, and it could introduce security holes in the target system if unnoticed.
- Computational soundness can be lost due to physical hardware fluctuations.

Desirable characteristics found on dynamically type checking:

- Abstract Data Types (ADTs) can be represented independently and modularly.

- The resultant typed assembly language is directly programmable -as no compiler is required.
- The resultant typed assembly language can be used as a target language by any compiler.
- The language constructs and definitions are highly adaptable and flexible at design time.
- The code produced is less bloated.
- The abstraction model produced is cleaner and therefore simpler to analyze.
- Implementation can be done straightforwardly.
- Proving soundness is less cumbersome.

Undesirable characteristics found on dynamically type checking:

- As type-checking occurs at runtime, code will execute slower than its static counterpart.
- Only the current path of execution is type-checked.
- An interpreter is required.

1.7 Thesis Roadmap

The following chapters will unwind as follows:

- Chapter II will be dedicated to Related Work.
- Chapter III will present DISM, the initial untyped language selected for this study, and will include its syntactic rules and operational semantics, followed by implementation details and code example.
- Chapter IV will present *Strong*-DISM. This is the resulting language after DISM has been type-extended. Furthermore, its syntactic rules and operational semantics will be included, followed by its implementation details and code example.

- Chapter V will present an analysis of the added rules with respect to language soundness, an example of exploits eliminated by the use of types, and the benchmarks from running similar code using DISM and *Strong*-DISM instructions.
- Chapter VI will present a general exposition of how the type-extension technique described in chapter IV could be effectively applied to ARM assembly language or to any other untyped assembly language.
- Chapter VII will present the conclusions of this thesis with summary, recommendations, and possible paths of future work.

CHAPTER 2: RELATED WORK

2.1 Proof-Carrying Code

PCC, proposed by George C. Necula and Peter Lee [33,34], is based on encoding a safety proof which contains the specification of the formal operational semantics of a native machine code language along with a set of rules to prove safety for all machine instructions.

This approach allows to publish the safety rules requirements that any external code must conform to if it wished to be granted insertion and execution rights by the host code [33,34]. If the external code is certified and validated, it must be only because the external code is in full compliance with the previously published host code rules, and this external code can then be trusted and allowed to execute by the host [33,34].

As type safety is one of the instances by which PCC certifies external code, D-TAL becomes an instance of PCC. However, the ways that PCC can certify code, go beyond types, and can be achieved by describing a meta-logic based certificate architecture [31].

2.2 Typed Assembly Language

Typed assembly language, a direct instance of PCC, uses types and type derivations as proofs of correctness [22]. Seminal work from Morrisett's '95 thesis dissertation [2] paved the way of further TAL implementations by providing specifications on a series of techniques and formally proved that it was possible to statically map types from high level typed languages containing elements such as abstract datatypes (ADTs), objects, modules, first-class polymorphism, subtyping, etc. [2] to low level monomorphic languages. Such techniques were

called type-directed compilation and dynamic type dispatch. Type-directed compilation [27] referred to static typing, and dynamic type dispatch to a way of delaying a type substitution until the "right" moment. The proofs presented by Morrisett's thesis were extensive and covered much of the translation of correctness at every step of compilation –done by intermediate compilers- from a high level ML-like language to a low level typed language, with their languages prototypes and compilers implementations included. It is important to mention that the dynamic type dispatch technique required the use of dynamic typing. In general, most types that cannot be determined statically at compile time are left to be type checked at runtime; therefore, some dynamic typing is needed sometimes to extend static types [2].

Further research produced TAL instances such as MTAL, that took steps to TAL's consolidation by demonstrating correctness over object linking for a typed assembly language [8]; Stack-based Typed Assembly Language (STAL) [13], an extension of TAL with stack constructs and stack types to support modern architectures with stack allocation; and TALx86 [22], which included the Popcorn compiler as part of the TALx86 Tools and proved over a real system that it is possible to establish correctness over code transformations from typed assembly language to machine code while still preserving security properties such as memory safety [2].

As more TAL implementations and research papers [2,8,13,14,18,19,20,22,23,24] appeared over the years, solid theoretical and practical foundation made it to newer implementations [17,20], and even to an experimental operating system [16]. D-TAL directly relates to and draws on much of the ideas and contributions made by TAL [15,16,19].

2.3 Dynamic Typing

Dynamic typing is at the core of this thesis. Dynamic typing systems and the properties of dynamic type checking have been thoroughly studied [10,66,67]. A remarkable paper by Fritz

Henglein, published in 1994 [69], describes a dynamically typed λ-calculus to which our D-TAL implementation, *Strong*-DISM, shares a close resemblance with [93].

In that paper Henglein talks about properties common to most dynamically typed languages such as runtime tagged values with their associated tagging and check-and-untag operations. He further describes the elements of type Dyn as runtime "(type) tagged" values or tag-value pairs where the tag indicates the type constructor or primitive type of the value component. Here is a description of the rules for dynamic type checking, as described by Henglein, that directly applies to this thesis:

> "For every type constructor tc of arity k there is a tagging operation tc! That maps elements of type tc(Dyn, . . . , Dyn) to Dyn by pairing them with their type…For every tagging operation tc! there is a corresponding check-and-untag operation tc? that maps elements of type Dyn to tc(Dyn, . . . , Dyn): it checks whether its argument has the tag tc; if so, it strips the tag and returns the untagged value; if not, it generates a (run-time) type error." [69]

Henglein also, amongst other proofs, produced proofs about safety when rewriting system properties, and minimally rewriting a system as well [69].

Other papers of relevancy to this thesis [65,66,67], with respect to the use of dynamic types, included further analyses of the topic over higher level languages with the inclusion of a formal calculi based on operational and denotational semantics.

2.4 Gradual Typing

The study of how the safety properties are preserved during the coalescing of static and dynamic typing disciplines has evolved into its own field, and it has been termed as gradual typing. Gradual typing is highly concerned with the preservation of a program semantic throughout its execution.

As many gradual typing systems rely on runtime type checking, a common problem to be solved by gradual typing systems is how soundness can be lost by the omission or improper type annotations. Most target languages that are dynamically typed are internally sound as their rules will not produce improper configurations (i.e.: stuck configuration) or undefined behavior (i.e.: memory corruption); however, in a gradual typing system, the gradual translations –when improperly done- may lead to a point in the translation in which values of the wrong type may inhabit variables with an already statically defined type, and this may further lead to unexpected hidden errors that are hard to debug. [65,68].

The goal of gradual typing is to enable the safe interaction of statically and dynamically typed code [29,66,67]. A gradual type system allows to define, before a program's compilation or execution, which portions of such programs will be statically or dynamically typed and how type soundness will be preserved at each stage.

A criteria of interest intrinsically relevant to this thesis that has been formalized [29,30,68,91] by gradual typing researchers are the concepts of *open-world soundness* and *gradual guarantee*. *Open-world soundness* states that a program that is well-typed, when translated from a gradually-typed language into an untyped target language, may interoperate arbitrarily with existing code at the untyped level without producing new –uncaught- type errors [29]. The premise of *gradual guarantee* asserts that no new errors are introduced by the weakening or removal of type annotations [91]. Both concepts, *open-world soundness* and *gradual guarantee*, were kept in observance all along the development of this thesis, especially during its implementation phase.

CHAPTER 3: DISM

This section will introduce and explain with details the characteristics of the base language selected for the language transformation studied in this thesis, from an untyped assembly language to a dynamically typed one, and will also explain why was the DISM implementation selected.

3.1 Why DISM

DISM offers the conceptual framework with the ideal characteristics required to analyze and visualize how an untyped assembly language is transformed into a typed assembly language, as DISM is a software-based interpreter [21], extensible, and designed to allow rapid implementation and prototyping.

At first, some ARM development kits were considered, but they were found not to be flexible enough to perform the required customizations nor simple enough to provide a clear angle of observation for key concepts that, otherwise, would have been buried under complexity. Hence, DISM became the preferred tool for virtualization and rapid prototyping that allowed the insertion of experimental, abstract, and dynamic concepts into observable implementation.

As added values, DISM is not only straightforward to use, but it is elegant, it is mature, and it is powerful. It also provides analysis tools that allow the live step-by-step detailed debugging of DISM programs. Indeed, DISM elicits the right level of abstraction to streamline how memory is utilized. Maybe, the best value found on DISM is its ability to highlight relevant aspects of computation.

3.2 DISM General Overview

DISM stands for Diminished Instruction Set Machine [92] and includes the assembly language of a virtual machine that emulates a RISC processor [36,37]. As its instructions are based on standard RISC instructions [36,37], they are very alike. DISM instructions semantically mean operations on registers, memory –via addresses- or constant values, and because the operands for DISM instructions are all representing natural numbers, DISM is considered to be untyped. For this thesis, DISM is considered the base language from where our conversion takes place.

3.3 DISM Data Structures

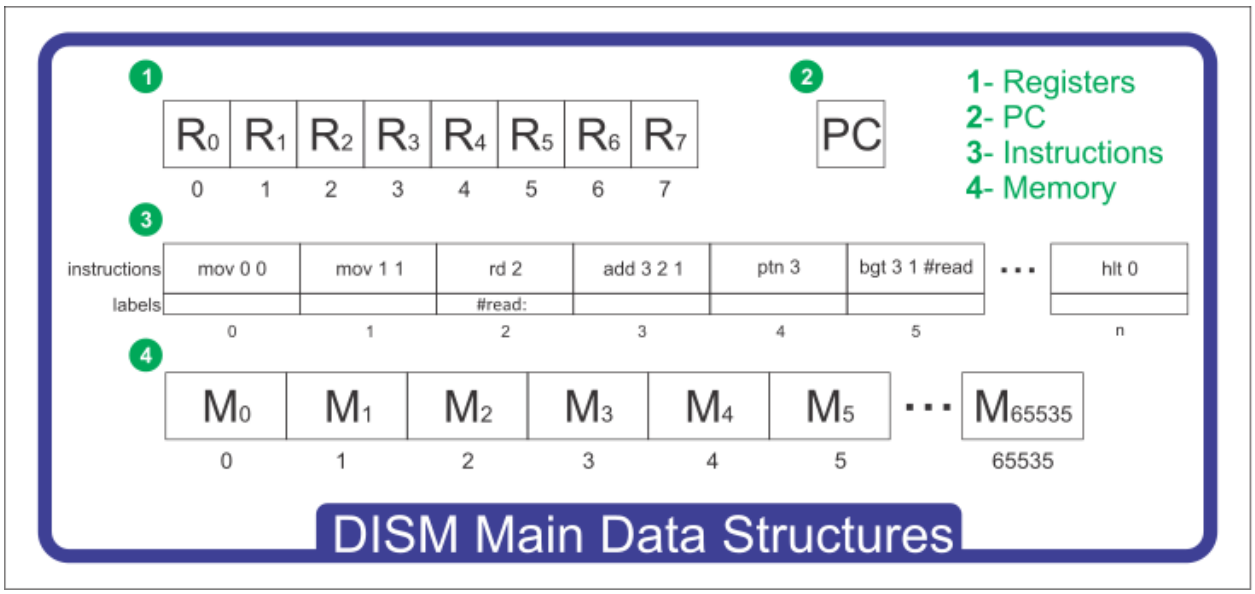

Figure 3.1 DISM logical data structures.

DISM has eight general purpose registers (see figure 3.1), a data memory array, a program counter (PC), and an implicit instruction's array. The registers are represented by an integer array, indexed from 0 to 7. The data memory is represented by an array with space available for 65536 unsigned integer values and indexed from 0 to 65535. The program counter represents a register, and it is implemented by a variable that holds the index value for the next instruction to execute. There is also an implicit (and transparent to the programmer) integer array

that contains the program's instructions and is created dynamically by a function that traverses the Abstract Syntax Tree (AST) of the parsed DISM program being executed. This transparency allows for DISM instructions not to be modified and only executed.

3.4 DISM Instructions

DISM has only twelve atomic instructions, and they contribute to the overall computation by producing side-effects over given memory types. DISM instructions are designed to operate on any of the 8 general purpose registers and can only interact with the data memory to perform data reads and writes via the *lod* and *str* instructions respectively. Figure 3.2, extracted from the version 0.5 of the *"Definition of DISM"* [92], contains all DISM instructions and their definitions.

Instruction	Meaning
`add d s1 s2`	`R[d] <- R[s1] + R[s2]`
`sub d s1 s2`	`R[d] <- R[s1] - R[s2] (R[d]<-0 when R[s2]>R[s1])`
`mul d s1 s2`	`R[d] <- R[s1] * R[s2]`
`mov d n`	`R[d] <- n`
`lod d s i`	`R[d] <- M[R[s]+i]`
`str d i s`	`M[R[d]+i] <- R[s]`
`jmp s i`	`PC <- R[s] + i`
`beq s1 s2 n`	`If R[s1] = R[s2] then PC <- n`
`bgt s1 s2 n`	`If R[s1] > R[s2] then PC <- n`
`rdn d`	`Read natural number from screen into R[d]`
`ptn s`	`Print natural number R[s] to screen`
`hlt s`	`Halt the DISM with code R[s]`

Figure 3.2 DISM instructions, opcodes and definitions.

Extending on the DISM instructions, opcodes and definitions provided in figure 3.2, *n* stands for a natural number, *i* for an integer, *s* for a source memory –the memory or register from where a value is read-, and *d* for a destination memory –the memory or register to where a value is written. The *lod* and *str* instructions automatically check that the programs can only access data within the DISM's allocated data memory. The *beq, bgt* and *jmp* instructions have in common that they can write to the program counter (PC) and, therefore, set value of the next instruction to be executed. The *beq* and *bgt* instructions are comparison-based, and the *jmp*

instruction uses pointer arithmetic to find a given instruction. In DISM, writes to the PC are guaranteed to be values to instructions that are part of the program, or otherwise, an exception terminates the program's execution; this behavior is transparently enforced by DISM. Also, opcodes -or tokens used to denote instructions- are restricted to be in lowercase, and all DISM programs must successfully be ended with the *hlt* instruction. This rule is with the purpose of disallowing a DISM program from executing nonexistent instructions.

DISM provides for the use of comments and symbolic labels. DISM considers comments any combination of characters after the ASCII semicolon symbol. To declare a symbolic label, the '#' symbol must be followed by an ASCII sequence of characters, in any combination, that includes the set of upper and lowercase letters from the Latin alphabet and the digits from '0' to '9' ended with the ASCII colon symbol.

Declarations of symbolic labels must always happen before an instruction. After a symbolic label has been declared, it can be utilized as a reference to an instruction, and consequently, can be used to replace the value to be written as argument to set the PC. An example that illustrates the use of symbolic labels can be found on section 3.6.

3.5 DISM Interpretation

Before execution, a DISM program is converted to an AST object, and all checks are performed dynamically by the DISM interpreter on this object. During the initialization of the virtual machine, all the values on the registers and memory array are initialized with the integer value '0'. The PC gets initialized with integer value '0' as well, referencing the location of first instruction on the instruction's array.

At execution time, the PC is incremented by 1 every time an instruction is executed –with the occasional exception of instructions that write to the PC- thus the PC is always referencing to the next program instruction on the instruction's array. Additionally, the instruction being

executed and the contents of registers and memory right after the instruction's execution can be output to the screen if debug mode is activated.

3.6 DISM Security Considerations

Although DISM provides beneficial runtime checks, e.g.: making sure programs can only execute instructions that are part of the program, DISM is untyped and uses a homogeneous positive (including zero) integer-based datatype for all its values. This single type makes possible that the values inhabiting a memory resource (i.e.: PC, registers, memory) can be effectively used to inhabit as values of any other memory resource, which makes possible the usage of this characteristic to create untyped-based exploits. Further, because of its single type encoding design, DISM does not have to check types. Any program that encoded types to be run on DISM would have to consider DISM mechanics in order to guarantee a safe execution.

3.7 A DISM Program Example

The following code corresponds to two equivalent DISM programs [25]. They illustrate the use of symbolic labels.

Program 1 uses symbolic labels.

```
            rdn 1 ;read n into register 1
            rdn 2 ;read m into register 2
            mov 3 1 ;move value 1 into register 3
#LOOP:      beq 2 0 #END ;if m==0 then goto end
            ptn 1 ;print n
            sub 2 2 3 ;decrement m
            jmp 0 #LOOP ;goto loop beginning
#END:       hlt 0 ;halt with code 0
```

Program 2 does not use symbolic labels.

```
            rdn 1 ;read n into register 1
            rdn 2 ;read m into register 2
            mov 3 1 ;move value 1 into register 3
            beq 2 0 7 ;if m==0 then goto end
            ptn 1 ;print n
            sub 2 2 3 ;decrement m
            jmp 0 3 ;goto loop beginning
            hlt 0 ;halt with code 0
```

CHAPTER 4: *Strong*-DISM, A D-TAL INSTANCE

Strong-DISM is the result of a series of straightforward enhancements to DISM. It encodes more than one type by making its instructions aware of types and adds dynamic type checking. The results of these enhancements produced an instance of TAL, and beyond, an instance of D-TAL.

As types were implicitly added to the instructions operands and yielding values, type checking enforced that registers and memory always contained the proper type of value. In order to support these implicit types and dynamic type checking, it was needed to add new instructions, reclassify existing instructions, extend the underlying support mechanism, and restructure the existing data structures. Once the conversion from DISM to a D-TAL had been completed, all the benefits of types were extended to any *Strong*-DISM program directly created by a programmer or to any higher-level program translated by a compiler into *Strong*-DISM. The resulting language contains the necessary elements to be able to cope with proof carrying by means of type checking [15].

As the scope of this thesis just focused on providing the minimal required functionality to support a safe program translation from a higher level language and focused on the analysis of a *Strong*-DISM program's runtime conditions, all the burden of compilations and the submission of a proof to show that it is possible a sound and gradual translation from a higher level language was left out as this type of proofs have been covered by previous TAL research [9,10,15]. Also, to be completely fair, it must be mentioned that most of the heavy lifting and initial work had been already done on DISM; thus, the work of modifying the base language became simpler.

Note how this approach, as it builds on existing work, allows the saving of considerable amounts of time and effort, when converting existing untyped assembly languages to typed assembly languages, especially if the base language is an already mature implementation, as in the case of DISM.

4.1 *Strong*-DISM General Overview

Strong-DISM is directly derived from DISM, and, as DISM, is the assembly language of a virtual machine that emulates a RISC processor. *Strong*-DISM instructions are based also on standard RISC instructions [39,40], and are semantically connected to operations on registers, memory –via addresses– or constant values. However, such operations are not performed on (or yield) a single type. Instead, *Strong*-DISM's instructions can be performed on, yield, and are represented by two types: ***nat*** (natural numbers) and ***inst*** (references to code instructions) types; hence, *Strong*-DISM can be considered a typed assembly language, or to be more precise, a dynamically typed assembly language.

4.1.1 *Strong*-SIM and *Strong_Mem*-SIM

To better understand how effective abstract concepts behaved when applied to a real model, two implementations of *Strong*-DISM were produced [93][94]. The name of the executable for the first implementation was *strong-sim-dism*, and it contained the abstract idea of keeping track of types in data memory by physically segregating memory. The second implementation's executable was called *strong_mem-sim-dism*. This second implementation did not separate physical memory by types, but placed all type-flagged values next to each other in a continuous array. Following on, *strong-sim-dism* implementation could be referred to as *Strong*-SIM, while *strong_mem-sim-dism* could be referred to as *Strong_Mem*-SIM.

4.2 *Strong*-DISM Data Structures

Both, *Strong_Mem*-SIM and *Strong*-SIM (see figures 4.1 and 4.2), inherited from DISM the general purpose registers (numbered from 0 to 7), the program-counter register, the implicit and transparent-to-the-programmer program instruction's array, and the data memory. However, there were substantial differences for both implementations of *Strong*-DISM.

4.2.1 Data Structures Implementation for *Strong*-SIM

For the *Strong*-SIM [93], the registers array inherited from DISM was left untouched, and register's support to keep track of ***nat*** and ***inst*** types was made possible by the addition of an extra array of eight elements. The elements of this array were indexed after the register's array, and they keep track of the types of the values in registers inhabiting a similar index (see figure 4.1).

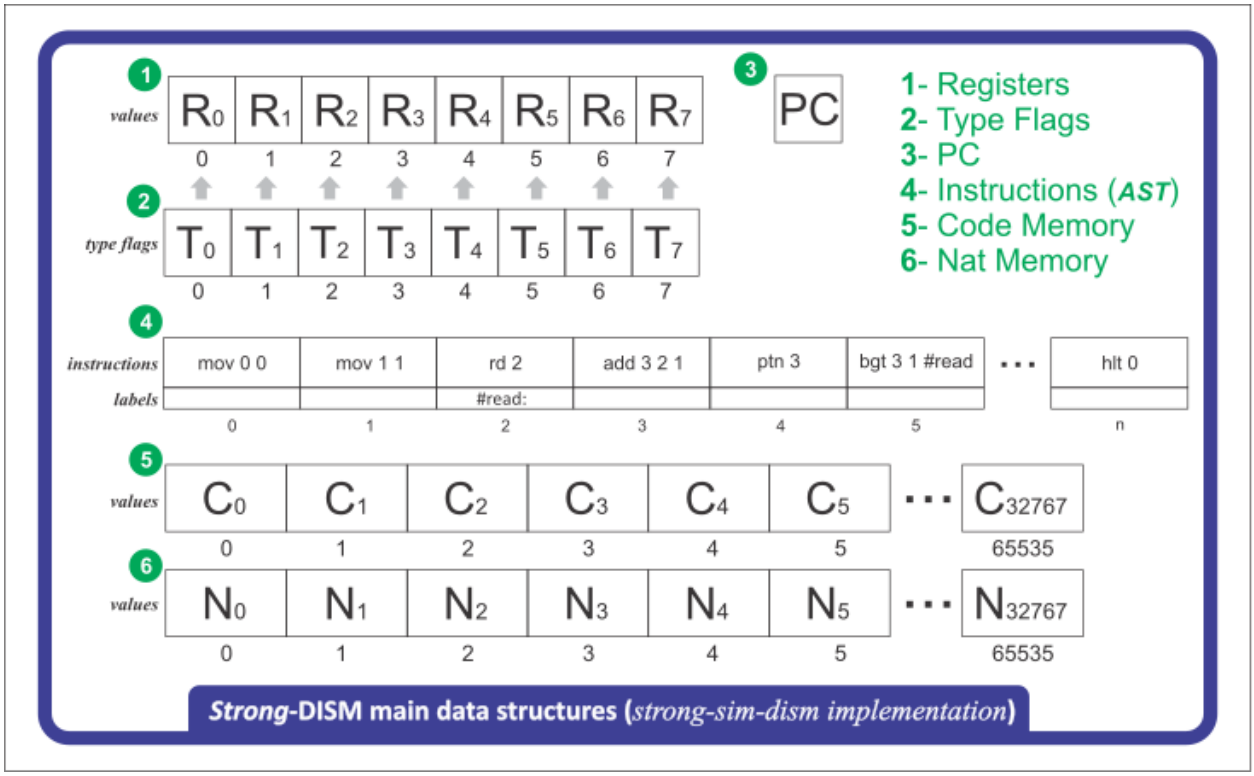

Figure 4.1 *Strong*-SIM logical data structures.

With respect to memory dedicated to data storage, the *Strong*-SIM implementation has two physically separated and distinct unsigned integer arrays to store ***nat*** and ***inst*** values respectively, being each of the arrays indexed from 0 to 32767 and each containing 32768 elements.

4.2.2 Data Structures Implementation for *Strong_Mem*-SIM

For *Strong_Mem*-SIM [94], the underlying DISM data structure used to represent the registers array was replaced by a double unsigned integer array (see figure 4.2), having one of the subarrays utilized to store register's values while the other used to annotate the ***nat*** or ***inst*** types of values inhabiting the elements of the first subarray at the same index.

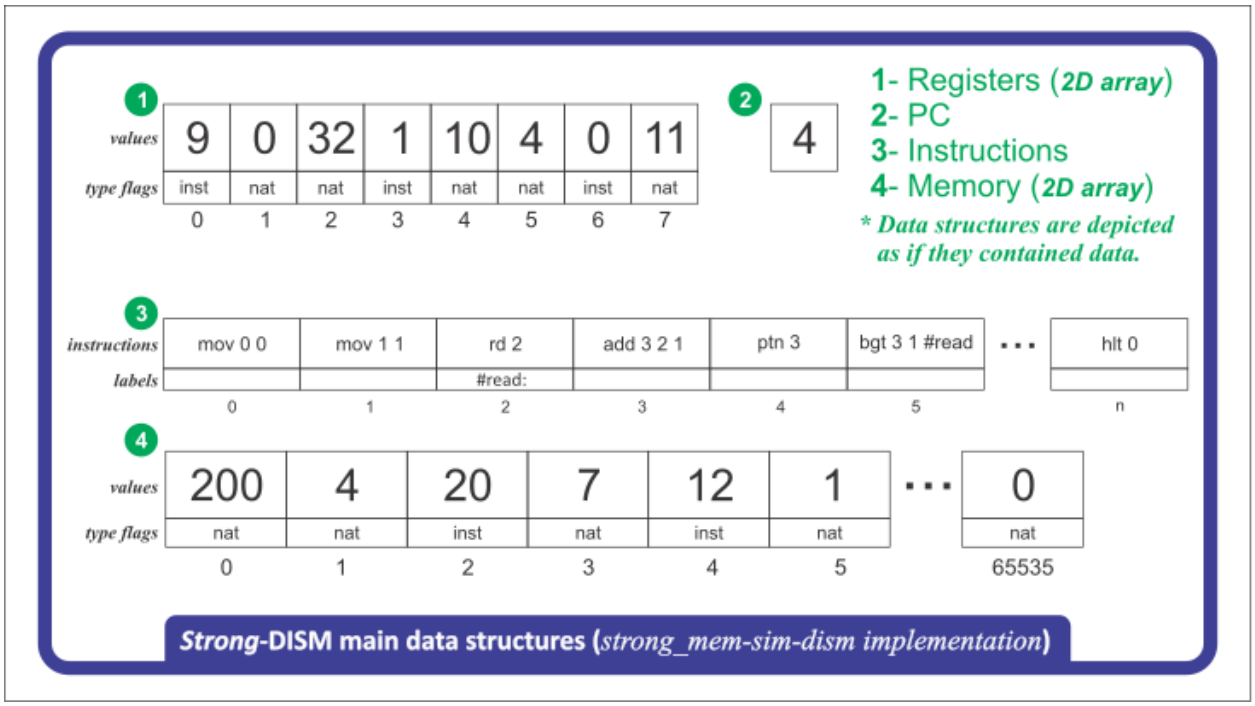

Figure 4.2 *Strong_Mem*-SIM logical data structures.

The implementation of memory, matched the registers' implementation, but this time the unsigned integer array contained 65536 elements, indexed from 0 to 65535. This 2-D array was also used exactly as the register's array was used. The elements of one subarray stored values, and the other stored the types of the values residing at the same index of the values subarray.

4.3 *Strong*-DISM Instructions

Strong-DISM has twenty instructions (see Figures 4.3 and 4.4 below). These instructions are all atomic and designed to produce a meaningful computation via side effects on memory. Except eight new instructions (*mvc*, *ldc*, *ltm*, *ltr*, *stc*, *rdc*, *bec*, and *ptr*), the remaining ones, inherited from DISM, use the same opcodes and produced exactly the same results as in DISM.

Instruction	*Machine Operation*
add d:nat s1:nat s2:nat	R[d]:nat <- R[s1]:nat + R[s2]:nat
sub d:nat s1:nat s2:nat	R[d]:nat <- R[s1]:nat - R[s2]:nat (*R[d]:nat <-0:nat when R[s2]:nat > R[s1]:nat*)
mul d:nat s1:nat s2:nat	R[d]:nat <- R[s1]:nat * R[s2]:nat
mov d:nat n:nat	R[d]:nat <- n:nat
mvc d:inst cp:inst	R[d]:inst <- cp:inst
lod d:nat s:nat i	R[d]:nat <- N[R[s]:nat + i]:nat
ldc d:inst s:nat i	R[d]:inst <- C[R[s]:nat + i]:inst
ltr d:nat s:nat	R[d]:nat <- T[s]:typ when s<>d
str d:nat i s:nat	N[R[d]:nat + i]:nat <- R[s]:nat
stc d:nat i s:inst	C[R[d]:nat + i]:inst <- R[s]:inst
jmp s:inst i	PC <- {R[s]:inst + i}:inst
beq s1:nat s2:nat n:inst	If R[s1]:nat = R[s2]:nat then PC <- n:inst
bec s1:inst s2:inst n:inst	If R[s1]:inst = R[s2]:inst then PC <- n:inst
bgt s1:nat s2:nat n:inst	If R[s1]:nat > R[s2]:nat then PC <- n:inst
rdn d:nat	Read natural number from screen into R[d]:nat
rdc d:inst	Read an instruction reference from screen into R[d]:inst
ptn s:nat	Print value of register R[s]:(nat or inst) to screen
ptr s:nat	Print type of register T[s]:typ to screen
hlt s:nat	Halt the *Strong*-DISM with code R[s]:nat

Figure 4.3 *Strong*-DISM instructions, opcodes and definitions used on *strong-sim-dism* simulator.

For both, *Strong*-SIM and *Strong_Mem*-SIM implementations, all instructions behave exactly the same way with the exception of the *ltm* instruction. The *ltm* instruction, due to the way memory was implemented, is only particular to the *Strong_Mem*-SIM implementation of *Strong*-DISM.

Type annotations shown in figures 4.3 and 4.4 are for programmers to understand how to associate the instructions opcodes with their corresponding operands' and yielding types. In practice, for *strong-sim-dism* and *strong_mem-sim-dism* simulators, types are implicitly annotated, being their inference and type-checking dynamically calculated.

Instruction	*Machine Operation*
add d:nat s1:nat s2:nat	R[d]:nat <- R[s1]:nat + R[s2]:nat
sub d:nat s1:nat s2:nat	R[d]:nat <- R[s1]:nat - R[s2]:nat (*R[d]:nat <-0:nat when R[s2]:nat > R[s1]:nat*)
mul d:nat s1:nat s2:nat	R[d]:nat <- R[s1]:nat * R[s2]:nat
mov d:nat n:nat	R[d]:nat <- n:nat
mvc d:inst cp:inst	R[d]:inst <- cp:inst
lod d:nat s:nat i	R[d]:nat <- M[R[s]:nat + i]:nat
ldc d:inst s:nat i	R[d]:inst <- M[R[s]:nat + i]:inst
ltm d:nat s:nat i	R[d]:nat <- (M[R[s]:nat + i]):typ
ltr d:nat s:nat	R[d]:nat <- (R[s]):typ when s<>d
str d:nat i s:nat	M[R[d]:nat + i]:nat <- R[s]:nat
stc d:nat i s:inst	M[R[d]:nat + i]:inst <- R[s]:inst
jmp s:inst i	PC <- {R[s]:inst + i}:inst
beq s1:nat s2:nat n:inst	If R[s1]:nat = R[s2]:nat then PC <- n:inst
bec s1:inst s2:inst n:inst	If R[s1]:inst = R[s2]:inst then PC <- n:inst
bgt s1:nat s2:nat n:inst	If R[s1]:nat > R[s2]:nat then PC <- n:inst
rdn d:nat	Read natural number from screen into R[d]:nat
rdc d:inst	Read an instruction reference from screen into R[d]:inst
ptn s:nat	Print value of register R[s]:(nat or inst) to screen
ptr s:nat	Print type of register R[s]:typ to screen
hlt s:nat	Halt the *Strong*-DISM with code R[s]:nat

Figure 4.4 *Strong*-DISM instructions, opcodes and definitions used on *strong_mem-sim-dism* simulator.

To further clarify on the *Strong*-DISM instructions presented in figures 4.3 and 4.4, *n* can be replaced only for a natural number, *i* for an integer, *s* for a source memory (from where a value is read), and d for a destination memory (to where a value is written). ***typ*** denotes a natural number value assigned by *Strong*-DISM to represent a type. Both *Strong*-SIM and *Strong_Mem*-SIM implementations encode '0' and '1' ***nat*** values as ***typ*** values to keep track of ***nat*** and ***inst*** types respectively. Although ***typ*** values can treated as ***nat*** data, they cannot be assigned nor modified by the programmer.

The *lod* and *str* instructions automatically check that programs can only access ***nat*** typed memory, and the *ldc* and *stc* instructions enforce that programs can only access ***inst*** typed memory. As in DISM, the *jmp*, *beq*, and *bgt* instructions, in conjunction with the newer *bec* instruction, can write to the PC, and *Strong*-DISM –as DISM does- transparently ensures that the

PC can only be written with references to instructions that are part of the program. In the case of *jmp* instruction, the operand for this instruction must be of type ***inst***. The *ltm* and *ltr* instructions allow to check the type of a value in memory or in a register respectively. Note that the *ltm* instruction is not present in *Strong*-SIM, as memories for ***nat*** and ***inst*** types are physically separated; therefore, this instruction becomes implicit for *Strong*-SIM. All other instruction including *rdc*, *rdn*, *ptr*, and *ptn* facilitate the interaction with a program's user by allowing the input or output of data. Finally, note that instructions will accordingly set the register's type flags (and memory type flags for the case of *strong_mem-sim-dism*) after a successful type checking.

As in DISM, opcodes are restricted to be in lowercase, and all *Strong*-DISM programs must successfully be ended with the *hlt* instruction. This rule disallows a program from executing nonexistent instructions. The *hlt* instruction is of type ***nat***. Comments and symbolic labels are inherited from DISM and behave the same way. Section 4.6 contains an example of the use of symbolic labels in *Strong*-DISM.

4.4 *Strong*-DISM Implementation

Although some of the register and memory implementations' details have been already revealed, there are still some pertinent operational details to be mentioned. All *Strong*-DISM programs, before being executed, are converted into an AST object that is dynamically evaluated. During the initialization of the virtual machine, all the values of the registers, type flags, PC and arrays are initialized to the symbol '0', which may have a different meaning on each instance. For example, initializing all type flags to zero is equivalent to initializing all memory and register to the ***nat*** type; while the PC would be pointing to the first instruction.

During execution, the PC is incremented by 1 every time an instruction is executed –with the exception of instructions that write to the PC– thus the PC is always referencing to the next program instruction on the instruction's array. Each instruction that is executed is dynamically

type checked in a specific order. For example, the *add* instruction first checks the types of the source registers corresponding to the second and third operands, to later set the type flag of the destination register to a ***nat*** type and finally writes the results of the addition as a value to the destination register. This whole instruction is considered to be performed atomically, and a further observation is that if multiple programs could be interpreted concurrently, then the value-and-type updates are to be performed atomically.

Although *Strong*-DISM inherits the DISM mechanism that ensures that every instruction being executed belongs to the program, instructions like *mvc*, *ldc*, and *rdc* that could potentially allow the insertion of an ***inst*** value that does not belong to the program; hence, *Strong*-DISM verifies that any value inserted by means of any of these instructions exist in the AST object. Furthermore, as the yielding types of the *mvc*, *ldc*, *stc*, *jmp*, *beq*, *bec*, *bgt*, and *rdc* instructions are of type ***inst***, *Strong*-DISM guarantees that values from these instructions are written to the appropriate memory or register type.

4.5 *Strong*-DISM Security

Previous research has demonstrated that typed assembly languages, when used as target languages, are fully capable of extending the type soundness guarantees provided by higher level languages [23,24]. Even though *Strong*-DISM is an instance of the simplest typed assembly language, as it has the minimal amount of types possible, does not lessen at all the intrinsic level of protection offered by types. Just adding types eliminates the classic version of the buffer –or stack– overflow exploits (see section 5.4), as the language guarantees that values of a given type will be accessed under the rules created for that specific type. Another aspect not to be overlooked is that higher level types need to be accompanied by the corresponding set of instructions so that the safety properties of programs are preserved by the target language when translated from higher level languages. When considering the typing rules of each new

instruction added to the target typed assembly language is paramount to be extremely careful to avoid puncturing the implicit soundness of types.

Strong-DISM does not prevent the execution of programs with levels of logic above the level of the language that may contain erroneous logic or malicious intentions; however, the language contains an articulated set of typed instructions that guarantee that types will be appropriately accessed and yielded; thus automatically and properly preserving types along the computation. This preservation of types is consistent with concept of preserving safety properties in a program, and can be used to prove that a program will behave as expected during all phases of its execution.

4.6 A *Strong*-DISM Program Example

The following code is the *Strong*-DISM equivalent to the first DISM programs shown in chapter 3.7 as it uses symbolic labels.

Program 1 uses symbolic labels.

```
       mvc 0 0       ;move value 0 into register 0
       mov 1 0       ;move value 0 into register 1
       rdn 2         ;read n into register 2
       rdn 3         ;read m into register 3
       mov 4 1       ;move value 1 into register 4
#LOOP: beq 2 1 #END  ;if m==0 then goto end
       ptn 2         ;print n
       sub 3 3 4     ;decrement m
       jmp 0 #LOOP   ;goto loop beginning
 #END: hlt 0         ;halt with code 0
```

Program 2 does not use symbolic labels.

```
       mvc 0 0       ;move value 0 into register 0
       mov 1 0       ;move value 0 into register 1
       rdn 2         ;read n into register 2
       rdn 3         ;read m into register 3
       mov 4 1       ;move value 1 into register 4
       beq 2 1 9     ;if m==0 then goto end
       ptn 2         ;print n
       sub 3 3 4     ;decrement m
       jmp 0 5       ;goto loop beginning
       hlt 0         ;halt with code 0
```

Note that most DISM programs can be executed on *Strong*-SIM and *Strong_Mem*-SIM, and all *Strong*-SIM programs can be executed by *Strong_Mem*-SIM. On the other hand, not all *Strong_Mem*-SIM programs cannot be executed by *Strong*-SIM nor by the DISM simulator.

CHAPTER 5: ANALYSIS AND TESTING

This chapter presents the results of different benchmarking tests destined to compare how the added dynamic type checking is detrimental to the runtime performance of *Strong*-DISM against DISM. Some analyses of how type checking occur may be presented in some cases. All the binaries for the simulators, their definitions, the code used to create the benchmark program, and all the test files are available for download and closer examination at:

http://research.binaryworldnexus.com

5.1 The Impact of Type Checking for *Strong*-DISM Performance

Type checking of *Strong*-DISM instructions (see 'Machine Operations' section from figures 4.3 and 4.4) extends the checks already performed by DISM. For example, the DISM version of the *sub* instruction checks that the registers declared on the instruction are valid, and that the minuend value is larger than the subtrahend. The same instruction for *Strong*-DISM requires two additional checks: the types of the registers for where the values for the minuend and subtrahend reside must be ***nat***. *Strong*-DISM, additionally, has to set the value of the flag for the destination register to ***nat*** as well. As the number of check and writes increase, the amount of computations needed per instruction increases as well, and this increase may significantly reflect on the overall performance, feasibility, and worthiness of the enhancements. With that in mind, a series of runs to benchmark performance were realized.

5.2 Benchmarking *Strong*-DISM and DISM Instructions

In order to produce comparable benchmarks to analyze how the addition of type checking significantly affected the performance of *Strong*-DISM against the performance of DISM, a

series of DISM and *Strong*-DISM assembly programs of varying complexity were used with the following intentions: first, to compare the executions of the same DISM program in the DISM simulator (*sim-dism*) against (when possible) the two implementations of *Strong*-DISM simulators (*strong-sim-dism* and *strong_mem-sim-dism*). Second, to compare equivalent DISM and *Strong*-DISM programs containing each one instances of their particular set of instructions. Also, as the exact number of instructions for DISM and *Strong*-DISM is known, the dynamic order of execution of instructions is known, and the underlying data structures are known, a more precise calculation of the amount of instructions executed is possible to be combined with logical analysis about how the underlying data structures were accessed by the interpreters, and how the interaction of some other system factors may have affected the benchmark results.

In general, five DISM programs were used. These programs were conceived so that they could be run on the DISM simulator and on any of the two of *Strong*-DISM simulators. Then, out of the original five DISM programs, the logic of three of these was replicated to produce equivalent *Strong*-DISM programs able to be run on all of the *Strong*-DISM simulators; while the remaining two programs, since they contained the *ltm* instruction, were replicated to be run only on *Strong_Mem*-DISM simulator.

Each of the programs was executed 100,000 times in a round robin fashion and benchmarked in the following order: 1) *simple.dism*, and *simple.stdism*, 2) *nm.dism*, and *nm.stdism*, 3) *edgy.dism*, and *edgy.stmdism*, 4) *divide_nat.dism*, and *divide_nat.stdism*, 5) *stor_lod_test.dism*, and *stor_lod_test.stmdism*. Programs with extension '.dism' were executed by all interpreters; programs with extension '.stdism' were executed by *strong-sim-dism* and *strong_mem-sim-dism*; and programs with extension '.stmdism' could only be executed by *strong_mem-sim-dism*.

Table 5.1: Statistics for all executables on simple.dism.

	Filename: simple **Filename language:** DISM		
Executing Programs:	*sim-dism*	*strong-sim-dism*	*strong_mem-sim-dism*
User Time (in microseconds)	**Average:** 6.01	**Average:** 6.02	**Average:** 6.04
	Total: 601421	**Total:** 602000	**Total:** 603900
System Time (in microseconds)	**Average:** 175.16	**Average:** 174.83	**Average:** 175.17
	Total: 17516189	**Total:** 17483230	**Total:** 17517487
AVG user time % of (+/-) improvement over a DISM program on the *same* file:		**-0.16%**	**-0.49%**

Table 5.2: Statistics for simple.dism and simple.stdism.

	Filename: simple **Language:** DISM	**Filename:** simple **Language:** *Strong*-DISM	
Executing Programs:	*sim-dism*	*strong-sim-dism*	*strong_mem-sim-dism*
User Time (in microseconds)	**Average:** 6.01	**Average:** 6.03	**Average:** 6.04
	Total: 601421	**Total:** 603231	**Total:** 604016
System Time (in microseconds)	**Average:** 175.16	**Average:** 175.16	**Average:** 174.97
	Total: 17516189	**Total:** 17516403	**Total:** 17497193
AVG *user* time % of (+/-) improvement over a DISM program on *equivalent* files:		**-0.33%**	**-0.49%**

Table 5.3: Statistics for all executables on nm.dism.

	Filename: nm **Filename language:** DISM		
Executing Programs:	*sim-dism*	*strong-sim-dism*	*strong_mem-sim-dism*
User Time (in microseconds)	**Average:** 6.57	**Average:** 6.69	**Average:** 6.77
	Total: 657221	**Total:** 669140	**Total:** 677072
System Time (in microseconds)	**Average:** 192.46	**Average:** 192.16	**Average:** 192.19
	Total: 19246126	**Total:** 19215646	**Total:** 19219049
AVG user time % of (+/-) improvement over a DISM program on the *same* file:		**-1.82%**	**-3.04%**

Table 5.4: Statistics for executing nm.dism on *sim-dism*, and nm.stdism on all *Strong*-DISM simulators.

	Filename: nm **Language:** DISM	**Filename:** nm **Language:** *Strong*-DISM	
Executing Programs:	*sim-dism*	*strong-sim-dism*	*strong_mem-sim-dism*
User Time (in microseconds)	**Average:** 6.57	**Average:** 6.59	**Average:** 6.79
	Total: 657221	**Total:** 658760	**Total:** 679103
System Time (in microseconds)	**Average:** 192.46	**Average:** 192.21	**Average:** 192.18
	Total: 19246126	**Total:** 19220825	**Total:** 19217532
AVG *user* time % of (+/-) improvement over a DISM program on *equivalent* files:		**-0.3%**	**-3.34%**

Table 5.5: Statistics for all executables on edgy.dism.

	Filename: edgy **Filename language:** DISM		
Executing Programs:	*sim-dism*	*strong-sim-dism*	*strong_mem-sim-dism*
User Time (in microseconds)	**Average:** 6.93	**Average:** 7.07	**Average:** 6.93
	Total: 693295	**Total:** 706543	**Total:** 693290
System Time (in microseconds)	**Average:** 200.20	**Average:** 203.93	**Average:** 198.80
	Total: 20020409	**Total:** 20392669	**Total:** 19880215
AVG user time % of (+/-) improvement over a DISM program on the *same* file:		**-2.02%**	**0%**

Table 5.6: Statistics for executing edgy.dism on *sim-dism*, and edgy.stdism on *strong_mem-sim-dism*.

	Filename: edgy **Language:** DISM	**Filename:** edgy **Language:** *Strong_Mem*-DISM
Executing Programs:	*sim-dism*	*strong_mem-sim-dism*
User Time (in microseconds)	**Average:** 6.93	**Average:** 7.01
	Total: 693295	**Total:** 701144
System Time (in microseconds)	**Average:** 200.20	**Average:** 200.84
	Total: 20020409	**Total:** 20083714
AVG *user* time % of (+/-) improvement over a DISM program on *equivalent* files:		**-1.15%**

Table 5.7: Statistics for all executables on divide_nat.dism.

	Filename: divide_nat **Filename language:** DISM		
Executing Programs:	*sim-dism*	*strong-sim-dism*	*strong_mem-sim-dism*
User Time (in microseconds)	**Average:** 8.90	**Average:** 8.94	**Average:** 8.94
	Total: 890159	**Total:** 893509	**Total:** 894440
System Time (in microseconds)	**Average:** 236.25	**Average:** 237.62	**Average:** 234.66
	Total: 23624834	**Total:** 23762027	**Total:** 23465721
AVG user time % of (+/-) improvement over a DISM program on the *same* file:		**-0.45%**	**-0.45%**

Table 5.8: Statistics for executing divide_nat.dism on *sim-dism*, and divide_nat.stdism on all *Strong*-DISM simulators.

	Filename: divide_nat **Language:** DISM	**Filename:** divide_nat **Language:** *Strong*-DISM	
Executing Programs:	*sim-dism*	*strong-sim-dism*	*strong_mem-sim-dism*
User Time (in microseconds)	**Average:** 8.90	**Average:** 9.01	**Average:** 9.23
	Total: 890159	**Total:** 900632	**Total:** 923472
System Time (in microseconds)	**Average:** 236.25	**Average:** 236.53	**Average:** 237.69
	Total: 23624834	**Total:** 23652618	**Total:** 23769484
AVG *user* time % of (+/-) improvement over a DISM program on *equivalent* files:		**-1.23%**	**-3.7%**

Table 5.9: Statistics for all executables on stor_lod_test.dism.

	Filename: stor_lod_test **Filename language:** DISM		
Executing Programs:	*sim-dism*	*strong-sim-dism*	*strong_mem-sim-dism*
User Time (in microseconds)	**Average:** 7.64	**Average:** 7.98	**Average:** 7.73
	Total: 763500	**Total:** 798215	**Total:** 773144
System Time (in microseconds)	**Average:** 180.27	**Average:** 186.52	**Average:** 180.11
	Total: 18027276	**Total:** 18652251	**Total:** 18011302
AVG user time % of (+/-) improvement over a DISM program on the *same* file:		**-4.45%**	**-1.17%**

Table 5.10: Statistics for executing stor_lod_test.dism on *sim-dism*, and stor_lod_test.stdism on *strong_mem-sim-dism*.

	Filename: stor_lod_test **Language:** DISM	**Filename:** stor_lod_test **Language:** *Strong_Mem*-DISM
Executing Programs:	*sim-dism*	*strong_mem-sim-dism*
User Time (in microseconds)	**Average:** 7.64	**Average:** 7.71
	Total: 763500	**Total:** 771170
System Time (in microseconds)	**Average:** 180.27	**Average:** 178.66
	Total: 18027276	**Total:** 17866240
AVG *user* time % of (+/-) improvement over a DISM program on *equivalent* files:		**-0.91%**

Benchmarking was performed on a Unix system running OS X 'EL Capitan'. The system had a quad-core Intel Core i7 processor running at 2.2 GHz, 16 GB of 1600 MHz DDR3 memory, and a solid state drive of 1 TB capable of read/write speeds of up to 2GBps. The program responsible for benchmarking the executables was written in C. At its core, the program consisted of two nested for loops. The inner loop called the *time.h*'s library function *getrusage* before and after a call to the *system* function from the *stdlib.h* library. The *system* call executed a set of programs matched with their different possible interpreters as around robin. The outer loop executed its contents 100,000 times in this instance. The user time was computed by adding the ru_utime.tv_sec and ru_utime.tv_usec values of each program, while the system time was computed as the user time but using the ru_stime.tv_sec and ru_stime.tv_usec values.

5.3 What the Benchmarks Indicate

Tables 5.1 to 5.10 show the results of the performance of *Strong*-DISM for its two implementations, against the performance of DISM's implementation. Average and total times are all given in microseconds. The results are separated by *user* time, the time it takes a program to perform its instructions, and by *system* time, the time it takes the system to perform system's operations related to the program being executed but at the kernel level.

In the case of *user* time, the time of our interest, it contains a blend of code statements that range from creating the DISM or *Strong*-DISM program's AST object –which is negligible— to interpreting the program instructions, which in some cases produce outputs to the screen. There are two types of resulting tables, the ones that show the three simulators (*sim-dism*, *strong-sim-dism*, and *strong_mem-sim-dism*) running the same DISM program, and the ones where the run of a DISM program is compared against an equivalent *Strong*-DISM program, each one being ran on its respective simulator.

In the first case, the idea is to observe how fast the DISM instructions are run by *Strong*-DISM implementations despite the added type checks and the memory reorganization. In the second case, the idea is to observe how two equivalent programs one created and containing instructions particular to DISM compares with an equivalent one designed to carry out most of the instructions particular to *Strong*-DISM. In the second case, the correspondence would not be in a one to one fashion, but would suggest how fast or slow would *Strong*-DISM programs run against their DISM counterparts. At the bottom of every table, a value in the form of a percentage indicates how fast or slow the *Strong*-DISM simulators performed against the DISM simulator on the same or an equivalent program. A valid note is that for all DISM programs to be able to run on any of the *Strong*-DISM interpreters, they are not to contain the DISM version of the *jmp* instruction. So, this instruction has been appropriately replaced by the branching *beq* and *bgt* instructions.

For example, if DISM interpreter spends 10 microseconds executing program1.dism, and *Strong*-DISM spends 11 microseconds executing the same program, then the *Strong*-DISM interpreter will have spent 1 microsecond more than the DISM interpreter or 10% more time. Therefore, the percentage of improvement on 10 microseconds is reflected as a negative percentage (-10%), because *Strong*-DISM took more time to execute the same program, and therefore, it performed slower. Had *Strong*-DISM executed the program in 9.8 microseconds, then there had been an improvement of 2%, as *Strong*-DISM would completed its execution 0.2 microseconds sooner than DISM; thus performing faster.

As it was expected, *Strong*-DISM implementations performed slightly slower than DISM's, being the only exception the execution of program *edgy.dism* (See tables 5.5 and 5.6). This program simulates a user inputting the nodes and edges on a connected graph, to later, print the edges of the recalled nodes. The simulated input of a user entering the edges for given nodes

was 1,1,1,2,1,3,1,4,2,2,2,3,2,4,3,3,3,4,4,4,0; followed by the input 1 to output edges of node 1 would output 1,2,3,4; the input 2 to output edges of node 2 would output 2,3,4; the input 3 for node 3 would output 3,4; and the input 4 for node 4 would output 4. Finally, the input 0 would terminate the program. This program, as it executes, makes many reads and writes to not contiguous memory elements in order to store and recall the nodes and edges at a location in memory; performs all its computations over ***nat*** values to calculate edges and nodes initial locations; and jumps randomly plenty of times. The equivalent program with extension '.stmdism' contains the same instructions, except that it uses the *ltm* instruction and has an added subroutine to branch to a halt if the expected type of the value found in memory is not of the type ***nat***.

The speculative observation indicates that *strong_mem-sim-dism* and *sim-dism* implementations made the same jumps when interpreting the DISM native program as both interpreters share a similar register and continuous memory structure. The only difference is that the typed interpreter uses a double array for its memory and registers, but this seems to be negligible when type checking values at a memory, or at a register, as they both benchmarked the same time (0% difference) with very similar system times. However, there were some more jumps for *strong-sim-dism* occurring probably between the type, segregated memories, and register arrays as it reflected an extra -2.02% time consumption. Because both implementations of *Strong*-DISM performed the same type checking of the native DISM program, the time spend for type checking could be discarded as one of them tied in execution, leaving only the additional jumps between arrays as reasons for the differential –observe how system time is higher.

Later, when the *Strong*-DISM native file was executed on *strong_mem-sim-dism* it yielded a negative 1.15%, and this toll on performance can be traced to the use of the *ltm*

instruction in conjunction with an extra comparison the program does. In this instance the amount of system time does not increase by much.

Programs containing this type of instructions do not seem to impact by much the performance of *Strong*-DISM when compared to DISM, especially the performance where the memory is continuous; less than a 1.5% decrease in performance does not seem to be a negative tradeoff when gaining the possibility of soundness is at stake.

For *stor_lod_test.dism* (See tables 5.9 and 5.10), a program very similar to *edgy.dism, Strong*-DISM implementations performed similarly to *edgy.dism*. *Strong*-DISM interpreters degraded their performance executing a native DISM program to -4.45% for *strong-sim-dism*, and -1.17% for *strong_mem-sim-dim*. Here the difference was that the writes and reads made by the program were not too randomized, but over contiguous memory, and that the jumps in memory were neither too distant from each other's locations. The same patterns observed for *edgy.dism* re-emerged during this execution. *stor_lod_test.stmdism*, the equivalent *Strong*-DISM native file also behaved as in the *edgy.stmdism*'s execution. However, this time, ***nat*** and ***inst*** values were stored to memory from registers and loaded from memory into registers during a combination of all the provided instructions. A -0.91% of performance degradation can be considered as a very benign sign.

The *divide_nat.dism* and *divide_nat.stdism* programs consist of the division of the ***nat*** 10,000 by the ***nat*** 2, and the final result of the division printed to the screen. Both programs are quasi-identical, except that the *divide_nat.stdism* contains the *Strong*-DISM version of the *jmp* instruction. These programs are designed to explore the performance of computations that only utilize the registers and, therefore, do not access data memory.

The benchmarks for their executions (See tables 5.7 and 5.8) show that the native DISM program executed by the *Strong*-DISM interpreters produced an equal but negligible -0.45%

performance degradation for *strong-sim-dism* and *strong_mem-sim-dism*, while the results for *divide_nat.stdism* execution show that one the two implementations of *Strong*-DISM may have benefited more from the underlying structure supporting type annotation for registers when running code containing instructions native to the language with -1.23% for *strong-sim-dism* and -3.37% for *strong_mem-sim-dism*. In general, these benchmarks show that the implementation of dynamic type checking does not degrade the execution of programs by much.

The files *nm.dism* and *nm.stdism* programs are very similar with the exception that *nm.stdism* contains the *jmp* instruction from *Strong*-DISM language. They both are designed to produce computations on registers. Both programs simulate a user entering two numbers where the first number is printed to screen as many times as the second number. In essence, these programs are very similar to *divide_nat.stdism* or *divide_nat.dism* as they do not access the data memory to perform stores or loads, and all the action is focused on the registers. The benchmarks (see tables 5.3 and 5.4) show that *strong-sim-dism* performance's degradation (-1.82% and -0.3) is slightly smaller that *strong_mem-sim-dism* (-3.04% and -3.34%), but they are within the expected range of degradation based on other similar executions.

Finally, on the *simple.dism* program the Strong-DISM simulators performed as expected, -0.16% for *strong-sim-dism* and -0.49% for *strong_mem-sim-dism*. The execution of program *simple.stdim* yielded -0.33% for *strong-sim-dism* and -0.49% for *strong_mem-sim-dism*. This program simply printed the initial ***nat*** value of a register and halted for the DISM version, and printed the initial nat value, printed the type of the register, and the halted for the *Strong*-DISM version. The type checks for the *Strong*-DISM version only happen during execution of the *hlt* instruction, which is minimal. Here the benchmarks suggest that the *strong-sim-dism* implementation of type annotation for registers performs a little bit faster as that is the only

difference between the *Strong*-DISM implementations. Nevertheless, the overall results for these executions are positive since the performance degradation is less than 1% for all.

The overall performance of the *Strong*-DISM interpretes behaved as expected, with the tendency of *strong-sim-dism* to perform better in computations that just used registers and *strong_mem-sim-dism* to perform better on programs that made stores and loads to and from data memory. The average of all the combined performance degradations for all the *Strong*-DISM benchmarks yielded a mere -1.41% which is not significantly detrimental considering the possibility of gaining programs' soundness with respect to types.

One of the positive aspects observed during this benchmarking was the enhanced visibility of every step of the program execution, as not only all the steps could be output to the screen, but every step of execution could be played one by one as they happened. This was an extension to what was already built in DISM. Also the operational rules for type checking were very clear and straightforward to apply, leaving little room for implementation or logical errors.

5.4 Types in Action

The programs of figures 5.1 and 5.2 (see below) contains code that simulates the conditions present during a buffer overflow. Suppose that program 1 is executing and runs out of execution time at instruction 10, *add 2 3 4*, and it is sent the waiting queue. Then, program 2 executes and finishes. When program 1 resumes its execution, the return address that was stored on the array will have changed as it had been overwritten by program 2. Then, in the case of DISM, when the *jmp* instruction of program 1 is executed, if the value of the overwritten instruction is part of the program, it will allow the jump, but the program will not follow its expected execution (unless the return address had been overwritten with the same value), or the program execution will be interrupted.

```
1           ;Program 1 – Writes an array of ten elements, stores the return
2           ;address, and loads the RA, before jumping to it
3            mov 0 0            ;
4            mov 2 0            ; used as a counter
5            mov 3 1            ; used to increment by 1
6            mov 4 10           ; end of the loop
7            mov 6 #RA          ; put return address in register 6
8#WRITE:     str 2 0 2          ; storing a sequence
9            add 2 2 3          ; increasing register 2 by 1
10           bgt 4 2 #WRITE     ;
11           str 2 0 6          ; storing return address
12           add 2 3 4          ; program performs some other computation
13           sub 1 3 4          ; program performs some other computation
14           mov 2 0            ; program computes return address location
15#GETRA:    add 2 2 3          ;
16           bgt 4 2 #GETRA     ;
17           lod 1 2 0          ; program loads return address
18           jmp 1 0            ; program jumps to return address
19#RA:       ptn 2              ;
20           ptn 3              ;
21           ptn 4              ;
22           ptn 6              ;
23#END:      hlt 0              ;
```

Figure 5.1 Example of conditions needed for a buffer overrun. Program 1.

```
0           ;Program 2 – Writes an array of 15 elements containing
1           ;the nat 18
2            mov 0 0            ;
3            mov 1 13           ;
4            mov 4 15           ;
5            mov 2 0            ;used as a counter
6            mov 3 1            ;
7            ptn 1              ;
8#WRITE:     str 2 0 1          ;storing 15 elements
9            add 2 2 3          ;increasing number stored on register 2
10           bgt 4 2 #WRITE     ;
11#END:      hlt 0              ;
```

Figure 5.2 Example of conditions needed for a buffer overrun. Program 2.

If the same program is executed on the *Strong*-DISM interpreter it will produce a dynamic type checking exception, because in the context of *Strong*-DISM, the *jmp* instruction operates over values of type ***inst***, and the value overwritten by program 2 is of type ***nat***. That is one of the beauties of type safety; values will be operated on in accordance to their types. One of the instructions, *ltm*, allows for a program to check the type of a value existing in data memory before being operated on, offering a great type-safety tool for programmers and for languages that use *Strong*-DISM as a target language.

An observation about typed instructions is that they can be restricted to be used by low level program from the operating system, eliminating the possibility of manipulation by user programs, in a similar fashion to the way automatic memory allocation and garbage collection is used.

CHAPTER 6: *Strong*-ARM, *Strong*-RISC, *Strong*-ETC.

The idea of implementing support for *Strong*-style type checking on RISC processors is not as far-fetched as it seems, especially with the tendency of buses to grow in size. With the latest standardization of 64-bit architectures, larger instruction sizes could allow the use of some bits to annotate the operands' types for a diverse set of instructions. These annotations, in the form of flagging bits or integers, in conjunction with a program's types lookup table could well serve as the structural ingredients needed to enforce dynamic type checking.

Besides the creation of new instructions, *Strong*-DISM basically associated the operands of instructions with types (by the implicit use of labeling), and later, checked the types of the operands before executing the instructions. Although, there was no "overloading" of any instruction, instructions such add or subtract could have been overloaded if it had been needed. For example, instruction variants like *add* r1:***nat*** r1 r2 and *add* r1:***inst*** r1 r2 would have served to indicate the overloaded addition of two natural numbers or two instructions (given the case that in reality instructions could be added).

In the case of *Strong*-DISM's particular implementation, and for the sake of clarity, the types have been statically encoded and embedded in the language. However, a version of the language in which the programmer would have declared and added arbitrary types to the language to be type checked dynamically at runtime could have been implemented. Just the programmer's or compiler's declaration of an associative lookup table containing the instructions and the possible types of operands allowed to be executed (and yielding types allowed to be produced) at runtime by an instruction, would have sufficed. If such table had been

created, then the insertion of “on the fly” arbitrary types by the programmer –or compiler— could have been type checked against such table at runtime anytime an instruction would use an operand. Another interesting aspect of this table is that for the concurrent executions of multiple programs, each program would have referred to its own table for type checking.

For the implementation of a *Strong*-ARM version of a dynamically typed assembly language, ARMv7 and 8 architectures already possess a Protected Memory System Architecture (PMSA) based on a Memory Protection Unit (MPU) that could be extended to preserve and access such tables in a read-only section of memory. Such customization would have allowed to read and correlate type flags for values existing on registers or memory with their respective types before instructions’ execution. This proposed system would work automatically in system ‘privileged’ mode, and could, for simplicity, allow the enumeration of types from a given compiler for easier implementation. The main obstacle for this dynamic type checking system would strive in the lack of bits to flag or annotate the type of a value. However, a solution around this issue could be to find a specific format to write the data. For example, similar to the arrays declare their size, in which section contiguous to the array is used as a header containing the array’s size, a header containing an encoded integer value as the datatype could be used as well. The beneficial part is that there is no limit to the amount of types that could be used. The assembler needed to support this mechanism would need to take care of creating a section to declare the association of types with instructions.

Initial implementation of *Strong*- over ARM or over any other RISC (or non-RISC) based system, would first focus on adding type support to a large enough subset of instructions from the target assembly language, to allow establishing and proving soundness for programs based on the selected instructions subset.

CHAPTER 7: CONCLUSION

7.1 Summary

To the effect of exposing the possible conveniences and benefits of a dynamic type checking discipline for the study and production of typed assembly languages, a series of formal steps were taken. First, a search for instances of related work is performed; second, an untyped assembly language to be used as base language for a transformation is presented; then, a dynamically typed assembly language is developed from the base language and implemented as two variants of the same language with different memory and registers structures, and layouts; and finally, benchmarks for the new language implementation, in order to analyze its performance and feasibility of the dynamic approach, are obtained.

During the research of related work, the closest topics found directly connected to this thesis were proof-carrying-code, as the oldest; typed assembly language, as the closest; and gradual typing as the most recent, but very close topic as well. Close attention was paid to the characteristics of dynamic typing intrinsic dynamic nature of the research.

The scrutiny for TALs revealed that a large body of knowledge with validated research exists on the topic; however, most of such work has been done following a static type checking discipline. This static approach, due to the bloated and complex code it usually produces, was found to require in most cases, a large and complex set of tools, e.g., formal methods and theorem provers, to be able to demonstrate the validity of these TALs typing rules as capable of preserving the characteristics that render a language as sound.

The search for related work on the gradual typing area, similar to TALs, also revealed the prevalence of the static type checking discipline.

Throughout the course of *Strong*-DISM's development the following aspects were found to be beneficial and convenient:

- The implementation was performed following a relatively easy, straightforward, and enumerable set of steps.
- Simplicity allowed to eliminate hidden runtime aspects of the implementation.
- The runtime rules developed were properly aligned and corresponded with the dynamic behavior of computations due to the natural parallelism between dynamic type checking and the way computations happen at the physical layer.
- Large and complex static type checking-related analysis tools did not have to be used [3,4,5,6,7].
- The code produced was not bloated at all.
- The transfer of abstract concepts to code was clear and straightforward.
- As *Strong*-DISM was treated as a target language, investigation areas concerning compilation to D-TAL needed not to be covered.
- A large body of extensively validated knowledge on TAL is found and it directly apply and inserts into the theoretical aspects of D-TAL [2,8,13,14,18,19,20,22,23, 24,25].
- Research and implementation times and cost were reduced.
- Resulting language is able to support gradual typing.
- D-TAL and static TAL versions equivalently support type's enforcement.
- A remarkable gain of directly observable insights about the mechanics of dynamic type checking for TALs is gathered.

- Because this version of D-TAL was implemented as a virtual machine, it has made visible, with great level of detail, the behavior of abstract concepts and practical mechanisms that would, otherwise, been obscured during the transformation of DISM to *Strong*-DISM.
- A relative small number of instructions is really needed to enforce type safety at the assembly level.

The benchmarks revealed that the two *Strong*-DISM implementation variants did have an averaged performance degradation of -1.41% with respect to DISM, the base language. Performance differences for the *Strong*-DISM implementations were noticeable in programs that made stores and loads versus programs who just performed on registers.

In general, besides the involvements related to the language's development process, the only physical consideration was the need of extra memory, or at least the consideration of a typed memory to implement the flagging of values by type, which could be easily implemented on existing architectures by the utilization on extra bits to flag values' types. Also a variant of this implementation that included an associative lookup table of operands types would allow the dynamic addition of types with no limit on the amount of types being added.

7.2 Recommendations

The following practical recommendations may prove to be beneficial for the implementation of low level dynamic type checking: it must be implemented to be handled automatically by the system in a sort of privileged mode; it must be kept transparent to the programmer to guarantee no tampering with the value-types –analog to the way memory allocation renders a language unsound if the action is left under the programmer's control versus having a system of memory allocation and garbage collection; the type checking rules must be implemented as close to the physical layer as possible, if not at the physical layer, to make

possible not only a faster execution but to reduce the possibility of tampering. Finally, to close the gap of exploits utilizing memory as a channel of insertion, memory must be always type checked, and values that could be construed as operands must always be type-annotated. If the total physical memory of a system is too expensive to type check, there should be, at least, some level of type checking present ultimately at the closest to registers cache level, or (maybe in detriment of the instruction size) a pre-conceived format that reserved bits to be used as type markers could be used.

7.3 Future Work

Future work would involve producing an implementation of *Strong*- over ARM (or any other feasible existing processor) including the provision of formal type safety and soundness proofs of *Strong*-'s execution on programs translated from a given high level language. Also the provision of more carefully crafted benchmarks on the performance of *Strong*-'s in general, but with emphasis on the analysis of type checking on flagged memory containing contiguous multiple datatypes performance would be considered for future work.

REFERENCES

[1] Schneider, Fred B., Morrisett, Greg., and Harper, Robert. “A language-based approach to security.” *Informatics* Springer, Berlin, Heidelberg. Pages 86-101, 2001.

[2] Morrisett, Greg. *Compiling with types*. Carnegie-Mellon University Pittsburgh PA School of Computer Science. 1995.

[3] Van Den Berg, Joachim., and Jacobs, Bart. “The LOOP compiler for Java and JML.” *Tools and Algorithms for the Construction and Analysis of Systems.* (2001): 299-312.

[4] Leroy, Xavier., “Formal certification of a compiler back-end or: programming a compiler with a proof assistant.” *ACM SIGPLAN Notices*. ACM. Vol. 41, No. 1, (2006, January): 42-54.

[5] Leroy, Xavier. “Formal verification of a realistic compiler.” *Communications of the ACM*, *52*.7 (2009): 107-115.

[6] Blazy, Sandrine., Dargaye, Zaynah., and Leroy, Xavier. “Formal verification of a C compiler front-end.” *FM 2006: Formal Methods*, (2006): 460-475.

[7] Berghofer, Stefan., Nipkow, Tobias., Urban, Christian., and Wenzel, Makarius. “Theorem proving in higher order logics: 22nd international conference.” *TPHOLs 2009, Munich, Germany, proceedings*. Springer. (August 17-20, 2009).

[8] Glew, Neal, and Greg Morrisett. "Type-safe linking and modular assembly language." In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 250-261. ACM, 1999.

[9] Walker, David., Crary, Karl., and Morrisett, Greg. “Typed memory management via static capabilities.” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *22*.4 (2001): 701-771.

[10] Krauss, A. (2015, November 20). Programming Concepts: Static vs. Dynamic Type Checking. Retrieved August 29, 2017, from https://thesocietea.org/2015/11/programming-concepts-static-vs-dynamic-type-checking/

[11] Khoo, Yit Phang., Chang, Bor-Yuh Evan., and Foster, Jeffrey S. Mixing type checking and symbolic execution. In *ACM Sigplan Notices*. ACM. Vol. 45, No. 6 (2010, June): 436-447.

[12] Pitts Andrew, M. (2015). Lecture Notes on Types for Part II of the Computer Science Tripos. Retrieved September 14, 2017, from https://www.cl.cam.ac.uk/teaching/0910/Types/typ.pdf.

[13] Ni, Zhaozhong, and Shao, Zhong. *A translation from typed assembly languages to certified assembly programming*. Technical report, Dept. of Computer Science, Yale Univ., New Haven, CT. 2005.

[14] Winwood, Simon., and Chakravarty, Manuel. Singleton: "A general-purpose dependently-typed assembly language." *Proceedings of the 7th ACM SIGPLAN workshop on Types in language design and implementation*. ACM. (2011, January): 3-14.

[15] Ahmed, Amal., Appel, Andrew W., Richards, Christopher D., Swadi, Kedar N., Tan, Gang., and Wang, Daniel C. (2010). "Semantic foundations for typed assembly languages." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *32*.3 (2010): 7.

[16] Yang, Jean., and Hawblitzel, Chris. "Safe to the last instruction: automated verification of a type-safe operating system." In *ACM Sigplan Notices*. ACM. Vol. 45, No. 6, (2010, June): pp. 99-110).

[17] Haas, Andreas., Rossberg, Andreas., Schuff, Derek L., Titzer, Ben L., Holman, Michael., Gohman, Dan., Wagner, Luke., Zakai, Alon, and Bastien, J. F. "Bringing the web up to speed with WebAssembly." *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* ACM. (2017, June): 185-200.

[18] Chen, Juan., Wu, Dinghao., Appel, Appel. W., and Fang, Hai. A provably sound TAL for back-end optimization. *ACM SIGPLAN Notices* ACM. Vol. 38, No. 5, (2003, June): 208-219.

[19] Xi, Hongwei., and Harper, Robert. "A dependently typed assembly language." *ACM SIGPLAN Notices*, *36*.10 (2001): 169-180.

[20] Patterson, Daniel., Perconti, Jamie., Dimoulas, Christos., and Ahmed, Amal. "FunTAL: Reasonably mixing a functional language with assembly." *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Barcelona, Spain*. (2017, June).

[21] Larmuseau, Adriaan., Patrignani, Marco., and Clarke, Dave. "Implementing a secure abstract machine." *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM. (2016, April): 2041-2048.

[22] Crary, K., Glew, Neal., Grossman, Dan., Samuels, Richard., Smith, F., Walker, D., and Zdancewic, S. "TALx86: A realistic typed assembly language." *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA* (1999, May): 25-35.

[23] Morrisett, Greg., Walker, David., Crary, Karl., and Glew, Neal. "From System F to typed assembly language." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *21*(3), (1999): 527-568.

[24] Walker, D. (2001, October 11). An Introduction to Typed Assembly Language. Retrieved September 14, 2017, from http://www.cs.cmu.edu/~dpw/papers/tal-intro.ps.

[25] Grossman, Dan., and Morrisett, Greg. Scalable certification for typed assembly language. In *International Workshop on Types in Compilation*. Springer, Berlin, Heidelberg. (2000, September): 117-145.

[26] Morrisett, Greg., Crary, Karl., Glew, Neal., and Walker, David. "Stack-based typed assembly language. In *International Workshop on Types in Compilation*." Springer, Berlin, Heidelberg. (1998, March): 28-52.

[27] Tarditi, David., Morrisett, Greg., Cheng, Perry., Stone, Chris., Harper, Robert., and Lee, Peter. "TIL: A Type-Directed Optimizing Compiler for ML." Retrieved September 14, 2017, from https://www.cs.cmu.edu/~rwh/papers/til/pldi96.pdf.

[28] Siewiorek, Daniel. P., Bell, Gordon., and Newell, Allen. C. "Computer Structures: principles and examples." McGraw-Hill, Inc..1982.

[29] Vitousek, Michael. M., Swords, Cameron., and Siek, Jeremy. G. "Big types in little runtime: open-world soundness and collaborative blame for gradual type systems." *ACM SIGPLAN Notices, 52*.1 (2017): 762-774.

[30] Walker, David. "A type system for expressive security policies." In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. (2000, January): 254-267.

[31] Appel, Andrew. W., and Felten, Edward W. "Models for security policies in proof-carrying code." *Technical Report TR-636-01*. 2001.

[32] Barthe, Gilles., Pavlova, Mariela., and Schneider, Gerardo. Precise analysis of memory consumption using program logics. SEFM'05.2005.

[33] Necula, George. C. "Proof-carrying code. design and implementation." *Proof and system-reliability. Springer Netherlands.* (2002): 261-288.

[34] Necula, George. C. “Proof-carrying code.” *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM. (1997, January): 106-119.

[35] “StrongARM“ SA-1110 Microprocessor Developers Manual.” PDF.” *Intel Corporation*. 2000.

[36] ARM, A. “Architecture reference manual. armv7-a and armv7-r edition. “ *ARM DDI C*, . (2012): *406*.

[37] Knaggs, Peter., and Welsh, Stephen. “ARM: Assembly Language Programming.” 2004.

[38] Skazikis, I. “The ARM Processor Architecture.” Retrieved August 26, 2017, from https://homepages.thm.de/~hg10013/Lehre/MMS/WS0304_SS04/Ioannis/index.htm. (2004, February 3).

[39] ARM. “ARM architecture reference manual.” *ARM*. (1996-1998, 2000, 2004, 2005).

[40] “ARM Architecture Overview.” Retrieved August 28, 2017, from https://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARM_Architecture_Overview.pdf.

[41] “The ARM Architecture.” Retrieved August 28, 2017, from http://www.bing.com/cr?IG=123193DBA0DA4830A7C23572AC71A16F&CID=06C3B2858F2A666F1338B8698E2C67F3&rd=1&h=dFrGBrJc153ddwUm3QKnM0uup9rTA-9Fb8pdtN96fLs&v=1&r=http%3a%2f%2fwww.arm.com%2ffiles%2fpdf%2fARM_Arch_A8.pdf&p=DevEx,5062.1.

[42] Ousterhout, J. K., Levy, J. Y., and Welch, B. B. “The safe-tcl security model. In Mobile Agents and Security.” *Springer Berlin Heidelberg*. (1998): 217-234.

[43] Crandall, J. R., and Chong, F. T. Minos: “Control data attack prevention orthogonal to memory model.” *Microarchitecture. MICRO-37 2004. 37th International Symposium* IEEE. (204): 221-232.

[44] Crandall, J. R., and F. T. ChongMinos. "Control data attack prevention orthogonal to memory model."

[45] Ames Jr, Stanley R., Morrie, Gasser., and Schell, Roger R. "Security Kernel Design and Implementation: An Introduction." *IEEE computer* 16.7 (1983): 14-22.

[46] Erlingsson, Úlfar. "Low-level software security: Attacks and defenses." *FOSAD*. pp. 92-134, Vol. 7. 2007.

[47] Shen, Xiaowei and Rudolph, Larry. “Commit-reconcile and fences (CRF): a new memory model for architects and compiler writers.“ *ACM SIGARCH Computer Architecture News*, *27* (2):150-161.

[48] Fournet, Cédric., Le Guernic, Gurvan., and Rezk, Tamara. “A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms.” *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. (2009): 432-441.

[49] Leroy, Xavier and Blazy, Sandrine. “Formal verification of a C-like memory model and its uses for verifying program transformations.” *Journal of Automated Reasoning*, *4*1.1 (2008): 1-31.

[50] Mowry, Todd. C., Lam, Monica. S., and Gupta, Anoop. “Design and evaluation of a compiler algorithm for prefetching.” *ACM Sigplan Notices* Vol. 27, No. 9. (1992, September): 62-73.

[51] Clarke, Edmund. M., Emerson, E. Allen., and Sifakis, Joseph. “Model checking: algorithmic verification and debugging.” *Communications of the ACM*, *52*.1 (2009):74-84.

[52] Newburn, Chris. J., So, Byoungro., Liu, Zhenying., McCool, Michael., Ghuloum, Anwar., Toit, Stefanus, D., and Guo, P. “Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language.” *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (pp. 224-235). IEEE Computer Society. (2011, April).

[53] Suffield, Andrew. “Bounds Checking for C and C++.” *BEng dissertation, Imperial College London*. (2003).

[54] Abrams, Philip Samuel. “*An APL machine”* (Doctoral dissertation, Stanford University). (1970).

[55] Li, Lian, Lin Gao, and Jingling Xue. “Memory coloring: A compiler approach for scratchpad memory management.” *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference.* IEEE. (2005, September): pp. 329-338.

[56] Pizlo, Filip., Fox, Jason. M., Holmes, David., and Vitek, Jan. “Real-time Java scoped memory: design patterns and semantics.” *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium* IEEE. (2004, May): (pp. 101-110).

[57] Shen, Xiaowei, and Rudolph, Larry. “Commit-reconcile and fences (CRF): a new memory model for architects and compiler writers.” *ACM SIGARCH Computer Architecture News*, *27*.2 (1999): 150-161.

[58] Cooper, Keith D., and Harvey, Timothy J.. “Compiler-controlled memory.” *ACM SIGOPS Operating Systems Review*. ACM. Vol. 32, No. 5, (1998, October): pp. 2-11.

[59] Shen, Xiaowei, and Larry Rudolph. “Commit-reconcile and fences (CRF): a new memory model for architects and compiler writers.” *ACM SIGARCH Computer Architecture News*, 27.2: (1999): 150-161.

[60] Duflot, Loïc, Etiemble, Daniel., and Grumelard, Olivier. "Using CPU system management mode to circumvent operating system security functions." *CanSecWest/core06* (2006).

[61] Lochbihler, A. “Making the Java memory model safe. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. Karlsruhe Institute of Technology. *35*.4 (2013).”

[62] Wang, David, Ganesh, Brinda, Tuaycharoen, Nuengwong, Baynes, Kathleen, Jaleel, Aamer, and Jacob, Bruce. “DRAMsim: a memory system simulator.” *ACM SIGARCH Computer Architecture News*, *33*.4 (2005): 100-107.

[63] Veen, Arthur H. “Dataflow machine architecture.” *ACM Computing Surveys (CSUR)*, *18*.4 (1986): 365-396.

[64] Pessl, Peter., Gruss, Daniel., Maurice, Clémentine., Schwarz, Michael., Mangard, Stefan. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." *USENIX Security Symposium*. 2016.

[65] Schumacher, Toni. “Static vs. Dynamic Typing.” *Concepts of Programming Languages–CoPL’15*, 22.

[66] Abadi, Martín, Cardelli, Luca., Pierce, Benjamin., and Plotkin, Gordon. “Dynamic typing in a statically typed language.” *ACM transactions on programming languages and systems (TOPLAS)*, *13*.2 (1991): 237-268.

[67] Abadi, Martín., Cardelli, Luca., Pierce, Benjamin., and Rémy, D. “Dynamic typing in polymorphic languages.” *Journal of functional programming*, *5*.1 (1995):111-130.

[68] Vitousek, Michael M., Kent, Andrew M., Siek, Jeremy G., and Baker, Jim. “Design and evaluation of gradual typing for Python.” *ACM SIGPLAN Notices*. Vol. 50. No. 2. ACM, (2014): 45-56.

[69] Henglein, Fritz. "Dynamic typing: Syntax and proof theory." *Science of Computer Programming* 22.3 (1994): 197-230.

[70] Cohen, Ernie, Michał Moskal, Stephan Tobies, and Wolfram Schulte. “A precise yet efficient memory model for C.” *Electronic Notes in Theoretical Computer Science*, *254* (2009): 85-103.

[71] Riley, H. Norton. "The von Neumann architecture of computer systems." Computer Science Department, California State Polytechnic University (1987).

[72] Burks, Arthur W., Herman H. Goldstine, and John Von Neumann. "Preliminary discussion of the logical design of an electronic computing instrument." *The Origins of Digital Computers*. Springer Berlin Heidelberg, (1982): 399-413.

[73] Schulte, Eric. "Non Von Neumann Computation (a survey)." (2010).

[74] Von Neumann, John. "First draft of a report on the EDVAC." *Contract No. W-670-ORD-492, Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia. Reprinted (in part) in Randell,* Brian (1982).

[75] Heath, F. G. "Entering the non-von-neumann era." *IEE Journal on Computers and Digital Techniques* 2.2 (1979): 57-58.

[76] Burks, Arthur W., Herman H. Goldstine, and John Von Neumann. "Preliminary discussion of the logical design of an electronic computing instrument." *The Origins of Digital Computers*. Springer Berlin Heidelberg, (1982): 399-413.

[77] "1977 ACM Turing Award Lecture." *a1977-Backus.pdf*, Communications August 1978 of Volume 21 the ACM. http://delivery.acm.org/10.1145/1290000/1283933/a1977-backus.pdf?ip=47.204.13.10&id=1283933&acc=OPEN&key=4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E6D218144511F3437&CFID=810412879&CFTOKEN=67409079&__acm__=1505679245_02303cb9b8876edb16d55dc159c81c7f . Accessed 17 Sept. 2017.

[78] Alpern, B., Carter, L., Feig, E., and Selker, T. "The uniform memory hierarchy model of computation." *Algorithmica*, *12*.2 (1994):72-109.

[79] Bensoussan, A., Clingen, C. T., and Daley, R. C. "The Multics virtual memory: concepts and design." *Communications of the ACM*, *1*.5 (1972): 308-318.

[80] Diatchki, I. S., and Jones, M. P. "Strongly typed memory areas." *Proceedings of ACM SIGPLAN 2006 Haskell Workshop* (2006): 72-83.

[81] Alglave, Jade., Kroening, Daniel., Lugton, John., Nimal, Vincent ., and Tautschnig, M. "Soundness of Data Flow Analyses for Weak Memory Models." *APLAS* (2011, December): pp. 272-288.

[82] "Physical Memory Study Guide." Retrieved September 17, 2017, from http://www.scsd.k12.wa.us/wrms/Info_tech/PHYSICAL_MEMORY_SG.pdf.

[83] Culler, D. E. "Dataflow architectures." *Annual review of computer science*, *1*,1, (1986): 225-253.

[84] Owens, S., Sarkar, S., and Sewell, P. “A better x86 memory model: x86-TSO.” In *International Conference on Theorem Proving in Higher Order Logics* (pp. 391-407). Springer, Berlin, Heidelberg. (2009, August).

[85] Owens, Scott, Sarkar , Susmit, and Sewell, Peter. ” Retrieved September 17, 2017, from https://herbsutter.com/2012/08/02/strong-and-weak-hardware-memory-models/.(2012, August)

[86] Sewell, Peter., Sarkar, Susmit., Owens, Scott., Nardelli, Francesco Zappa., and Myreen, Magnus O. " x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors." *Communications of the ACM* 53.7 (2010): 89-97.

[87] Edwards, James, Frederick Strahm, John Richardson, and Ylian Saint-Hilaire. “Countering Buffer Overrun Security Vulnerabilities in a CPU.” Patent 20020144141. 3 October 2002.

[88] Sederlund, Edward R., Lindesmith, Robert J., Root, Larry A., Dupree, Wayne P., Thomas, Lowell V. “Extended Harvard Architecture Computer Memory System with Programmable Variable Address Increment.” Patent 5,555,424. (10 September 1996).

[89] Anthony I. Stansfield. “Programmable Logic Device with Memory that Can Store Routing Data of Logic Data.” Patent 5,473,267. 5 December 1995.

[90] Herman, Morton B. “Software Security Method Using Partial Fabrication of Proprietary Control Word Decoders and Microinstruction Memories.” Patent 4,513,174. 23 April 1985.

[91] Siek, Jeremy G., et al. “Refined Criteria for Gradual Typing.” *LIPIcs - Leibniz International Proceedings in Informatics*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, 2015, drops.dagstuhl.de/opus/volltexte/2015/5031/.

[92] Ligatti, Jay. “Definition of DISM (Diminished Instruction Set Machine).” DISM-definition.pdf [Pdf]. Tampa: CSE at University of South Florida. (2015, August 21).

[93] Hernandez, Ivory. “Definition of *Strong*-DISM.” Tampa: CSE at University of South Florida. (2017).

[94] Hernandez, Ivory. “Definition of *Strong_MEM*-DISM.” Tampa: CSE at University of South Florida. (2017).

[95] Fischer, Patrick C. "Turing machines with restricted memory access." *Information and Control* 9.4 (1966): 364-379.

[96] Govindavajhala, Sudhakar, and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. 2003 IEEE Symposium on Security and Privacy, 11 May 2003, www.bing.com/cr?IG=DD4D1608C52A4DE09804E12B9AE7BFD1&CID=0278457AC31C6C41012F4E5FC21A6D16&rd=1&h=l4PZwunEfW8TgXnoNHXXuFUXFq-EyVmpEPur0zbwxsE&v=1&r=https%3a%2f%2fwww.cs.princeton.edu%2f%7eappel%2fpapers%2fmemerr.pdf&p=DevEx,5063.1.

ABOUT THE AUTHOR

Ivory Hernandez graduated with a dual Bachelor of Science in Computer Science (2012) and Computer Engineering (2013) from the University of South Florida (USF) where he is expected to complete this semester (Fall 2017) his Master of Science in Computer Science. Previous to study computer sciences he worked as a visual artist, after having obtained a B.A. degree from San Alejandro Academy of Fine Arts. He has lived in Tampa since 2001. Two of his principles are “Practice Makes Perfect” and “With rush I get faster, without: farther”.