SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

Latent Behaviour Analysis

Submitted by

Eric G. ROTHSTEIN MORRIS

Thesis Advisor

Dr. Sudipta CHATTOPADHYAY

Information Systems Technology and Design (ISTD)

A thesis submitted to the Singapore University of Technology and Design in fulfillment of the requirement for the degree of Doctor of Philosophy

2021

PhD Thesis Examination Committee

TEC Chair:	Prof. XXXX
Main Advisor:	Prof. XXXX
Co-advisor(s):	Prof. XXXX (if any)
Internal TEC member 1:	Prof. XXXX
Internal TEC member 2:	Prof. XXXX
External TEC member 1:	Prof. XXXX (optional)
External TEC member 2:	Prof. XXXX (optional)

Abstract

Information Systems Technology and Design (ISTD)

Doctor of Philosophy

Latent Behaviour Analysis

by Eric G. ROTHSTEIN MORRIS

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too…

Publications

Journal Papers, Conference Presentations, etc...

Acknowledgements

The acknowledgments and the people to thank go here, don’t forget to include your project advisor…

Contents

PhD Thesis Examination Committee **i**

Abstract **ii**

Publications **iii**

Acknowledgements **iv**

1 Introduction **1**
- 1.1 Story Time 1
  - 1.1.1 Subsection 1 1
  - 1.1.2 Subsection 2 1
- 1.2 Main Section 2 2

2 Preliminaries and Notation **3**
- 2.1 Universal (Co)Algebra 3
  - 2.1.1 Categories, Functors and Monads 3
  - 2.1.2 $\mathbb{T}$-algebras, F-coalgebras and λ-bialgebras 3
  - 2.1.3 Catamorphisms and Anamorphisms 3
- 2.2 Spectre 3
  - 2.2.1 Spectre V1 4
  - 2.2.2 Spectre V2 4
- 2.3 Cyber-Physical System 4

3 Latent Behaviours **5**
- 3.1 Introduction 5
- 3.2 Latent Behaviour of a Bialgebra 5

4 Side-Channel Repair **6**
- 4.1 General Idea 6
- 4.2 Related work and how do we distinguish from them 6
- 4.3 CTP (Cryptocoding) Axioms, and our solution 6
- 4.4 Formalisation 8
  - 4.4.1 Language-Based Semantics CTP 9
- 4.5 The IMP Language 11
- 4.6 Memory Footprint for the IMP Language 11
  - 4.6.1 Secure Constant Memory 11
- 4.7 The IMP Language with Division 12
  - 4.7.1 Examples of Side Channel Attacks 12
  - 4.7.2 Transformations Rules for CTP in IMP 12

4.8 Taint Analysis . . . 14
4.9 The Enforcement Algorithm . . . 14
4.9.1 Premises Behind the Enforcement Algorithm . . . 14
4.9.2 An LLVM Primer . . . 15
4.9.3 The `Linearise` Algorithm . . . 15
4.9.4 A Premise for Efficient Repaired Code . . . 15
4.9.5 Enforcing Safe Behaviour . . . 17
4.9.6 A GKAT to Correctly Unwind Basic Block Loops . . . 18
4.9.7 Types . . . 18
4.9.8 Instructions . . . 19
4.9.9 Central idea . . . 19
4.9.10 Subsection 1 . . . 19
4.9.11 Subsection 2 . . . 19
4.10 Main Section 2 . . . 20

5 First chapter $\Sigma_{\omega}^{i}\infty$ 21
5.1 Welcome and Thank You . . . 21
5.2 Learning LaTeX . . . 21
5.2.1 A (not so short) Introduction to LaTeX . . . 21
5.2.2 A Short Math Guide for LaTeX . . . 22
5.2.3 Common LaTeX Math Symbols . . . 22
5.2.4 LaTeX on a Mac . . . 22
5.3 Getting Started with this Template . . . 22
5.3.1 About this Template . . . 23
5.4 What this Template Includes . . . 23
5.4.1 Folders . . . 23
5.4.2 Files . . . 24
5.5 Filling in Your Information in the **`main.tex`** File . . . 25
5.6 The `main.tex` File Explained . . . 25
5.7 Thesis Features and Conventions . . . 26
5.7.1 Printing Format . . . 26
5.7.2 Using US Letter Paper . . . 26
5.7.3 References . . . 27
A Note on bibtex . . . 27
5.7.4 Tables . . . 28
5.7.5 Figures . . . 28
5.7.6 Typesetting mathematics . . . 29
5.8 Sectioning and Subsectioning . . . 30
5.9 In Closing . . . 30

A Appendix Title Here 31

Bibliography 32

List of Figures

2.1 Multiplication square. 3
2.2 Unit triangle. 3

5.1 An Electron 29

List of Tables

5.1 The effects of treatments X and Y on the four groups studied. 28

For/Dedicated to/To my…

Chapter 1

Introduction

1.1 Story Time

We model systems based on what they are supposed to do, but what can we say about the things that they *could* do?

Actually, the thesis would be super interesting if potential behaviour problems are presented in terms of a game between the system and the attacker, with the attacker following some sort of reactive strategy, and the system as well. At some point, one or the other wins because it is a deterministic game if the system is deterministic.

How can we model attackers? How can we model their actions? How can we prepare against them?

Many attacks are discovered through testing: an experienced hacker creates a proof of concept to illustrate how some system may be attacked. Then, an abstraction is created to explain why the attack works, and to suggest countermeasures against it.

You can probably show several papers that display this pattern.

Fuzzing is probably one of the most popular techniques for system transformations.

1.1.1 Subsection 1

Nunc posuere quam at lectus tristique eu ultrices augue venenatis. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Aliquam erat volutpat. Vivamus sodales tortor eget quam adipiscing in vulputate ante ullamcorper. Sed eros ante, lacinia et sollicitudin et, aliquam sit amet augue. In hac habitasse platea dictumst.

1.1.2 Subsection 2

Morbi rutrum odio eget arcu adipiscing sodales. Aenean et purus a est pulvinar pellentesque. Cras in elit neque, quis varius elit. Phasellus fringilla, nibh eu tempus venenatis, dolor elit posuere quam, quis adipiscing urna leo nec orci. Sed nec nulla auctor odio aliquet consequat. Ut nec nulla in ante ullamcorper aliquam at sed dolor. Phasellus fermentum magna in augue gravida cursus. Cras sed pretium lorem. Pellentesque eget ornare odio. Proin accumsan, massa viverra cursus pharetra, ipsum nisi lobortis velit, a malesuada dolor lorem eu neque.

1.2 Main Section 2

Sed ullamcorper quam eu nisl interdum at interdum enim egestas. Aliquam placerat justo sed lectus lobortis ut porta nisl porttitor. Vestibulum mi dolor, lacinia molestie gravida at, tempus vitae ligula. Donec eget quam sapien, in viverra eros. Donec pellentesque justo a massa fringilla non vestibulum metus vestibulum. Vestibulum in orci quis felis tempor lacinia. Vivamus ornare ultrices facilisis. Ut hendrerit volutpat vulputate. Morbi condimentum venenatis augue, id porta ipsum vulputate in. Curabitur luctus tempus justo. Vestibulum risus lectus, adipiscing nec condimentum quis, condimentum nec nisl. Aliquam dictum sagittis velit sed iaculis. Morbi tristique augue sit amet nulla pulvinar id facilisis ligula mollis. Nam elit libero, tincidunt ut aliquam at, molestie in quam. Aenean rhoncus vehicula hendrerit.

Chapter 2

Preliminaries and Notation

This first page should have an index from concepts to the page where they are explained/defined. There should be a bit of redundancy with the glossary at the end, but I feel it makes the thesis easier to navigate: search preliminaries, and you're there. THIS SECTION HAS CONTENT THAT I DID NOT DEVELOP. The newer content comes in the following chapters

2.1 Universal (Co)Algebra

Make sure you create a wonderful introduction to the world of (co)algebras and explain why you want to use them. They are a wonderful way to unify the formalism of the thesis.

2.1.1 Categories, Functors and Monads

2.1.2 $\mathbb{T}$-algebras, F-coalgebras and λ-bialgebras

A $\mathbb{T}$-*algebra* for a monad $\mathbb{T} = (T, \eta, \mu)$ is a pair $(X, \mathbb{T}(X) \xrightarrow{a} X)$ of an object X and a morphism $a \colon \mathbb{T}(X) \to X$ which, due to the relationship between monads and adjunctions, satisfy two properties: the *multiplication square* and the *unit triangle*, shown in Figures 2.1 and 2.2.

$$\begin{tikzcd}
T^2(X) \arrow[r, "T(a)"] \arrow[d, "\mu_X"'] & T(X) \arrow[d, "a"] \\
T(X) \arrow[r, "a"] & X
\end{tikzcd}$$

FIGURE 2.1: Multiplication square.

$$\begin{tikzcd}
T(X) \arrow[r, "a"] \arrow[dr, "1_{T(X)}"'] & X \arrow[d, "\eta"] \\
 & T(X)
\end{tikzcd}$$

FIGURE 2.2: Unit triangle.

2.1.3 Catamorphisms and Anamorphisms

2.2 Spectre

And Meltdown?

2.2.1 Spectre V1

2.2.2 Spectre V2

2.3 Cyber-Physical System

This section probably makes more sense in its own chapter, when we explain what the latent behaviour of a CPS is.

Chapter 3

Latent Behaviours

Find a suitable quote?

If you only do what you can do, you will never be more than what you are now.

3.1 Introduction

Give an interesting motivational example: What is the notion of latent behaviour? Why do we care about them? How is it useful to study them?

3.2 Latent Behaviour of a Bialgebra

Chapter 4

Side-Channel Repair

4.1 General Idea

In the following, whenever we write *program*, we mean *non-concurrent program* The premises of our work are the following:

- Every program can be linearised using a set of *predication axioms*.

4.2 Related work and how do we distinguish from them

- Ashay Rane: we do not use *cmove* as a primitive instruction, since it may be vulnerable to spectre attacks

prove this?

4.3 CTP (Cryptocoding) Axioms, and our solution

Taken from **cryptocoding**

- Compare secrets in constant time.
  - We implement a constant-time comparison function for each type at the LLVM IR. Do note that there are several comparison operations: leq, geq, neq, etc. Ideally, we have a way to implement them at the IR level, but this might not be possible; in other words, there may be the need to leave these as primitive operations that are translated differently for the different architectures. This latter approach has its advantages though: there is room for optimization and we can ensure that it, in fact, gets translated to a sequence of constant time instructions; however, it requires more muscle, i.e., we need to be super careful on how we translate them so that they are, in fact, constant time.
- Avoid branchings controlled by secret data
  - The suggested solution:

    > Timing leaks may be mitigated by introducing dummy operations in branches of the program in order to ensure a constant execution time. It is however more reliable to avoid branchings altogether, for example by implementing the conditional operation as a straight-line program. To select between two inputs a and b depending on a selection bit (a la CTSelect).

  - We implement predication axioms that allow us to transform control structures into arithmetic expressions using Shannon expansions.

- Avoid table look-ups indexed by secret data
  - The suggested solution:

    Replace table look-up with sequences of constant-time logical operations, for example by bitslicing look-ups (as used in NaCl's implementation of AES-CTR, or in Serpent. For AES, constant-time non-bitsliced implementations are also possible, but are much slower. (check blog for resources on how to do this).
  - The two available options is to somehow implement a rudimentary version of ORAM or make lookups on a set generated by the secret (which should probably be similar to what bit slicing is).

- Avoid secret-dependent loop bounds
  - The suggested solution:

    Make sure that all loops are bounded by a constant (or at least a non-secret variable). In particular, make sure, as far as possible, that loop bounds and their potential underflow or overflow are independent of user-controlled input (you may have heard of the Heartbleed bug).
  - Our solution: each secret has an associated iteration bound (i.e., loops that depend on that secret will be unwinded that number amount of times).

- Prevent compiler interference with security-critical operations (i.e., compilers can mess up your patches)
  - The suggested solution:

    Look at the assembly code produced and check that all instructions are there. (This will not be possible for typical application sizes, but should be considered for security-sensitive code.)
    Know what optimizations your compiler can do, and carefully consider the effect of each one on security programming patterns. In particular, be careful of optimizations that can remove code or branches, and code that prevents errors which "should be impossible" if the rest of the program is correct.
    When possible, consider disabling compiler optimizations that can eliminate or weaken security checks.
    Note that such workarounds may not be sufficient and can still be optimized out.
  - Our solution: we suggest that CTFixes happen as latter as possible in the compilation chain. We can only offer guarantees at the LLVM IR level, the rest is target specific (this is acceptable, since the repair paper works at the same level).

- Prevent confusion between secure and insecure APIs

- Avoid mixing security and abstraction levels of cryptographic primitives in the same API layer
- Use unsigned bytes to represent binary data

  > Some languages in the C family have separate signed and unsigned integer types. For C in particular, the signedness of the type char is implementation-defined. This can lead to problematic code.

  - Their suggestion:

    > In languages with signed and unsigned byte types, implementations should always use the unsigned byte type to represent bytestrings in their APIs.

  - our solution: LLVM IR does uses both signed and unsigned. We need an axioms to only yield unsigned byte types.
- Clean memory of secret data
  - Our suggestion: although this might take a lot of time, it is a good idea to reset all input secrets and their intermediate operations.
- Use strong randomness
  - Our suggestion: we cannot really enforce this. We assume it. (TBH I dunno how does randomness work on the LLVM IR).
- Always typecast shifted values.

read blog

Do note also the following:

- Casting operations at higher level may be dangerous: they might introduce branching at the lower level for some architectures (there is an example where a _Bool is casted into an uint_32 in the blog.

4.4 Formalisation

We use Guarded Kleene Algebra with Tests (GKAT) **GKAT** to define our transformation rules. the GKAT syntax is as follows:

$$
\begin{array}{lll}
\multicolumn{3}{l}{b, c, d, \in \text{BExp} ::=} \\
\mid & 0 & \textbf{False} \\
\mid & 1 & \textbf{True} \\
\mid & t \in T & t \\
\mid & b \cdot c & b \textbf{ and } c \\
\mid & b + c & b \textbf{ or } c \\
\mid & \bar{b} & \textbf{not } b
\end{array}
\qquad
\begin{array}{lll}
\multicolumn{3}{l}{e, f, g, \in \text{Exp} ::=} \\
\mid & p \in \Sigma & \textbf{do } p \\
\mid & b \in \text{BExp} & \textbf{assert } b \\
\mid & e \cdot f & e;f \\
\mid & f +_b g & \textbf{if } b \textbf{ then } f \textbf{ else } g \\
\mid & e^{(b)} & \textbf{while } b \textbf{ do } e
\end{array}
$$

We take a layered approach to constant time programming (CTP); initially, we consider actions $a \in \Sigma$ to be abstract (i.e., only symbols), then, we give semantics to actions by mapping them to a set of instructions in LLVM IR. This layered approach allows us to exclusively focus on program structure, and how branching and iteration could introduce side channels.

4.4.1 Language-Based Semantics CTP

The language-based semantics of a GKAT expression e corresponds to a language of *guarded strings* **GKAT** Informally, a guarded string is an interleaved sequence of *(logical) atoms* and actions, which model how actions change the state of the system. Informally, an *atom* is a valuation of all tests in the set of tests T (e.g., if $T = \{t_1, t_2\}$, the expressions $\overline{t_1} \wedge \overline{t_2}$, $\overline{t_1} \wedge t_2$, $t_1 \wedge \overline{t_2}$, and $t_1 \wedge t_2$ are the atoms). Formally, an atom is a non-zero minimal element in the free boolean algebra in T **KAT** We denote atoms by α, β, and γ, and the set of atoms by At. The formal definition of a guarded string is as follows.

Definition 4.4.1 (Guarded String)**.** Given a set of actions Σ and a set of tests T, a *guarded string* g is an element of the set $\mathrm{GS} := \mathrm{At} \cdot (\Sigma \cdot \mathrm{At})^*$.

Here is where you explain what the semantics of GKAT expressions are in terms of guarded strings, and you say that infinite loops yield an empty language, which is super weird, right?

Definition 4.4.2 (Resource Metrics and Resource Consumption)**.** A *resource metric* is a map $m\colon \mathrm{GS} \to \Sigma \to \mathbb{R}^+$. Given a resource metric m and a guarded string g, the value of $m(g)(a)$ is the resource consumption for the action a after executing all the actions in g. The *resource consumption* of g, denoted $\mathrm{RC}(g)$, is the accumulated resource consumption of its actions; formally,

$$\mathrm{RC}(g) \triangleq \begin{cases} 0, & \text{if } g \in \mathrm{At}; \\ \mathrm{RC}(h) + m(h)(a), & \text{if } g = h \cdot a \cdot \alpha; \end{cases} \tag{4.1}$$

Henceforth, we will assume that resource metrics are history-independent, i.e., resource consumption is independent of previous actions or initial conditions, meaning that resource metrics are instead maps of type $\Sigma \to \mathbb{R}^+$.

Are there relevant cases where this is not true? Answer: actually, the memory footprint may probably be one of these cases (more precisely, memory footprint should also consider the interaction environment in shared caches... it can get really complicated if we want to model everything...)

Definition 4.4.3 (Weak Constant Time)**.** Given a language of guarded strings $L \subseteq \mathrm{GS}$, we say that L has *weak constant time* if and only if, for all $g_1, g_2 \in L$, $\mathrm{RC}(g_1) = \mathrm{RC}(g_2)$.

We also define a strong notion of constant time to study more modular attackers.

Definition 4.4.4 (Strong Constant Time)**.** Given a language of guarded strings $L \subseteq \mathrm{GS}$, we say that two guarded strings $g_1, g_2 \in L$ are *strong constant time equivalent*, denoted

$g_1 \equiv_{\mathbb{X}} g_2$, if and only if,

$$g_1 \equiv_{\mathbb{X}} g_2 \triangleq \begin{cases} \textbf{true}, & \text{if } g_1 \in \text{At and } g_2 \in \text{At}, \\ g_1' \equiv_{\mathbb{X}} g_2' \wedge m(g_1')(a_1) = m(g_2')(a_2), & \text{if } g_i = g_i' \cdot a_i \cdot \alpha_i \text{ for } i = 1, 2; \end{cases} \tag{4.2}$$

We say that L has *strong constant time* if and only if, for all $g_1, g_2 \in L$, $g_1 \equiv_{\mathbb{X}} g_2$.

To obtain the notion of constant time that is used for security analysis, we need to match traces that are only equal on their public variables. Atoms themselves may match even when public values are different (e.g. if p is public, the test $p \leq 128$ is satisfied by any public value below 128); thus, we need to make the state of public variables explicit. For such purposes, let $\mathscr{V}$ be the set of variables, and let $[\![\mathscr{V}]\!]$ be the set of valuation functions which map variables to values; we override the definition of guarded strings so that the set of guarded strings is now generated by the grammar

$$\text{GS} := ([\![\mathscr{V}]\!] \times \text{At}) \cdot (\Sigma \cdot ([\![\mathscr{V}]\!] \times \text{At}))^* . \tag{4.3}$$

Thus, guarded strings now satisfy either the pattern (Γ, α) or the pattern $(\Gamma, \alpha) \cdot a \cdot h$, where Γ is a valuation of the variables, α is an atom and h is a guarded string.

Remark 4.1. If the domains of variables are finite, this formulation reduces to the original formulation as follows: for each variable p and each of its possible values v, we include the proposition $p = v$ in the set of tests.

Definition 4.4.5 (Public Equality)**.** Whenever two valuations Γ_1 and Γ_2 are equal on their public variables, we denote it by $\Gamma_1 =_p \Gamma_2$. Two guarded strings g_1 and g_2 are equal on their public variables, denoted $g_1 =_p g_2$, if and only if their *initial* valuations are equal on public variables.

Depending on the attacker model, we might choose one of the following definitions to suit our needs best:

Definition 4.4.6 (Secure Weak Constant Time)**.** Given a language of guarded strings $L \subseteq \text{GS}$, we say that L has *secure weak constant time* guarantees if and only if, for all $g_1, g_2 \in L$:

$$g_1 =_p g_2 \Rightarrow \text{RC}(g_1) = \text{RC}(g_2). \tag{4.4}$$

Definition 4.4.7 (Secure Strong Constant Time)**.** Given a language of guarded strings $L \subseteq \text{GS}$, we say that L has *secure strong constant time* guarantees if and only if, for all $g_1, g_2 \in L$:

$$g_1 =_p g_2 \Rightarrow g_1 \equiv_{\mathbb{X}} g_2. \tag{4.5}$$

4.5 The IMP Language

We explore the simple imperative programming language with variable assignments and boolean expressions from **GKAT** IMP; the language is defined as follows:

– arithmetic expressions:	$a \in \mathscr{A} ::= x \in \mathscr{V} \mid n \in \mathbb{N} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2$
– boolean expressions:	$b \in \mathscr{B} ::= \textbf{false} \mid \textbf{true} \mid a_1 < a_2 \mid \textbf{not}\ b \mid b_1\ \textbf{and}\ b_2 \mid b_1\ \textbf{or}\ b_2$
– commands:	$c \in \mathscr{C} ::= \textbf{skip} \mid x := a \mid c_1; c_2 \mid \textbf{if}\ b\ \text{then}\ c_1\ \text{else}\ c_2 \mid \textbf{while}\ b\ \textbf{do}\ c$

As stated in **GKAT** this language can be modelled in GKAT using actions for assignments and primitive tests for comparisons as follows:

$$\Sigma = \{x := a \mid x \in \mathscr{V}, a \in \mathscr{A}\}, \quad T = \{a_1 < a_2 \mid a_1, a_2 \in \mathscr{A}\}. \tag{4.6}$$

Following **GKAT** we interpret GKAT expressions over the space of variable assignments $[\![\mathscr{V}]\!] \triangleq \mathscr{V} \to \mathbb{N}$:

$$\begin{aligned}
\texttt{eval}(x := a) &\triangleq \{(\sigma, \sigma[x := n]) \mid \sigma \in [\![\mathscr{V}]\!], n = \mathscr{A}[\![a]\!]_\sigma\}, \\
\sigma[x := n] &\triangleq \lambda y. \begin{cases} n, & \text{if } y = x; \\ \sigma(y), & \text{otherwise}, \end{cases} \\
\texttt{sat}(a_1 < a_2) &\triangleq \{\sigma \in [\![\mathscr{V}]\!] \mid \mathscr{A}[\![a_1]\!]_\sigma < \mathscr{A}[\![a_2]\!]_\sigma\},
\end{aligned}$$

where $\mathscr{A}[\![a]\!]_\sigma$ denotes the arithmetic evaluation of a in the context of σ. Finally, sequential composition, conditionals and while loops are modelled by their GKAT counterparts, and **skip** is modelled by 1.

4.6 Memory Footprint for the IMP Language

A memory footprint considers how variables are contiguously allocated in memory. To capture the notion of a memory footprint, we are going to give a dimension to variables so they now work as finite vectors indexed by integers. Given a variable $\vec{x} \in \mathscr{V}$ and an arithmetic expression $a \in \mathscr{A}$, the expression $\vec{x}[a]$ refers to the value stored in $\vec{x}$ at position a.

This treatment of variables as vectors not change the expressivity of the IMP language, and it is purely to address the fact that, for several computer architectures, when we read a memory location (e.g, $\vec{x}[a]$), its contiguous locations are also loaded into the cache (e.g., $\vec{x}[a+1]$ to $\vec{x}[a+k-1]$, where k is the cache line size).

4.6.1 Secure Constant Memory

Now that actions have a more concrete semantics, we can describe how we expect a program in IMP to interact with cache memory. We follow **Chattopadhyay** and describe caches using three main parameters: cache line size (CLS) in bytes, the number of cache sets (CS) and an associativity and replacement policy (P).

Our main challenge in defining a resource consumption function to model how actions affect the state of the cache stems from the case where caches are shared by different processes. If we want our transformed programs to be resilient to attacks like FLUSH+RELOAD, then

4.7 The IMP Language with Division

We modify the simple imperative programming language with variable assignments and boolean expressions from **GKAT** –IMP– so that it includes division; the language is defined as follows:

$$\begin{array}{rl} \text{arithmetic expressions} & a \in \mathscr{A} ::= x \in \mathscr{V} \mid n \in \mathbb{R}_\perp \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \mid a_1 \div a_2 \\ \text{boolean expressions} & b \in \mathscr{B} ::= \textbf{false} \mid \textbf{true} \mid a_1 < a_2 \mid a_1 - a_2 \mid \textbf{not}\ b \mid b_1\ \textbf{and}\ b_2 \mid b_1\ \textbf{or}\ b_2 \\ \text{commands} & c \in \mathscr{C} ::= \textbf{skip} \mid x := a \mid c_1; c_2 \mid \textbf{if}\ b\ \text{then}\ c_1\ \text{else}\ c_2 \mid \textbf{while}\ b\ \textbf{do}\ c \end{array}$$

where $\mathbb{R}_\perp = \mathbb{R} \uplus \{\perp\}$, and $\perp$ is a symbol to denote *not-a-number* (NaN).

As stated in **GKAT** this language can be modelled in GKAT using actions for assignments and primitive tests for comparisons as follows:

$$\Sigma = \{x := a \mid x \in \mathscr{V}, a \in \mathscr{A}\}, \quad T = \{a_1 < a_2 \mid a_1, a_2 \in \mathscr{A}\}. \tag{4.7}$$

Following **GKAT** we interpret GKAT expressions over the space of variable assignments $[\![\mathscr{V}]\!] \triangleq \mathscr{V} \to \mathbb{R}_\perp$:

$$\begin{aligned} \texttt{eval}(x := a) &\triangleq \{(\sigma, \sigma[x := n]) \mid \sigma \in [\![\mathscr{V}]\!], n = \mathscr{A}[\![a]\!]_\sigma\}, \\ \sigma[x := n] &\triangleq \lambda y. \begin{cases} n, & \text{if } y = x; \\ \sigma(y), & \text{otherwise,} \end{cases} \\ \texttt{sat}(a_1 < a_2) &\triangleq \{\sigma \in [\![\mathscr{V}]\!] \mid \mathscr{A}[\![a_1]\!]_\sigma < \mathscr{A}[\![a_2]\!]_\sigma\}, \end{aligned}$$

where $\mathscr{A}[\![a]\!]_\sigma$ denotes the arithmetic evaluation of a in the context of σ. Finally, sequential composition, conditionals and while loops are modelled by their GKAT counterparts, and **skip** is modelled by 1.

4.7.1 Examples of Side Channel Attacks

4.7.2 Transformations Rules for CTP in IMP

Branch balancing is a technique to enforce CTP at the language level. This technique adds dummy instructions to shorter branches so that both branches have the same resource consumption. We capture this notion with the following repair rule.

Definition 4.7.1 (Branch Balancing).

$$(x := a_1) +_b 1 \rightsquigarrow (x := a_1) +_b (x := x)$$

To balance branches, we want an instruction that has the same resource consumption as $x := a$, but acts as an identity on x, hence the choice for $x := x$. To avoid undefined behaviour, we assume that all variables are initialised by default with a value of 0 so the assignment $x := x$ is always well defined.

While branch balancing may seem like a sound solution, the compiler may remove all dummy instructions inserted since it considers them to be no-ops, taking us back to an unbalanced program at the binary level. To overcome this limitation, we rely on the correspondence between program structure and arithmetic expressions, captured by the following repair rule.

Definition 4.7.2 (Predication)**.**

$$(x := a_1) +_b (x := a_2) \rightsquigarrow x := b \otimes a_1 + (\textbf{not } b) \otimes a_2$$

where $\otimes$ is multiplication, but it interprets the boolean condition as 0 or 1 depending on whether the condition evaluates to **false** or **true**, respectively.

The predication rule forces the evaluation of all terms in both branches, so it is more resource intensive than branch balancing; however, predication addresses two of our objectives: one is to prevent the compiler from removing dummy assignments introduced during branch balancing, and the second is to remove branches that might be vulnerable to spectre attacks.

Loops have two problems: in general, they are both vulnerable to spectre attacks and timing side-channel attacks. The former vulnerability is due to the branching implicitly introduced by the loop's guard, and the latter is due to the variable number of executions of the loop's body. Both of these vulnerabilities can be addressed by a combination of branch balancing, predication and the following transformation rule.

Definition 4.7.3 (k-Truncation)**.** To k-truncate a loop means that we transform $e^{(b)}$ into a sequence of k conditional statements by unwinding the loop as follows

$$e^{(b)} \rightsquigarrow \underbrace{(e +_b 1) \cdot (e +_b 1) \ldots (e +_b 1)}_{k \text{ times}}$$

An equivalence relation between $e^{(b)}$ and its k-truncation can be obtained if k is a public over-approximation of the (secret-dependent) loop bound of $e^{(b)}$.

Compilers normally avoid this type of unwinding when combined with predication, because it would cause programs to slow down. However, in the context of CTP, this type of delays are desirable since they make loops safe in terms of resource consumption, and the lack of branching eliminates spectre vulnerabilities.

The programmer should tell us somehow how many times we unwind each loop that depends on a secret. FaCT forces loops to be of the form “for x from e to e” to fix iteration increments so they know how to unwind.

4.8 Taint Analysis

Here is an interesting problem: if we have a non-security critical section (e.g., a loop that does not depend on secret) inside a section we are applying linearisation, do we need to linearise it? Probably not! The problem is that this is not compatible with our current implementation, because the current implementation linearises everything. If we introduce breakpoints we can allow loops in the branches.

A naive definition of Taint analysis is the following: a variable is tainted if an only if, from belongs to the transitive closure of the small-step taint procedure, which marks all uses of a tainted variable as tainted. There is no untainting. We assume that, after basic code optimisations have taken place, all uses of a variable are productive an monotonous in the security hierarchy (i.e., expressions are not constant W.R.T secret inputs, and they do not inherently lose their security label). Maybe we could empower the language with annotations that override whatever taint analysis infers?

4.9 The Enforcement Algorithm

We present a k-step enforcement algorithm that enforces CTP techniques at the level of LLVM IR (compiler level) to eliminate timing side channels introduced by the use of branching and loops.

Resolve k

We implement these algorithms as LLVM passes, so it is useful to understand how code is organised in LLVM.

4.9.1 Premises Behind the Enforcement Algorithm

- Programmers need not worry about CTP at the language level, but rather delegate as much work as possible to the compiler. The compiler may ask for information about loop bounds and about which fields are secret (these can be annotations provided by the programmer). When compiling with the right flags, the compiler may optimise the code using standard optimisations, but must ultimately enforce CTP.

- The sequential composition of CTP programs is CTP. This is better understood when working with GKAT expressions: if all *actions* are CTP, then their sequential composition is CTP. Even if actions are CTP, GKAT *expressions* are not necessarily CTP; as we know, branching and iteration do not intrinsically satisfy CTP.

- The attacker model is a rather strong in the sense that it can measure resource consumption of prefix executions and not only total executions, which is equivalent to an attacker that can measure the resource consumption of each individual action.

- The enforcement algorithm is sound: a repaired version $R[P]$ of program P is functionally equivalent to P in terms of input and output, but $R[P]$ satisfies the secure constant time property.

- The enforcement algorithm is probably not transparent: if a program P satisfies CTP, its repaired version $R[P]$ will also satisfy CTP, but $R[P]$ and P may differ in terms of resource consumption, with $R[P]$ requiring more time to execute than P.

  This is just my intuition, but I would like to get empirical evidence.

4.9.2 An LLVM Primer

LLVM IR is a strongly typed language. We interpret an LLVM IR program as a directed graph of basic blocks. A *basic block* (BB) is a finite sequence of non-terminating instructions followed by a single terminating instruction. In this work, a *terminating* instruction is either: 1) a return instruction, which terminates the program, 2) an unconditional transition to a single BB, or 3) a conditional branching where, depending on the branching condition, there is a transition to one of two BBs.

Without loss of generality, values used inside BBs are only accessible to the BB where they are declared (this can be enforced with the `opt` optimisation flag `-reg2mem`). We use *load* and *store* instructions to allow cross BB communication .

A figure/example may help?

that uses single-static assignment for virtual registers and has memory read/write operations to enable communication between

A function in LLVM consists of a header BB, and, without loss of generality, a single exit BB.

Dominator/Postdominator

4.9.3 The **`Linearise`** Algorithm

We start with a function F that has a single entry block and a single exit block. In a nutshell, the algorithm advances from the entry block until it finds a branch or the exit block. In the case of a branch $\texttt{br}cAB$, the algorithm recursively linearises the branches starting in blocks A and B until their postdominator $\texttt{postDom}(A, B)$, and then proceeds to continue linearisation towards the exit block.

Why don't we try to describe the algorithm considering loops? It's going to be a fun exercise.

4.9.4 A Premise for Efficient Repaired Code

The premise behind why our algorithm could compete against other algorithms that protect against spectre attacks is that fencing, the SoTA defence against Spectre attacks, prevents speculative execution (i.e., puts a halt in instruction pipeling) while our transformation promotes safe pipelining.

Algorithm 1 Linearisation algorithm

```
1: procedure LINEARISE(IN, OUT, $\Delta$)
2:     if IN==OUT then
3:         return [IN]
4:     else
5:         $t \leftarrow$ terminator(IN)
6:         if $t$ is jmp NXT then                              ▷ Unconditional Jump
7:             return (IN : Linearise(NXT, OUT, $\Delta$))           ▷ Continue from NXT
8:         else if $t$ is br C L R then               ▷ Conditional Branching on Secret
9:             PD $\leftarrow$ PostDominator(L, R)
10:            L′ $\leftarrow$ Linearise(L, PD)
11:            R′ $\leftarrow$ Linearise(R, PD)
12:            M $\leftarrow$ Predication(C, L′, R′)
13:            return (IN : M : Linearise(PD, OUT, $\Delta$))          ▷ Continue from PD
14:        end if
15:    end if
16: end procedure
```

Algorithm 2 Euclid's algorithm

```
1: procedure EUCLID($a, b$)                                  ▷ The g.c.d. of a and b
2:     $r \leftarrow a \bmod b$
3:     while $r \neq 0$ do                               ▷ We have the answer if r is 0
4:         $a \leftarrow b$
5:         $b \leftarrow r$
6:         $r \leftarrow a \bmod b$
7:     end while
8:     return $b$                                                    ▷ The gcd is b
9: end procedure
```

4.9.5 Enforcing Safe Behaviour

I originally thought that this section should be part of the GKAT rules, but after some consideration, if we were to assume that sanitisation is part of the actions (i.e. is a part of CTP practices), then this section becomes largely irrelevant for the theory. However, since LLVM can have problems when accessing arrays or dividing by zero, this should be mentioned in the LLVM section, probably.

Dividing by zero or accessing a position outside of an array are two examples of computations that may raise exceptions during runtime. These exceptions can be mitigated by sanitising the inputs to these functions. This sanitisation is often responsibility of the programmer, who uses conditionals to ensure that the inputs are safe to use. Since our repair rules remove conditionals by forcing the execution of both branch paths, they may introduce runtime exception that were not present in the original program. This is a well-known problem when using predication (see **Rane**), and to overcome it, we need to shift sanitisation of inputs to the compiler or to the architecture.

Consider the program **if** b **then** $y := z/x$ **else skip**. The repair rules would transform it into the program $y := b \otimes (z/x) + (\textbf{not}\, b) \otimes y$. If b is a condition that rules out whether x is not equal to zero, we might run into problems should we evaluate z/x in the repaired program. If $\otimes$ is to follow CTP practices, its execution time must not vary on the value of its (secret) input parameters, so early termination by checking if any of the parameters is zero is not allowed. If both parameters are fully evaluated before applying multiplication, we can be certain that the expression z/x is going to be evaluated, even if x is equal to 0. To avoid raising an exception at runtime, we enforce input sanitisation at compiler level by using a variable $x_{\texttt{div}}$ to assign a default safe value (e.g., 1) to the parameter of the division function in case the original variable is unsafe, yielding the following repaired program

$$x_{\texttt{div}} := (x \neq 0) \otimes x + (x = 0) \otimes 1 \; ; \; y := b \otimes (z/x_{\texttt{div}}) + (\textbf{not}\, b) \otimes y \; . \tag{4.8}$$

Since we evaluate $z/x_{\texttt{div}}$ instead of z/x in the repaired program, the repaired program is also safe if the original program was safe. We remark that the choice of the default safe value should be irrelevant in the context of CTP because, if the division operation also follows CTP practices, then it should not matter which values we choose as long as they do not break the functionality of the program.

Now, consider the program **if** b **then** $y := A[x]$ **else skip**. Similarly to the previous example, if we repair this program naively, we obtain the program $y := b \otimes A[x] + (\textbf{not}\, b) \otimes y$ which might introduce runtime exceptions should $A[x]$ be executed when x is greater than the size of A. We can sanitise the input by

$$x_{\texttt{idx}} := (x < \textbf{size}(A)) \otimes x; \; y := b \otimes A[x_{\texttt{idx}}] + (\textbf{not}\, b) \otimes y \; ,$$

OR

$$x_{\texttt{idx}} := x \;\textbf{mod}\; (\textbf{size}(A)); \; y := b \otimes A[x_{\texttt{idx}}] + (\textbf{not}\, b) \otimes y \; ,$$

to avoid undesired side effects.

I have the hunch that substituting the expression $A[x]$ for $A[x \textbf{ mod } (\textbf{size}(A))]$ might leak via memory SC that the secret is a multiple of the array index that corresponds to $x \textbf{ mod } (\textbf{size}(A))$. However, this substitution for $A[0]$ also leaks that the secret is greater than the size of the array. Now, using a secret as an index should be frowned upon by CTP guidelines, but do we want to define the problem away? probably not. What this is missing is the memory SC mitigation, and any of the options above should be valid.

4.9.6 A GKAT to Correctly Unwind Basic Block Loops

The objective of this section is to provide the background theory to prove that the loop transformation that we use in the CFG corresponds to an unwinding transformation of GKAT iteration expressions. For that, we show that the CFG of an LLVM IR program models a normal GKAT coalgebra. If that is the case, then, from a normal GKAT coalgebra, we can use Kleene's theorem for GKAT to obtain a GKAT expression that models the program. Once we have the GKAT expression, we can unwind loops using k-truncation. Somehow, this should be equivalent to unwinding a loop in the CFG (even if the loop has multiple exit points). If I am not mistaken, we want to identify two blocks: the header block and the merging block of the loop. The header is unique by definition, but the merging block is the postdominator of all exiting blocks in the loops, and that is what we will end up iterating using GKAT iteration.

The Control Flow Graph (CFG) of LLVM IR consists of Basic Blocks (BBs). Each BB has a single entry point and a single exit point, the latter represented by a *terminator instruction*. Let us consider only two types of termination instructions for BBs:

1. `return` v, which ends the current execution and returns a value v; and
2. `branch` α `:` BB_1 `;` BB_2, which states that we should continue execution from BB_1 if the condition α is valid, or continue execution from BB_2 otherwise.

The `branch` instruction can be generalised to a `switch` instruction, but we work with binary branching since it is possible to express a switch associating on the right, so we go o a basic block on the else branch that continues resolving the cases of the switch following the classical `switch` $g_i : c_i \equiv$ `if` g_1 `then` c_1 `elseif` g_2 `then` c_2 `elseif...` equivalence that most decent programming languages have (which is why I never use switches in C++, since they are not equivalent).

Let us consider a GKAT where the set of actions Σ is the set of BBs and the tests are the conditions used in `branch` termination instructions.

4.9.7 Types

For our first LLVM IR sublanguage, we support the following types:

- Void type, which cannot be assigned to variables.
- Integer types of the form i_N where $N \in [1..2^{23} - 1]$.

An element of type i_N is a vector of N bits; consequently, the type i_N is populated by maps of the form $N \to 2$. There are two standard interpretations for $x\colon i_N$, an *unsigned* interpretation $\mathrm{u}\colon i_N \to \mathbb{N}$, where

Set Notation

$$\mathrm{u}(x) = \sum_{n=0}^{N-1} x(n) \times 2^{x(n)}, \tag{4.9}$$

and a *signed* interpretation $\mathrm{s}\colon i_N \to \mathbb{Z}$, where

$$\mathrm{s}(x) = \begin{cases} \sum_{n=1}^{N-1} x(n) \times 2^{x(n)}, & \text{if } x(0) = 0; \\ -\sum_{n=1}^{N-1} x(n) \times 2^{x(n)}, & \text{if } x(0) = 1. \end{cases} \tag{4.10}$$

In other words, the first bit $x(0)$ determines whether x represents a positive or a negative number under the signed representation.

For the set of instructions we need to introduce the notion of variables and assignments. The set of *variables* v is generated by the regular expression

$$v ::= (@ \mid \%) \cdot (\text{A-Z} \mid \text{a-z}) \cdot (\text{A-Z} \mid \text{a-z} \mid \text{0-9} \mid . \mid \_ \mid \$)^* . \tag{4.11}$$

A *variable assignment* is an instruction of the form $v := e$, where e is a *non-void function* or an *arithmetic expression*. Arithmetic expressions are

- Language
- Arithmetic operations that use division.

4.9.8 Instructions

Overflows

4.9.9 Central idea

The main theoretical idea of our work is

what is it?

4.9.10 Subsection 1

Nunc posuere quam at lectus tristique eu ultrices augue venenatis. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Aliquam erat volutpat. Vivamus sodales tortor eget quam adipiscing in vulputate ante ullamcorper. Sed eros ante, lacinia et sollicitudin et, aliquam sit amet augue. In hac habitasse platea dictumst.

4.9.11 Subsection 2

Morbi rutrum odio eget arcu adipiscing sodales. Aenean et purus a est pulvinar pellentesque. Cras in elit neque, quis varius elit. Phasellus fringilla, nibh eu tempus venenatis, dolor elit posuere quam, quis adipiscing urna leo nec orci. Sed nec nulla auctor odio aliquet consequat. Ut nec nulla in ante ullamcorper aliquam at sed dolor. Phasellus fermentum magna in augue gravida cursus. Cras sed pretium lorem. Pellentesque eget ornare odio. Proin accumsan, massa viverra cursus pharetra, ipsum nisi lobortis velit, a malesuada dolor lorem eu neque.

4.10 Main Section 2

Sed ullamcorper quam eu nisl interdum at interdum enim egestas. Aliquam placerat justo sed lectus lobortis ut porta nisl porttitor. Vestibulum mi dolor, lacinia molestie gravida at, tempus vitae ligula. Donec eget quam sapien, in viverra eros. Donec pellentesque justo a massa fringilla non vestibulum metus vestibulum. Vestibulum in orci quis felis tempor lacinia. Vivamus ornare ultrices facilisis. Ut hendrerit volutpat vulputate. Morbi condimentum venenatis augue, id porta ipsum vulputate in. Curabitur luctus tempus justo. Vestibulum risus lectus, adipiscing nec condimentum quis, condimentum nec nisl. Aliquam dictum sagittis velit sed iaculis. Morbi tristique augue sit amet nulla pulvinar id facilisis ligula mollis. Nam elit libero, tincidunt ut aliquam at, molestie in quam. Aenean rhoncus vehicula hendrerit.

Chapter 5

First chapter $\Sigma_{\omega}^{i}\infty$

5.1 Welcome and Thank You

Welcome to this LaTeX Thesis Template, a beautiful and easy to use template for writing a thesis using the LaTeX typesetting system.

If you are writing a thesis (or will be in the future) and its subject is technical or mathematical (though it doesn't have to be), then creating it in LaTeX is highly recommended as a way to make sure you can just get down to the essential writing without having to worry over formatting or wasting time arguing with your word processor.

LaTeX is easily able to professionally typeset documents that run to hundreds or thousands of pages long. With simple mark-up commands, it automatically sets out the table of contents, margins, page headers and footers and keeps the formatting consistent and beautiful. One of its main strengths is the way it can easily typeset mathematics, even *heavy* mathematics. Even if those equations are the most horribly twisted and most difficult mathematical problems that can only be solved on a super-computer, you can at least count on LaTeX to make them look stunning.

5.2 Learning LaTeX

LaTeX is not a WYSIWYG (What You See is What You Get) program, unlike word processors such as Microsoft Word or Apple's Pages. Instead, a document written for LaTeX is actually a simple, plain text file that contains *no formatting*. You tell LaTeX how you want the formatting in the finished document by writing in simple commands amongst the text, for example, if I want to use *italic text for emphasis*, I write the `\emph{text}` command and put the text I want in italics in between the curly braces. This means that LaTeX is a "mark-up" language, very much like HTML.

5.2.1 A (not so short) Introduction to LaTeX

If you are new to LaTeX, there is a very good eBook – freely available online as a PDF file – called, "The Not So Short Introduction to LaTeX". The book's title is typically shortened to just *lshort*. You can download the latest version (as it is occasionally updated) from here: `http://www.ctan.org/tex-archive/info/lshort/english/lshort.pdf`

It is also available in several other languages. Find yours from the list on this page: `http://www.ctan.org/tex-archive/info/lshort/`

It is recommended to take a little time out to learn how to use LaTeX by creating several, small 'test' documents, or having a close look at several templates on:
`http://www.LaTeXTemplates.com`
Making the effort now means you're not stuck learning the system when what you *really* need to be doing is writing your thesis.

5.2.2 A Short Math Guide for LaTeX

If you are writing a technical or mathematical thesis, then you may want to read the document by the AMS (American Mathematical Society) called, "A Short Math Guide for LaTeX". It can be found online here: `http://www.ams.org/tex/amslatex.html` under the "Additional Documentation" section towards the bottom of the page.

5.2.3 Common LaTeX Math Symbols

There are a multitude of mathematical symbols available for LaTeX and it would take a great effort to learn the commands for them all. The most common ones you are likely to use are shown on this page: `http://www.sunilpatel.co.uk/latex-type/latex-math-symbols/`

You can use this page as a reference or crib sheet, the symbols are rendered as large, high quality images so you can quickly find the LaTeX command for the symbol you need.

5.2.4 LaTeX on a Mac

The LaTeX distribution is available for many systems including Windows, Linux and Mac OS X. The package for OS X is called MacTeX and it contains all the applications you need – bundled together and pre-customized – for a fully working LaTeX environment and work flow.

MacTeX includes a custom dedicated LaTeX editor called TeXShop for writing your '**`.tex`**' files and BibDesk: a program to manage your references and create your bibliography section just as easily as managing songs and creating playlists in iTunes.

5.3 Getting Started with this Template

If you are familiar with LaTeX, then you should explore the directory structure of the template and then proceed to place your own information into the *THESIS INFORMATION* block of the **`main.tex`** file. You can then modify the rest of this file to your unique specifications based on your degree/university. Section 5.5 on page 25 will help you do this. Make sure you also read section 5.7 about thesis conventions to get the most out of this template.

If you are new to LaTeX it is recommended that you carry on reading through the rest of the information in this document.

Before you begin using this template you should ensure that its style complies with the thesis style guidelines imposed by your institution. In most cases this template style and layout will be suitable. If it is not, it may only require a small change to bring the template in line with your institution's recommendations. These modifications will need to be done on the **`MastersDoctoralThesis.cls`** file.

5.3.1 About this Template

This LaTeX Thesis Template is originally based and created around a LaTeX style file created by Steve R. Gunn from the University of Southampton (UK), department of Electronics and Computer Science. You can find his original thesis style file at his site, here: `http://www.ecs.soton.ac.uk/~srg/softwaretools/document/templates/`

Steve's **`ecsthesis.cls`** was then taken by Sunil Patel who modified it by creating a skeleton framework and folder structure to place the thesis files in. The resulting template can be found on Sunil's site here: `http://www.sunilpatel.co.uk/thesis-template`

Sunil's template was made available through `http://www.LaTeXTemplates.com` where it was modified many times based on user requests and questions. Version 2.0 and onwards of this template represents a major modification to Sunil's template and is, in fact, hardly recognisable. The work to make version 2.0 possible was carried out by Vel and Johannes Böttcher.

Martín Ochoa at SUTD modified it to comply with the university's requirements.

5.4 What this Template Includes

5.4.1 Folders

This template comes as a single zip file that expands out to several files and folders. The folder names are mostly self-explanatory:

Appendices – this is the folder where you put the appendices. Each appendix should go into its own separate **`.tex`** file. An example and template are included in the directory.

Chapters – this is the folder where you put the thesis chapters. A thesis usually has about six chapters, though there is no hard rule on this. Each chapter should go in its own separate **`.tex`** file and they can be split as:

- Chapter 1: Introduction to the thesis topic
- Chapter 2: Background information and theory
- Chapter 3: (Laboratory) experimental setup
- Chapter 4: Details of experiment 1
- Chapter 5: Details of experiment 2
- Chapter 6: Discussion of the experimental results
- Chapter 7: Conclusion and future directions

This chapter layout is specialised for the experimental sciences.

Figures – this folder contains all figures for the thesis. These are the final images that will go into the thesis document.

5.4.2 Files

Included are also several files, most of them are plain text and you can see their contents in a text editor. After initial compilation, you will see that more auxiliary files are created by LaTeX or BibTeX and which you don't need to delete or worry about:

example.bib – this is an important file that contains all the bibliographic information and references that you will be citing in the thesis for use with BibTeX. You can write it manually, but there are reference manager programs available that will create and manage it for you. Bibliographies in LaTeX are a large subject and you may need to read about BibTeX before starting with this. Many modern reference managers will allow you to export your references in BibTeX format which greatly eases the amount of work you have to do.

MastersDoctoralThesis.cls – this is an important file. It is the class file that tells LaTeX how to format the thesis.

main.pdf – this is your beautifully typeset thesis (in the PDF file format) created by LaTeX. It is supplied in the PDF with the template and after you compile the template you should get an identical version.

main.tex – this is an important file. This is the file that you tell LaTeX to compile to produce your thesis as a PDF file. It contains the framework and constructs that tell LaTeX how to layout the thesis. It is heavily commented so you can read exactly what each line of code does and why it is there. After you put your own information into the *THESIS INFORMATION* block – you have now started your thesis!

Files that are *not* included, but are created by LaTeX as auxiliary files include:

main.aux – this is an auxiliary file generated by LaTeX, if it is deleted LaTeX simply regenerates it when you run the main **`.tex`** file.

main.bbl – this is an auxiliary file generated by BibTeX, if it is deleted, BibTeX simply regenerates it when you run the **`main.aux`** file. Whereas the **`.bib`** file contains all the references you have, this **`.bbl`** file contains the references you have actually cited in the thesis and is used to build the bibliography section of the thesis.

main.blg – this is an auxiliary file generated by BibTeX, if it is deleted BibTeX simply regenerates it when you run the main **`.aux`** file.

main.lof – this is an auxiliary file generated by LaTeX, if it is deleted LaTeX simply regenerates it when you run the main **`.tex`** file. It tells LaTeX how to build the *List of Figures* section.

main.log – this is an auxiliary file generated by LaTeX, if it is deleted LaTeX simply regenerates it when you run the main **`.tex`** file. It contains messages from LaTeX, if you receive errors and warnings from LaTeX, they will be in this **`.log`** file.

main.lot – this is an auxiliary file generated by LaTeX, if it is deleted LaTeX simply regenerates it when you run the main **`.tex`** file. It tells LaTeX how to build the *List of Tables* section.

main.out – this is an auxiliary file generated by LaTeX, if it is deleted LaTeX simply regenerates it when you run the main **`.tex`** file.

So from this long list, only the files with the **`.bib`**, **`.cls`** and **`.tex`** extensions are the most important ones. The other auxiliary files can be ignored or deleted as LaTeX and BibTeX will regenerate them.

5.5 Filling in Your Information in the `main.tex` File

You will need to personalise the thesis template and make it your own by filling in your own information. This is done by editing the **`main.tex`** file in a text editor or your favourite LaTeX environment.

Open the file and scroll down to the second large block titled *THESIS INFORMATION* where you can see the entries for *University Name*, *Department Name*, etc ...

Fill out the information about yourself, your group and institution. You can also insert web links, if you do, make sure you use the full URL, including the `http://` for this. If you don't want these to be linked, simply remove the `\href{url}{name}` and only leave the name.

When you have done this, save the file and recompile `main.tex`. All the information you filled in should now be in the PDF, complete with web links. You can now begin your thesis proper!

5.6 The `main.tex` File Explained

The **`main.tex`** file contains the structure of the thesis. There are plenty of written comments that explain what pages, sections and formatting the LaTeX code is creating. Each major document element is divided into commented blocks with titles in all capitals to make it obvious what the following bit of code is doing. Initially there seems to be a lot of LaTeX code, but this is all formatting, and it has all been taken care of so you don't have to do it.

Begin by checking that your information on the title page is correct. For the thesis declaration, your institution may insist on something different than the text given. If this is the case, just replace what you see with what is required in the *DECLARATION PAGE* block.

Then comes a page which contains a funny quote. You can put your own, or quote your favourite scientist, author, person, and so on. Make sure to put the name of the person who you took the quote from.

Following this is the abstract page which summarises your work in a condensed way and can almost be used as a standalone document to describe what you have done. The text you write will cause the heading to move up so don't worry about running out of space.

Next come the acknowledgements. On this page, write about all the people who you wish to thank (not forgetting parents, partners and your advisor/supervisor).

The contents pages, list of figures and tables are all taken care of for you and do not need to be manually created or edited. The next set of pages are more likely to be optional and can be deleted since they are for a more technical thesis: insert a list of abbreviations you have used in the thesis, then a list of the physical constants and numbers you refer to and finally, a list of mathematical symbols used in any formulae. Making the effort to fill these tables means the reader has a one-stop place to refer to instead of searching the internet and references to try and find out what you meant by certain abbreviations or symbols.

The list of symbols is split into the Roman and Greek alphabets. Whereas the abbreviations and symbols ought to be listed in alphabetical order (and this is *not* done automatically for you) the list of physical constants should be grouped into similar themes.

The next page contains a one line dedication. Who will you dedicate your thesis to?

Finally, there is the block where the chapters are included. Uncomment the lines (delete the `%` character) as you write the chapters. Each chapter should be written in its own file and put into the *Chapters* folder and named **`Chapter1`**, **`Chapter2`**, etc... Similarly for the appendices, uncomment the lines as you need them. Each appendix should go into its own file and placed in the *Appendices* folder.

After the preamble, chapters and appendices finally comes the bibliography. The bibliography style (called *`authoryear`*) is used for the bibliography and is a fully featured style that will even include links to where the referenced paper can be found online. Do not underestimate how grateful your reader will be to find that a reference to a paper is just a click away. Of course, this relies on you putting the URL information into the BibTeX file in the first place.

5.7 Thesis Features and Conventions

To get the best out of this template, there are a few conventions that you may want to follow.

One of the most important (and most difficult) things to keep track of in such a long document as a thesis is consistency. Using certain conventions and ways of doing things (such as using a Todo list) makes the job easier. Of course, all of these are optional and you can adopt your own method.

5.7.1 Printing Format

This thesis template is designed for double sided printing (i.e. content on the front and back of pages) as most theses are printed and bound this way. Switching to one sided printing is as simple as uncommenting the *`oneside`* option of the `documentclass` command at the top of the **`main.tex`** file. You may then wish to adjust the margins to suit specifications from your institution.

The headers for the pages contain the page number on the outer side (so it is easy to flick through to the page you want) and the chapter name on the inner side.

The text is set to 11 point by default with single line spacing, again, you can tune the text size and spacing should you want or need to using the options at the very start of **`main.tex`**. The spacing can be changed similarly by replacing the *`singlespacing`* with *`onehalfspacing`* or *`doublespacing`*.

5.7.2 Using US Letter Paper

The paper size used in the template is A4, which is the standard size in Europe. If you are using this thesis template elsewhere and particularly in the United States, then you may have to change the A4 paper size to the US Letter size. This can be done in the margins settings section in **`main.tex`**.

Due to the differences in the paper size, the resulting margins may be different to what you like or require (as it is common for institutions to dictate certain margin sizes).

If this is the case, then the margin sizes can be tweaked by modifying the values in the same block as where you set the paper size. Now your document should be set up for US Letter paper size with suitable margins.

5.7.3 References

The `biblatex` package is used to format the bibliography and inserts references such as this one (Hawthorn, Weber, and Scholten, 2001). The options used in the **`main.tex`** file mean that the in-text citations of references are formatted with the author(s) listed with the date of the publication. Multiple references are separated by semicolons (e.g. (Wieman and Hollberg, 1991; Hawthorn, Weber, and Scholten, 2001)) and references with more than three authors only show the first author with *et al.* indicating there are more authors (e.g. (Arnold et al., 1998)). This is done automatically for you. To see how you use references, have a look at the **`Chapter1.tex`** source file. Many reference managers allow you to simply drag the reference into the document as you type.

Scientific references should come *before* the punctuation mark if there is one (such as a comma or period). The same goes for footnotes[1]. You can change this but the most important thing is to keep the convention consistent throughout the thesis. Footnotes themselves should be full, descriptive sentences (beginning with a capital letter and ending with a full stop). The APA6 states: "Footnote numbers should be superscripted, [...], following any punctuation mark except a dash." The Chicago manual of style states: "A note number should be placed at the end of a sentence or clause. The number follows any punctuation mark except the dash, which it precedes. It follows a closing parenthesis."

The bibliography is typeset with references listed in alphabetical order by the first author's last name. This is similar to the APA referencing style. To see how LaTeX typesets the bibliography, have a look at the very end of this document (or just click on the reference number links in in-text citations).

A Note on bibtex

The bibtex backend used in the template by default does not correctly handle unicode character encoding (i.e. "international" characters). You may see a warning about this in the compilation log and, if your references contain unicode characters, they may not show up correctly or at all. The solution to this is to use the biber backend instead of the outdated bibtex backend. This is done by finding this in **`main.tex`**: *`backend=bibtex`* and changing it to *`backend=biber`*. You will then need to delete all auxiliary BibTeX files and navigate to the template directory in your terminal (command prompt). Once there, simply type `biber main` and biber will compile your bibliography. You can then compile **`main.tex`** as normal and your bibliography will be updated. An alternative is to set up your LaTeX editor to compile with biber instead of bibtex, see here for how to do this for various editors.

[1]Such as this footnote, here down at the bottom of the page.

TABLE 5.1: The effects of treatments X and Y on the four groups studied.

Groups	**Treatment X**	**Treatment Y**
1	0.2	0.8
2	0.17	0.7
3	0.24	0.75
4	0.68	0.3

5.7.4 Tables

Tables are an important way of displaying your results, below is an example table which was generated with this code:

```
\begin{table}
\caption{The effects of treatments X and Y on the four groups studied.}
\label{tab:treatments}
\centering
\begin{tabular}{l l l}
\toprule
\tabhead{Groups} & \tabhead{Treatment X} & \tabhead{Treatment Y} \\
\midrule
1 & 0.2 & 0.8\\
2 & 0.17 & 0.7\\
3 & 0.24 & 0.75\\
4 & 0.68 & 0.3\\
\bottomrule\\
\end{tabular}
\end{table}
```

You can reference tables with `\ref{<label>}` where the label is defined within the table environment. See **`Chapter1.tex`** for an example of the label and citation (e.g. Table 5.1).

5.7.5 Figures

There will hopefully be many figures in your thesis (that should be placed in the *Figures* folder). The way to insert figures into your thesis is to use a code template like this:

```
\begin{figure}
\centering
\includegraphics{Figures/Electron}
\decoRule
\caption[An Electron]{An electron (artist's impression).}
\label{fig:Electron}
\end{figure}
```

Also look in the source file. Putting this code into the source file produces the picture of the electron that you can see in the figure below.

FIGURE 5.1: An electron (artist's impression).

Sometimes figures don't always appear where you write them in the source. The placement depends on how much space there is on the page for the figure. Sometimes there is not enough room to fit a figure directly where it should go (in relation to the text) and so LaTeX puts it at the top of the next page. Positioning figures is the job of LaTeX and so you should only worry about making them look good!

Figures usually should have captions just in case you need to refer to them (such as in Figure 5.1). The `\caption` command contains two parts, the first part, inside the square brackets is the title that will appear in the *List of Figures*, and so should be short. The second part in the curly brackets should contain the longer and more descriptive caption text.

The `\decoRule` command is optional and simply puts an aesthetic horizontal line below the image. If you do this for one image, do it for all of them.

LaTeX is capable of using images in pdf, jpg and png format.

5.7.6 Typesetting mathematics

If your thesis is going to contain heavy mathematical content, be sure that LaTeX will make it look beautiful, even though it won't be able to solve the equations for you.

The "Not So Short Introduction to LaTeX" (available on CTAN) should tell you everything you need to know for most cases of typesetting mathematics. If you need more information, a much more thorough mathematical guide is available from the

AMS called, "A Short Math Guide to LaTeX" and can be downloaded from: `ftp://ftp.ams.org/pub/tex/doc/amsmath/short-math-guide.pdf`

There are many different LaTeX symbols to remember, luckily you can find the most common symbols in The Comprehensive LaTeX Symbol List.

You can write an equation, which is automatically given an equation number by LaTeX like this:

```
\begin{equation}
E = mc^{2}
\label{eqn:Einstein}
\end{equation}
```

This will produce Einstein's famous energy-matter equivalence equation:

$$E = mc^{2} \tag{5.1}$$

All equations you write (which are not in the middle of paragraph text) are automatically given equation numbers by LaTeX. If you don't want a particular equation numbered, use the unnumbered form:

```
\[ a^{2}=4 \]
```

5.8 Sectioning and Subsectioning

You should break your thesis up into nice, bite-sized sections and subsections. LaTeX automatically builds a table of Contents by looking at all the `\chapter{}`, `\section{}` and `\subsection{}` commands you write in the source.

The Table of Contents should only list the sections to three (3) levels. A `chapter{}` is level zero (0). A `\section{}` is level one (1) and so a `\subsection{}` is level two (2). In your thesis it is likely that you will even use a `subsubsection{}`, which is level three (3). The depth to which the Table of Contents is formatted is set within **`MastersDoctoralThesis.cls`**. If you need this changed, you can do it in **`main.tex`**.

5.9 In Closing

You have reached the end of this mini-guide. You can now rename or overwrite this pdf file and begin writing your own **`Chapter1.tex`** and the rest of your thesis. The easy work of setting up the structure and framework has been taken care of for you. It's now your job to fill it out!

Good luck and have lots of fun!

Guide written by —
Sunil Patel: www.sunilpatel.co.uk
Vel: LaTeXTemplates.com

Appendix A

Appendix Title Here

Write your Appendix content here.

Bibliography

Arnold, A. S. et al. (1998). “A Simple Extended-Cavity Diode Laser”. In: *Review of Scientific Instruments* 69.3, pp. 1236–1239. URL: http://link.aip.org/link/?RSI/69/1236/1.

Hawthorn, C. J., K. P. Weber, and R. E. Scholten (2001). “Littrow Configuration Tunable External Cavity Diode Laser with Fixed Direction Output Beam”. In: *Review of Scientific Instruments* 72.12, pp. 4477–4479. URL: http://link.aip.org/link/?RSI/72/4477/1.

Wieman, Carl E. and Leo Hollberg (1991). “Using Diode Lasers for Atomic Physics”. In: *Review of Scientific Instruments* 62.1, pp. 1–20. URL: http://link.aip.org/link/?RSI/62/1/1.