Thesis for The Degree of Licentiate of Engineering

Bits of Knowledge: Combining Probabilistic and Formal Techniques for Secure and Low-Power Hardware Design

Henrik Jansson Valter

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden, 2025

Bits of Knowledge: Combining Probabilistic and Formal Techniques for Secure and Low-Power Hardware Design

Henrik Jansson Valter

Department of Computer Science and Engineering
Division of Computing Science
Chalmers University of Technology and University of Gothenburg
SE-41296 Gothenburg, Sweden
Phone: +46(0)31 772 1000

Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2025.

"With great power comes great responsibility"
- Uncle Ben, Spider-Man

Abstract

Power consumption is a major concern in hardware design. Additionally, power usage can be exploited in side-channel attacks, turning power into a security vulnerability. This thesis lays the groundwork for developing side-channel resistant hardware by developing tools that combine power analysis, formal verification, and probabilistic models in order to rigorously establish security guarantees.

We begin by presenting a simple power model for CMOS circuits, computable using BDD-based symbolic simulation. This allows the power consumption to be expressed directly as a function of the circuit inputs, shifting the focus to symbolically representing the input distribution. While there are methods for generating symbolic inputs, they have no guarantees with regards to the distribution of generated vectors. On the other hand, there are methods that do have some guarantee on the distribution, but these do not support symbolic simulation. The latter methods are also restricted to generating uniform distributions. This problem is addressed in one of our papers. We introduce methods for defining arbitrary input distributions in a way that supports symbolic simulation, using BDDs as the core computational tool. Beyond power analysis, these introduced methods are widely applicable in both software and hardware verification.

We also discuss the implementation and evaluation of a low-power custom processor for high-level languages, detailing decisions for minimizing energy consumption for both core and memory. This is compared to a low-power RISC-V core running a high-level language in software, showing favorable results for the custom design.

Keywords

Low-power computing, Formal verification, Probabilistic modeling, Computer Architecture, Functional programming

List of Publications

Appended publications

This thesis is based on the following publications:

[**Paper I**] Jeremy Pope, Carl-Johan H. Seger, **Henrik Valter**, *Higher-order Hardware: Implementation and Evaluation of the Cephalopode Graph Reduction Processor*
MEMOCODE, 2024.

[**Paper II**] **Henrik Jansson Valter**, Carl-Johan H. Seger, *BDD-Based Methods for Constrained and Biased Simulation-Vector Generation*
Under submission.

Author contribution

For **Paper I**, the Cephalopode architecture was jointly designed by Jeremy Pope and Carl Seger, while I conducted the evaluation in its entirety, with the exception of writing the benchmark programs. In addition to carrying out the evaluation and writing the associated paper sections, I contributed the introduction, conclusion, and abstract. I was also responsible for presenting the work at the conference.
For **Paper II**, I prepared the initial drafts and wrote the bulk of the paper. Carl later contributed substantial revisions and improvements. We shared responsibility equally for the implementation and evaluation components.

Acknowledgments

I would like to begin by expressing my sincere gratitude to my PhD advisor, Carl Seger. Your dedication and commitment to supervision have meant so much to me and I greatly appreciate your passion for the subjects. I am also incredibly thankful to Mary Sheeran for being an excellent co-advisor, always challenging me with thought-provoking questions about my research. For the same reasons I would like to thank my examiner Per Stenström.

Additionally, I extend my appreciation to Jeremy Pope for welcoming me onto my first paper and giving me the opportunity to contribute to such a fascinating project. Regarding the first paper, my contribution would not have been possible without Lennart Augustsson, who implemented MicroHs and helped me get it running on the RISC-V processor. The final person I would like to thank in connection with the first paper is Lars Svensson, who guided me in getting started with the CAD tools.

Next, I would like to thank Patrik Jansson for extensive feedback on both the second paper and this thesis. Furthermore, I want to thank Mads Dam and Tamara Rezk for interesting discussions regarding applying my research in the domain of computer security.

I also express my gratitude to my colleagues in the department. Rather than listing everyone, I will highlight a few whose support has been especially invaluable: Robert, Abhiroop, Prabhat, Katya, Wincent, Niklas, and Hanna. Lastly, I want to extend my heartfelt thanks to my friends, family, and everyone else who has supported me, even during the difficult times. Above all, I am deeply grateful to my wife, Julia Jansson Valter. This thesis would not have been possible without your love, help, and encouragement.

Henrik Jansson Valter
Göteborg, March 2025

Contents

Part I

Summary

Chapter 1

Introduction

Power consumption is a critical design concern for systems across the computing spectrum, from large-scale data clusters to resource-constrained embedded systems.

To illustrate this at the scale of the largest computing systems, consider the TOP500 project, which ranks the world's most powerful non-distributed supercomputers. As of November 2024, the leading system was El Capitan, located at Lawrence Livermore National Laboratory in California, with a reported power consumption of nearly 30 MW [1]. A rough calculation shows that such a system would consume approximately 262,800 MWh of energy annually. For context, the average US household consumes around 10 MWh per year [2], meaning El Capitan alone uses as much electricity as 26,000 households. As another example, Bitcoin mining has been estimated to consume about 160,000 GWh annually, equivalent to the energy use of roughly 1.6 million US homes [3].

For medium-scale computing systems, such as typical desktop machines, power consumption became a major design constraint starting in the mid-to-late 1990s. Prior to that, performance was the dominant priority in computer architecture. This focus was enabled by trends such as Moore's Law, which observed that the number of transistors on a dense integrated circuit doubles roughly every two years [4], and Dennard scaling, which predicted that power consumption would remain proportional to chip area as transistors shrink [5]. These developments allowed processor clock speeds to steadily increase. However, by the late 1990s, it became evident that further frequency gains were unsustainable due to thermal limitations. Clock speeds plateaued around 3 GHz, beyond which cooling becomes prohibitively difficult.

In small-scale systems like embedded devices, another trade-off must be considered. These systems are often battery-powered and expected to operate for years without maintenance, making ultra-low energy consumption essential. As a result, available compute power and memory on such chips are extremely limited. Additionally, many of these systems must meet real-time constraints, where unpredictable pauses—such as those caused by garbage collection—are unacceptable. Given these constraints, developers typically rely on low-

level languages like C, C++, or even assembly. While these languages offer high performance and minimal runtime overhead, they sacrifice the safety and abstraction benefits provided by higher-level languages. This trade-off becomes especially risky when these devices are internet-connected, forming part of the so-called Internet of Things (IoT). A 2024 report by the US Cybersecurity and Infrastructure Security Agency (CISA) highlighted this risk, urging developers to move away from C/C++ in favor of memory-safe high-level languages, noting that the majority of security vulnerabilities assigned a CVE (Common Vulnerabilities and Exposures) stem from issues inherent in low-level programming [6].

The first paper included in this thesis introduces the Cephalopode processor—a low-power, custom-designed processor built to support the execution of high-level languages directly in hardware. Specifically, it targets a functional programming language similar to Haskell. Functional languages help eliminate many classes of low-level bugs and provide powerful safety features such as strong static typing, higher-order functions, and polymorphism. The latter two promote significant code reuse, and less code generally translates to a reduced risk of errors. However, the standard Haskell runtime is far too large for the constrained memory typical of embedded systems, and its reliance on garbage collection makes it unsuitable for real-time applications. Cephalopode addresses these limitations with a lightweight execution engine and includes a concurrent, hardware-based garbage collector. We explore this in more detail in the paper summary in Section 3.1.

In the remainder of this thesis, we focus on more direct aspects of power consumption within the context of computer security. In simple terms: if an attacker can monitor a device's power consumption while it processes secret data, can they extract that secret? In 1999, Kocher et al. introduced a landmark method called differential power analysis (DPA) to address this question [7]. They demonstrated that by carefully observing the power consumption of a device executing rounds of the Data Encryption Standard (DES), it was possible to recover the secret key. This discovery generated significant interest in the field, leading to a detailed follow-up paper in 2011 [8].

One might question the practicality of such attacks, as they appear to require physical access to the circuit—at which point there are simpler methods to extract cryptographic secrets. Is it possible to perform power analysis attacks remotely? Perhaps. In their 2023 paper, Wang et al. introduce the *Hertzbleed* attack [9], which exploits the dynamic power management of a chip. By increasing the chip's power consumption, the attack induces thermal throttling, causing the processor to lower its clock frequency. This frequency reduction results in measurable slowdowns that can be exploited using traditional timing side-channel techniques. In effect, the attack translates remote power side-channel leakage into a remote timing side-channel attack. That said, the practicality of Hertzbleed remains debatable, as the authors themselves acknowledge in a follow-up study [10].

These attacks fall under the category of side-channel attacks, which exploit correlations between physical measurements—such as power consumption or, more commonly, timing variations—and secret information processed by a

system. The threat posed by side-channel attacks has intensified with the advent of machine learning techniques, as shown in works by Dubrova et al. [11]–[13]. While these studies effectively demonstrate the feasibility and success of such attacks, they provide limited insight into the underlying circuit behaviors, data dependencies, or leakage mechanisms that make them possible. In short, they show *that* attacks succeed, but offer little understanding of *why* they do.

This thesis aims to take steps toward addressing this gap. We focus on developing tools and methodologies to answer critical questions: Can systems be designed to resist (remote) power side-channel attacks? How can such side-channels be detected? Is it possible to formally prove that a circuit is secure? And if complete security cannot be achieved, can we at least characterize and bound the vulnerabilities that remain?

Answering these questions requires powerful tools and methods for analyzing the power consumption of circuits. Specifically, we need techniques that allow us to assess the sensitivity of power consumption to particular circuit inputs — for instance, how much the power usage depends on the choice of secret key—or to evaluate the power behavior when inputs or outputs follow certain probabilistic distributions. A large portion of this thesis, including the second paper, is dedicated to developing power models capable of addressing these types of questions. Research in this area naturally blends concepts from circuit-level power estimation (from a VLSI perspective), formal verification (to establish guarantees and reduce large input spaces), and probabilistic analysis.

Circuit-level power estimation can be approached from either an empirical or an analytical perspective. These terms are fairly intuitive: empirical estimation involves directly measuring the power consumption of a running chip, while analytical estimation derives power models based on equations and abstractions. Empirical measurements tend to be more accurate, capturing higher-order effects that analytical models may overlook. However, they require completing the full design and fabrication process before meaningful data can be collected. In contrast, analytical models offer early-stage feedback during the design process, though with less accuracy [14].

In this work, we adopt a white-box, analytical approach to power estimation. Although this simplified model provides useful insights, a more detailed investigation is required to make definitive claims about side-channel resilience.

1.1 Aims

This thesis aims to integrate probabilistic and formal methods to support secure and energy-efficient hardware design. The first paper introduces and evaluates a custom processor architecture designed for high-level functional languages, targeting resource-constrained embedded systems where minimizing energy consumption is critical. This design is motivated by the security vulnerabilities commonly found in low-level languages. The core of the thesis focuses on developing tools and methodologies for analyzing power consumption in digital circuits. In this context, the second paper makes a central contribution by

enabling the efficient generation of input vectors based on arbitrary input distributions, supporting both scalar and symbolic simulation. These tools are specifically designed to aid in the construction of hardware that is resistant to power side-channel attacks.

Chapter 2

Background

This chapter presents the background necessary for the rest of the thesis. We begin with an introduction to Boolean formulas and how they are implemented in CMOS (Complementary Metal-Oxide-Semiconductor) circuits, a widely used technology in digital hardware. Next, we explore the sources of power dissipation in such circuits and construct a simplified model to estimate it. Due to the exponential size of the input space, we apply formal verification methods to explore this space efficiently. This leads to a discussion of methods for power estimation, which lays the groundwork for the discussions for the first paper, and also presents symbolic methods for power estimation, which is the motivation for the second paper.

2.1 Boolean formulas

Boolean formulas are built using the two constants true and false, along with logical operators that combine them. These operators include negation (NOT), conjunction (AND), and disjunction (OR). In computer science, these are typically denoted by $\neg$, $\wedge$, and $\vee$, respectively. Boolean variables, like constants, can take on the values true or false. For example, consider the Boolean formula

$$f = (a \vee b \vee c) \wedge (\neg a \vee b),$$

which evaluates to true whenever $b = \text{true}$. In this work we adopt a different notation, preferred in the context of computer engineering. Negation is written using an overline, conjunction as $\cdot$ (or omitted), and disjunction as $+$. Under this convention, the formula above becomes

$$f = (a + b + c)(\overline{a} + b)$$

Furthermore, we will denote the constants true and false with T and F, respectively. This is to match the notation of VossII, which will be introduced later in Section 2.7. We denote the set $\{F, T\}$ by $\mathbb{B}$. In computer engineering the notation 0 and 1 is also common, representing low and high voltage.

Multiple Boolean formulas can represent the same function, meaning they produce identical results for all possible variable assignments. For instance, the formulas $f = \overline{a + b}$ and $g = \overline{a} \cdot \overline{b}$ are equivalent for any assignment of a and b due to De Morgan's law.

Boolean formulas are commonly expressed in one of two standard forms: Conjunctive Normal Form (CNF) and Disjunctive Normal Form (DNF). A formula is in CNF if it is written as a conjunction of disjunctions (also known as a product of sums), while it is in DNF if written as a disjunction of conjunctions (or sum of products). For example, the formula

$$f = (a + b + c)(\overline{a} + b)$$

is in CNF, whereas the formula

$$g = \overline{a}c + b$$

is in DNF. Note that both of these functions represent the same Boolean formula.

Before proceeding, there is an important note to make regarding terminology. In the context of this thesis, we will refer to Boolean formulas F and T as *scalars*, and all others, i.e. those that have variable dependencies, as *symbolic*. In contrast, a *vector* in this work typically refers to an array of Boolean formulas, the elements of which can be scalar or symbolic. This should not be confused with the standard mathematical terminology of scalars and vectors.

2.2 Combinational circuits

In this work, our focus is primarily on circuits that implement Boolean functions—that is, combinational circuits. These digital logic circuits compute pure Boolean functions and do not contain any memory elements. This is in contrast to sequential circuits, which include memory and state, with finite state machines being the simplest example. Figure 2.1 shows a tiny combinational circuit with two input bits a and b and a single output bit e, where e implements the Boolean function $e = \overline{a} + ab$.

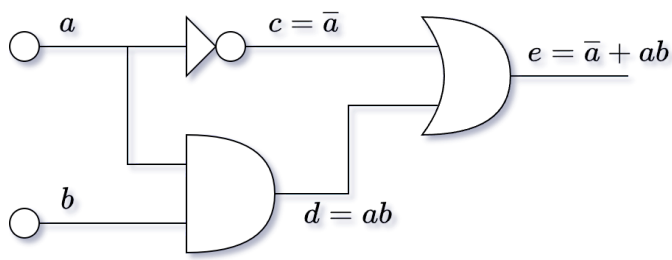


Figure 2.1: A tiny combinational circuit implementing the formula $e = \overline{a} + ab$.

In a more general sense, we can think of combinational circuits as a mapping from a vector of n bits to a vector of m bits, as illustrated in Figure 2.2.

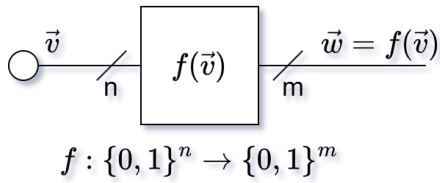


Figure 2.2: A general view of combinational circuits, as a mapping from n to m bits.

In our analysis, we will consider circuits implemented using CMOS (Complementary Metal-Oxide-Semiconductor), which is the most widely used technology for building integrated circuits such as processors and memory. It relies on two types of transistors: NMOS (N-channel MOS) and PMOS (P-channel MOS). NMOS transistors conduct electricity when their control signal (called the gate) is high, while PMOS transistors conduct when their gate is low. By combining NMOS and PMOS transistors, CMOS circuits achieve low static power consumption, because very little current flows while maintaining the current state [4].

In a CMOS gate, logic functions are realized by combining PMOS and NMOS transistors in a way that the output corresponds to a Boolean function of the inputs. Figure 2.3 shows an example with a NAND gate built using CMOS technology. A NAND gate outputs F only when both inputs are T, and T otherwise. As seen in the figure, when both A and B are T, the lower (NMOS) network connects the output to ground, while the upper (PMOS) network disconnects V_{dd}, resulting in output F. In all other cases, the PMOS network connects V_{dd}, and the NMOS network is incomplete, resulting in a T. The use of these complementary pull-up and pull-down networks is what gives CMOS—Complementary Metal-Oxide-Semiconductor—its name.

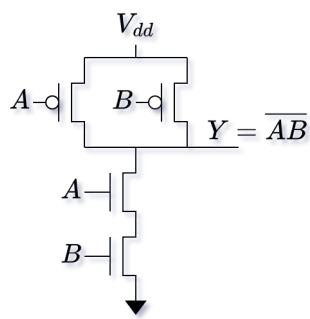


Figure 2.3: A NAND gate in CMOS.

2.3 Power consumption in CMOS circuits

The power consumption of CMOS circuits can be broadly categorized into static power, which is consumed when the circuit is powered but idle, and dynamic power, which arises from circuit activity during operation. While the primary focus of this work is on dynamic power, we begin with a brief discussion of static power.

Historically, static power constituted only a negligible portion of a chip's total power consumption. However, with continued technology scaling, this is no longer the case—static power has become a significant contributor, particularly in embedded systems, which are often idle for extended periods. The primary source of static power is subthreshold leakage, which is heavily influenced by the threshold voltage, V_{th}. Reducing this leakage requires a higher threshold voltage, but this comes at the cost of reduced switching speed, since circuits operate more slowly as V_{th} increases [4], [14].

This trade-off was carefully considered during the synthesis of the Cephalopode processor, specifically in the selection of standard cell libraries. We opted for regular threshold voltage cells over super-low threshold variants in order to minimize static power consumption, aligning with our goals for energy efficiency [15]. As result, the static power consumption only made up 1.5 % of the power consumption in Cephalopode.

We now focus on dynamic power consumption, which remains the primary contributor to power usage in contemporary digital circuits. This type of power consumption can be broken down into switching power P_{sw}, resulting from signal transitions within the circuit, and short-circuit power P_{sc}. Short-circuit power arises due to non-instantaneous input transitions, leading to both PMOS and NMOS transistors conducting simultaneously and creating a direct path from the supply to ground. P_{sc} can account for 10-20 % of the total dynamic power [4], [16]. In this work, we concentrate solely on switching power, which constitutes the remaining 80-90 %.

For switching power, energy is consumed in both charging *and* discharging the load capacitance C driven by the gate output line (Y in Figure 2.3). For a given gate g, the energy consumption can be expressed as:

$$E_g = C_g \cdot V_g^2 \cdot f_g \tag{2.1}$$

where C_g, V_g, and f_g represents the load capacitance, supply voltage and clock frequency of the gate, respectively. A common assumption is that toggling the gate from 0 to 1 and 1 to 0 consumes equal energy. Under this assumption, the energy required for a single toggle becomes

$$E_g = \frac{C_g \cdot V_g^2 \cdot f_g}{2}. \tag{2.2}$$

Let α_g denote the average number of toggles per clock cycle at gate g. We then arrive at the formula:

$$P_g = \frac{\alpha_g \cdot C_g \cdot V_g^2 \cdot f_g}{2}. \tag{2.3}$$

To find the total dynamic power, we sum the contributions from all gates $g \in G$:

$$P = \frac{1}{2} \sum_{g \in G} \alpha_g \cdot C_g \cdot V_g^2 \cdot f_g. \tag{2.4}$$

Finally, we assume all gates are driven by the same supply voltage and clock frequency, which simplifies to

$$P = \frac{V^2 f}{2} \sum_{g \in G} \alpha_g \cdot C_g. \tag{2.5}$$

Among the parameters in this equation, only α_g, known as the *activity factor*, depends on the workload. Accurately and efficiently determining this parameter is a central challenge in power estimation, and there is extensive literature addressing this issue [17]–[21]. A crucial insight is that α can be computed without the need for complex simulation tools: for a given input vector, the output of each gate can be evaluated as a Boolean function of the inputs, allowing us to check whether a transition (toggle) has occurred.

2.4 Power models

With the previous section in mind, we will now construct a power model for the dynamic power consumption, and then apply this to compute the power of some simple combinational circuits. Assuming V and f are known and constant, we know that the power is proportional to

$$\sum_{g \in G} \alpha_g \cdot C_g \tag{2.6}$$

From this, it is clear that even if we knew the switching activity α_g for every gate, summing them alone would not give an accurate measure of total power, as we would also need the capacitance C_g of each gate.

In this work, we adopt the observation made by Chou et al., which states that the capacitance of a gate is roughly proportional to its fanout—that is, the number of gates its output drives [22]. Based on this assumption, we arrive at the expression

$$\sum_{g \in G} \alpha_g \cdot \text{fanout}_g, \tag{2.7}$$

where fanout_g denotes the fanout of gate g. This simplified expression maintains proportionality to equation 2.6, and is therefore also directly linked to the dynamic power in equation 2.5. Proportionality allows for comparisons—for example, if circuit A yields a value twice that of circuit B for this equation, then its dynamic power consumption is also twice as high.

This sum of fanout multiplied by switching activity will serve as our working power model throughout the remainder of this work. While it technically represents a capacitance-related term proportional to power, we refer to it simply as power for convenience. This simplification is justified by our focus

on relative comparisons between circuits or input distributions, rather than on absolute power values.

Given a gate-level circuit description and corresponding input vectors, this expression can be readily evaluated. For example, consider Figure 2.4, which shows our previously introduced simple circuit. We apply a transition on input a from F to T, while keeping input b constant at T. This causes node c to toggle from T to F, and node d from F to T, resulting in toggles at gates a, c, and d. Since a drives two gates (fanout of 2), and c and d each drive one, the total estimated power is $2 + 1 + 1 = 4$ for this particular vector pair.

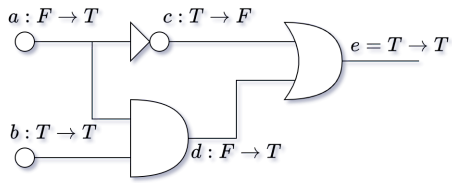


Figure 2.4: Example power computation with scalar values.

So far, we have assumed idealized gates that respond instantaneously to input changes. This is known as a *zero-delay* model, where each gate switches at most once per input change. In reality, however, gates have finite delays, which can lead to unintended transient transitions known as *glitches*. These extra transitions occur because gate outputs may not update simultaneously. A classic example is shown in Figure 2.5, where, logically, the output should remain at F. But when the input changes from F to T, the inverter momentarily delays its transition, causing both inputs of the AND gate to briefly be T, which flips the output to T before it returns to F. As a result, the number of transitions in one cycle can be greater than one. This introduces another dimension to

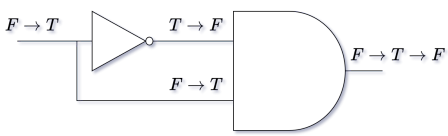


Figure 2.5: Glitching example.

consider: the delay model. We have already discussed the zero-delay model, where gates switch instantaneously, meaning no glitches occur. Other models include the unit-delay model, in which all gates have the same fixed delay, and the variable-delay model, where each gate may have a different delay.

To summarize the power model, we examine a combinational circuit with n inputs, where the inputs are updated instantaneously at the beginning of each clock cycle. During the cycle, and before the next update, each gate in the circuit may undergo several transitions between Boolean values, depending on the delay model chosen. The power consumed by a gate is proportional to the number of these transitions, scaled by its capacitance, which we approximate

using the fanout of the gate. Therefore, the overall power consumption can be expressed as a function of two consecutive input vectors, $\vec{v}^0$ and $\vec{v}^1$, representing the inputs at the beginning of successive clock cycles. In the zero-delay case, we have:

$$P(\vec{v}^0, \vec{v}^1) = \sum_{g \in G} \text{fanout}_g \cdot (g(\vec{v}^0) \oplus g(\vec{v}^1)), \tag{2.8}$$

where $g(\cdot)$ represents the Boolean function computed computed by the circuit between the input and (including) gate g, and $\oplus$ denotes the XOR operation, indicating whether the output of gate g changes between the two input vectors.

Taking glitches into account, our equation for P becomes more complicated. Instead of having a single potential transition $g(\vec{v}^0) \oplus g(\vec{v}^1)$, we may now have multiple intermediate glitch values g^a, g^b, g^c that depend on both $\vec{v}^0$ and $\vec{v}^1$. For simplicity, say we only have one intermediate value g^a. Then there are two potential transitions $g(\vec{v}^0) \to g^a(\vec{v}^0, \vec{v}^1)$ and $g^a(\vec{v}^0, \vec{v}^1) \to g(\vec{v}^1)$, so we obtain the formula:

$$P(\vec{v}^0, \vec{v}^1) = \sum_{g \in G} \text{fanout}_g \cdot \big((g(\vec{v}^0) \oplus g^a(\vec{v}^0, \vec{v}^1)) + (g^a(\vec{v}^0, \vec{v}^1) \oplus g(\vec{v}^0)\big) \tag{2.9}$$

With this, we can compute the power of changing from some input vector $\vec{v}^0$ to some other input vector $\vec{v}^1$, each of them n bits wide, meaning we have an input space of 2^{2n}.

To compute the average power, we would need to consider the entire input space of vector pairs. For a circuit with n inputs, this results in 2^n possible input combinations, and thus 2^{2n} input pairs to examine. This exponential growth makes a brute-force approach impractical. Fortunately, formal methods provide powerful techniques for efficiently exploring large input spaces. In the following sections, we explore how to leverage these tools.

2.5 Boolean satisfiability (SAT)

The Boolean satisfiability problem (SAT) asks whether there exists an interpretation that satisfies a Boolean formula. In other words, given a Boolean formula f that depends on variables $\vec{v}$, it asks whether we can find an assignment $\vec{m}$ of the variables of $\vec{v}$ such that $f(\vec{m}) = T$. A satisfying assignment is called a *model*. If there exists a model to a formula, the formula is satisfiable, and otherwise unsatisfiable. For example, formula

$$f = (a + b + c)(\overline{a} + b)$$

has a model $(a = F, b = F, c = T)$, meaning f is satisfiable. On the other hand, $g = a \cdot \overline{a}$ is unsatisfiable.

SAT is particularly interesting because it was the first problem proven to be NP-complete [23]. This means that there is no known algorithm that can solve all instances of SAT in polynomial time, and yet, any problem in the class NP can be reduced to SAT in polynomial time. In other words, if we could efficiently solve SAT, we could efficiently solve all problems in NP. This

result makes SAT a fundamental problem in computational complexity theory and a common focus in areas such as algorithm design, optimization, artificial intelligence, and formal verification.

In the context of combinational circuits, SAT can be viewed as the problem of determining whether there exists an input vector that causes the circuit to produce an output satisfying a given condition. If the circuit has n input bits, then there are 2^n possible input combinations to consider. To illustrate the scale of this, suppose the circuit has 270 input bits, which is still a rather small circuit. A rough calculation shows:

$$2^{270} = (2^{10})^{27} \approx (10^3)^{27} = 10^{81},$$

which is roughly the estimated number of atoms in the observable universe. This highlights just how quickly the number of input combinations grows and why the problem becomes computationally challenging.

Although SAT problems are inherently challenging, SAT solvers have significantly advanced over the years, with modern solvers capable of handling instances involving hundreds of thousands of variables. A foundational milestone in this progress was the DPLL algorithm [24], which introduced a systematic method for variable selection and efficient propagation of their implications. This approach was further enhanced in the GRASP solver, which pioneered conflict analysis techniques that generate new clauses to prevent recurring conflicts [25]. This strategy, now known as conflict-driven clause learning (CDCL), underpins nearly all contemporary SAT solvers.

Each year, the SAT competition is organized, where developers and researchers submit their solvers to compete against each other. This event showcases the latest advancements in SAT-solving techniques and provides a platform for comparing the performance of different solvers on a variety of complex problems. The competition has played a significant role in driving innovation in the field of automated reasoning and optimization[1]. For a general overview of modern SAT solving we recommend the handbook of satisfiability [26].

2.6 Binary Decision Diagrams (BDDs)

A Binary Decision Diagram (BDD) is a data structure designed for the efficient representation and manipulation of Boolean formulas [27]. In a BDD, a Boolean function is expressed as a directed acyclic graph (DAG), where each non-terminal node corresponds to a Boolean variable, and its outgoing edges represent the two possible values of the variable ($a = F$ or $a = T$). The terminal nodes are the constants F and T. Throughout this work, the term BDD refers specifically to Reduced Ordered BDDs (ROBDDs). In this subset, variables appear in a fixed order along every path, and redundant nodes are eliminated. As a result, ROBDDs provide a canonical representation of Boolean functions, meaning that equivalent functions have identical ROBDD representations.

[1] Information regarding the International SAT Competition can be found at `https://satcompetition.github.io/`

Figure 2.6 illustrates the BDD representation of the Boolean function $ab+a\bar{c}$ with variable ordering a, b, c. Each node in the diagram is labeled with its corresponding variable. The low child (resulting from setting the variable to 0) is connected via a dotted edge, while the high child (resulting from setting the variable to 1) is connected via a solid edge.

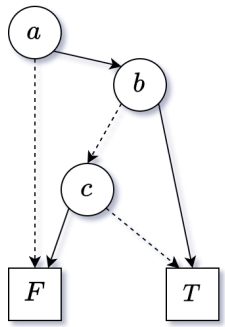


Figure 2.6: BDD representation of the Boolean formula $ab + a\bar{c}$.

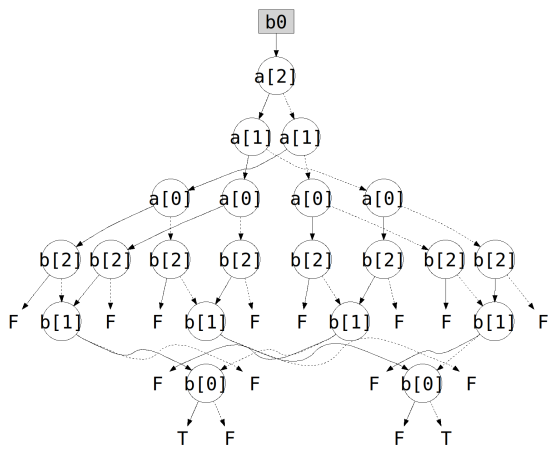
While constructing a BDD from a Boolean formula is generally NP-hard, and the resulting graph may grow exponentially in the worst case, BDDs remain efficient for many practical applications. In cases where the constraint exhibits structural regularity or redundancy, BDDs often provide a compact and manageable representation.

As previously noted, BDDs require that variables follow a fixed order along every path. Selecting an efficient variable ordering is critical, as it has a significant effect on the size of the BDD. For example, Figure 2.7(a) shows the BDD for the equality of three-bit wide vectors a and b using a suboptimal variable order. In contrast, Figure 2.7(b) presents a much more compact representation achieved by interleaving the bits of a and b. This optimized ordering reduces the number of BDD nodes[2] from 20 to 8. When scaling up to 16-bit values for a and b, the improved ordering brings the node count down from 196,604 to just 47.

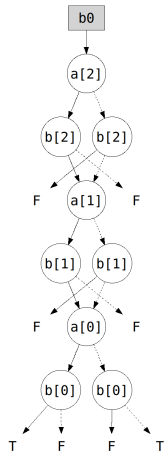
A substantial amount of research has focused on the challenge of variable ordering, with particular emphasis on automatic dynamic ordering techniques. The most widely adopted method is *sifting*, in which each variable is systematically repositioned within the ordering to identify the placement that minimizes the BDD size [28]. Many BDD libraries implement dynamic reordering strategies, which play a crucial role in preventing exponential blowup and thereby making BDDs practical for applications such as formal verification. Nonetheless, there exist functions—such as integer multiplication—whose BDD representations are inherently exponential regardless of ordering. In such cases, alternative representations or techniques should be explored.

Given a Boolean formula represented with a BDD, determining satisfiability is trivial, since all unsatisfiable formulas are represented as the constant F.

[2]Measured using `bv_size` in VossII.



(a) Bad variable order.



(b) Good variable order.

Figure 2.7: BDDs for the equality of three-bit wide vectors a and b with bad and good variable ordering.

Determining satisfiability therefore involves just checking whether $f = F$.

2.7 VossII

As stated, this thesis explores this intersection between formal methods, specifically symbolic model checking, and probabilistic analysis. For this purpose we use the **VossII** platform, a hardware design and verification system.

The original Voss formal verification system was developed at the University of British Columbia between 1990 and 1995 as an evaluation-based decision procedure for the HOL theorem prover. Initially, it integrated a lambda evaluator, an BDD package, and a symbolic trajectory evaluation (STE) engine. Over time, Voss evolved into a more user-friendly system with a Hindley-Milner type system, making it a powerful scripting, specification, and implementation language for formal verification. Its versatility led to widespread adoption, particularly at Intel (renamed Forte), where it became the backbone of formal equivalence verification and execution unit verification for two decades. Despite enhancements for large-scale industrial use, its core approach remained robust. The VossII system is a re-implementation of Forte, released as open-source under the Apache 2.0 license and available at GitHub[3].

The meta-language used in VossII is called *fl*, a functional language with lazy evaluation semantics. As such, it should be quite familiar and readable to users of other lazy functional languages like Haskell. To illustrate the language, Listing 2.1 presents four different implementations of the n-th Fibonacci number. The first example relies on basic recursion with an if-then-else structure. The second employs pattern matching and demonstrates that, due to laziness, the

[3]https://github.com/TeamVoss

error expression is never evaluated. These initial implementations suffer from exponential time complexity due to re-evaluating results, which the third version addresses by manually caching results in a list. This also introduces other basic operations, such as `el` for list indexing, the cons operator `:`, and an alternative if-then-else syntax: `condition => then | else`. The final implementation uses memoization, which is enabled by replacing `letrec` with `cletrec`. With memoization, once a function is evaluated, its result is stored in a hash table, so that subsequent calls with the same arguments return the cached result.

Listing 2.1: Four implementations of the Fibonacci sequence in VossII.

```
letrec fib1 n =
  IF n == 0 THEN 0 ELSE
  IF n == 1 THEN 1 ELSE
  fib1 (n-1) + fib1 (n-2);

letrec fib2 0 = 0
/\     fib2 1 = 1
/\     fib2 n =
  let unevaluted_expression = error "message" in
  fib2 (n-1) + fib2 (n-2);

let fib3 n =
  letrec rec m results =
    m > n => results |
    let new = el 1 results + el 2 results in
    rec (m+1) (new:results) in
  let results = rec 2 [1,0] in
  n==0 => 0 | hd results;

cletrec fib4 n =
  n <= 1 => n |
  fib4 (n-1) + fib4 (n-2);
```

As a verification platform, VossII offers extensive support for BDDs and SAT. We introduce these next.

2.7.1 BDDs in VossII

In VossII, every value of type *bool* is a Boolean formula built as a BDD. Such a value can have the constant values T and F for true and false, or be a function such as $\overline{a}+b$, where a and b are Boolean variables. In fact, all Boolean formulas in VossII are maintained as BDDs. This treatment of BDDs as first-class objects makes VossII ideal for our purposes.

A Boolean variable can be created with the `variable` function, which accepts a string as input and returns a Boolean formula containing a variable with the given name. For example, `let myvar = variable "a";` defines a variable *myvar* of type *bool* (i.e., a BDD) representing the Boolean formula a. To create $f = (a+b+c)(\overline{a}+b)$, we can write:

```
let a = variable "a";
let b = variable "b";
let c = variable "c";
let f = (a OR b OR c) AND (NOT a OR b);
```

When printing `f`, we obtain the expression $b + \overline{a}c$, which is a simplification of the original formula.

The `depends` function can be used to retrieve a list of the variables that an expression depends on. For example, `depends f` returns `["a", "b", "c"]`. Note that the result is a list of *strings* representing variable names, not *bool* values; `depends` provides the names of the variables, not the corresponding Boolean formulas.

VossII supports dynamic reordering of BDDs, which is automatically triggered when their size exceeds a certain threshold. It uses the standard sifting algorithm, common in many BDD libraries. Additionally, users can specify a custom variable ordering using the `var_order` function, as illustrated in Section 2.7.4.

2.7.2 SAT in VossII

Constructing a BDD for a Boolean formula just to check satisfiability is inefficient—building the BDD itself is often more complex than the satisfiability check. To address this, VossII offers an alternative datatype called *bexpr*, which represents Boolean expressions as AND-inverter graphs. These can be built in time linear to the size of the Boolean formula. Once a *bexpr* is created, satisfiability can be checked using a SAT solver. For this, VossII uses MiniSAT [29], which is a well-known CDCL solver.

Listing 2.2 shows a simple example using *bexprs*. Each logical operation on *bool*s has a corresponding *bexpr* version prefixed with a "b"—for example, `AND` becomes `bAND`. To check satisfiability, VossII uses the `bget_model` function, which takes a list of *bexprs*, assumes their conjunction, and attempts to find a satisfying assignment within a given timeout (in seconds). It returns a list of variable assignments if satisfiable. In the first case, we check satisfiability of $a \cdot b$, which requires both to be T, and *res1* correctly returns `[("a", bT), ("b", bT)]`. The second formula is unsatisfiable, indicated by an empty result list.

Listing 2.2: Small demonstration of bexprs and SAT.

```
let a = bvariable "a";
let b = bvariable "b";
let ex1 = a bAND b;
let ex2 = bNOT (a bAND b);
let res1 = bget_model [ex1] 60;
let res2 = bget_model [ex1 bAND ex2] 60;
```

2.7.3 Symbolic simulation with STE

Symbolic execution, commonly referred to as symbolic simulation in hardware contexts, is a technique where symbolic values represent variables during system simulation instead of using concrete data. This enables exploration of multiple execution paths simultaneously, helping to identify errors or vulnerabilities in hardware or software without needing to test every possible input.

In VossII, symbolic simulation is available through the `STE` command. As input it takes a circuit and an `ant` list, which defines input stimuli, along with other parameters not relevant to this work. The `ant` list contains tuples of five values (w, nd, v, f, t), which specify that when w holds, node nd should take the value v from time f until, but not including, time t.

Listing 2.3 demonstrates an example of using `STE` to simulate an OR-gate $c = a + b$. In this case, `ant` sets node a to the constant F and node b to the Boolean function $f(b) = b$. Both nodes have their first argument set to T, meaning the stimuli are applied unconditionally, and they are applied during phase $[0, 1)$, i.e., in phase 0. This example uses both scalar and symbolic simulation.

Listing 2.3: Tiny example of symbolic simulation with STE.

```
let ant = [
  (T, "a", F, 0, 1),
  (T, "b", variable "b", 0, 1)
];
let ste = STE "-e" my_or_gate [] ant [] [];
let out = get_trace_val ste "c" 0;
out;
```

We expect the value b to appear at the output since $F + b = b$. To retrieve the value of the output node c at phase 0, we use the `get_trace_val` function, which returns `[(b, b')]`. The result is a pair of values because STE implements symbolic trajectory evaluation [30], encoding the node's value using dual-rail (H,L) logic to represent both normal and error states. For this work, we do not require this feature, so we can simply examine the H value.

VossII also provides syntactic sugar for defining `ant`, as shown in Listing 2.4. In this case, we simulate a 4-bit bitwise AND operation and define input vectors. Internally, this corresponds to the tuples we saw earlier. The a input is set to be 3 in the first phase and then symbolic, while the b input is given the scalar value 5 for the first phase and then also symbolic in the second phase.

Listing 2.4: Symbolic simulation with vectors.

```
let ant =
  "a[3:0]" is 3 for 1 phase followed_by
              "a[3:0]" for 1 phase
  and
  "b[3:0]" is 5 for 1 phase followed_by
              "b[3:0]" for 1 phase;
let ste = STE "-e" and4 [] ant [] [];
let out t =
  let vars = md_expand_vector "c[3:0]" in
  map (\v. fst (get_trace_val ste v t)) vars;
out 0;
out 1;
```

To obtain the output, we first need the names of the output nodes, which we get by using `md_expand_vector` to expand `"c[3:0]"` into the list `["c[3]",`

`"c[2]", "c[1]", "c[0]"`. We then apply the `get_trace_val` function to each of these nodes, as before, but this time using the `fst` function to disregard the dual-rail encoding. In the first phase, the output is $[F, F, F, T]$, which correctly represents the bitwise AND of 3 ($[F, F, T, T]$) and 5 ($[F, T, F, T]$). In the second phase, the output is $[[a[3] \cdot b[3]], ...[a[0] \cdot b[0]]]$, as expected.

Internally, `STE` uses *bool*s, that is, BDDs, for simulation. It also provides `bSTE` for *bexpr*-based simulation, where the results can be passed to a SAT solver. The interface for `bSTE` is similar to `STE`, with the main difference being the use of *bexpr* instead of *bool* and the absence of syntactic sugar for defining `ant`.

2.7.4 Illustrative example: functional verification of adders

This section provides an illustrative example that brings together several of the techniques explored thus far in this thesis. Using VossII, we will verify the correctness of an adder circuit by applying SAT, BDDs, and symbolic simulation. In the process, we also highlight the impact of variable ordering on BDD performance.

Suppose we are given two 32-bit adders: a Ripple Carry Adder (RCA), which we assume to be correct, and a Carry-Select Adder (CSA), whose correctness we want to verify. The verification involves checking that, for all possible input combinations, the CSA produces the same output as the RCA. For simplicity, we will omit the carry-in and carry-out bits from consideration.

As an initial step, we perform basic scalar verification by applying a few simple input values and checking that the outputs match the expected results. Specifically, we will verify that the adder computes $12+3 = 15$ and $37-10 = 27$ correctly. We set up the input stimuli:

```
let ant =
    "a[31:0]" is 12 for 1 phase followed_by
                 37 for 1 phase
  and
    "b[31:0]" is 3 for 1 phase followed_by
              (-10) for 1 phase;
let ste = STE "-e" csa_ckt [] ant [] [];
```

After running the simulation, we extract the sum bits as follows:

```
let sum t =
  let nds = md_expand_vector "sum[31:0]" in
  map (\nd. fst (get_trace_val ste nd t)) nds;
bl2int (sum 0); // returns 15
bl2int (sum 1); // returns 27
```

Here, the `sum` function collects each bit of the sum into a Boolean list *bl*, which we then interpret as an integer. As expected, the resulting values are 15 and 27. Verification can also be performed during simulation by specifying a consequent in the STE run. However, for this demonstration, we choose to verify the results manually.

To verify all possible input combinations, we would need to check all $2^{32+32} = 2^{64}$ input pairs—an approach that is clearly infeasible using scalar values as we just did. Instead, we turn to BDD-based verification. Our approach is to simulate both the RCA and CSA using symbolic inputs, extract the Boolean functions representing each output bit, and then check that they are equal.

We begin by defining a new antecedent, where the inputs are driven with symbolic inputs rather than scalars:

```
let antecedent =
    "a[31:0]" is "a[31:0]" for 1 phase
  and
    "b[31:0]" is "b[31:0]" for 1 phase;
let rca_ste = STE "-e" rca_ckt [] antecedent [] [];
let csa_ste = STE "-e" csa_ckt [] antecedent [] [];
```

Running the simulation for the RCA completes in 24 ms, which is reasonably quick. In contrast, the simulation for the CSA takes significantly longer. This slowdown is due to the large number of multiplexers in carry-select adders, making them particularly sensitive to variable ordering. To address this, we stop the simulation and apply a new variable ordering that interleaves the bits of a and b:

```
let as = md_expand_vector "a[31:0]";
let bs = md_expand_vector "b[31:0]";
var_order (interleave [as,bs]);
```

After applying this ordering, we rerun the simulations and observe that it executes in roughly the same time as the RCA. Once the simulation is complete, we can easily verify that the outputs represent the same function:

```
let sum ste =
  let nds = md_expand_vector "sum[31:0]" in
  map (\nd. fst (get_trace_val ste nd 0)) nds;
let rca_sum = sum ste_rca;
let csa_sum = sum ste_csa;
let equal = csa_sum == rca_sum; // returns T
```

We conclude this section by verifying the adder using the built-in SAT solver. To do this, we must run the simulator using *bexprs* as the underlying Boolean representation instead of BDDs. This only requires a slight modification of the setup code:

```
let bant =
    let nd2term nd = (bT, nd, bvariable nd, 0, 1) in
    let as = md_expand_vector "a[31:0]" in
    let bs = md_expand_vector "b[31:0]" in
    (map nd2term as)@(map nd2term bs);
let rca_ste = bSTE "-e" rca_ckt [] bant [] [];
let csa_ste = bSTE "-e" csa_ckt [] bant [] [];
let sum ste =
  let nds = md_expand_vector "sum[31:0]" in
  map (\nd. fst (bget_trace_val ste nd 0)) nds;
let rca_sum = sum ste_rca;
let csa_sum = sum ste_csa;
```

To check for equality, we first create a list by combining the results with XOR, meaning the i-th entry in the list represents the *inequality* of the i-th bits of the results. If the disjunction of all such inequalities is *unsatisfiable*, the equality holds:

```
let inequality_list =
  map (\(k,r). k bXOR r) (zip ksa_sum rca_sum);
let any_bit_different =
  accumulate (defix bOR) inequality_list;
let sat_res = bget_model [any_bit_different] 10;
let equal = empty sat_res;
```

`bget_model` calls the SAT solver, and we interpret the empty result as the formula being unsatisfiable. We again find that the adders are equivalent. Moreover, the result is computed an order of magnitude faster than with BDDs.

With this, we have seen how VossII can be used for verification, utilizing symbolic simulation with BDDs and SAT.

2.7.5 Parametric substitution

Parametric substitution provides a way to restrict input vectors to those satisfying a particular constraint, guaranteeing that all valid vectors are included (completeness) and no invalid ones are possible (soundness). The method constitutes a cornerstone in formal verification based on symbolic simulation [31]. It is useful for verifying properties that must hold under a specific precondition on the inputs, such as $P \implies Q$. Parametric substitution eliminates false alarms, as vectors that violate P are never generated and thus cannot lead to a violation of Q. At the same time, any "bad" vectors—those that satisfy P but violate Q—can still be generated, ensuring that the verification is comprehensive.

VossII supports parametric substitution with the `param` function, whose functionality can be formalized as follows. Let $\mathbb{B}^n$ be the set of Boolean vectors of length n, and S be the subset of $\mathbb{B}^n$ that satisfies a given constraint $C : \mathbb{B}^n \to \mathbb{B}$, which we assume is satisfiable. Applying `param` to C creates a parametric substitution $(param\ C) : \mathbb{B}^n \to S$, which (1) only maps to values that satisfy C, as specified by the type signature, and (2) is surjective, meaning that any value that satisfies S can be obtained.

For example, C could enforce a mutual exclusion (mutex) condition, meaning that exactly one bit in the vector is set. Valid inputs under this constraint would include $\langle F, F, F, T\rangle$ and $\langle F, T, F, F\rangle$, while an input like $\langle F, T, F, T\rangle$ would be invalid. We can define the constraint and parametric substitution as:

```
let a = bv_variable "a[3:0]";
let C = bv_ones a = '1;
param C;
```

which returns a substitution

$$\langle a[3],\ \overline{a[3]}\,a[2],\ \overline{a[3]}\,\overline{a[2]}\,a[1],\ \overline{a[3]}\,\overline{a[2]}\,\overline{a[1]}\rangle. \tag{2.10}$$

Any substitution for the variables $a[3]$, $a[2]$, and $a[1]$ will yield a mutex input vector, and each of the four such vectors have some substitution that generates it. Conceptually, we can now construct a combinational circuit that implements these four Boolean functions, effectively mapping the variables to the circuit inputs, as shown in Figure 2.8.

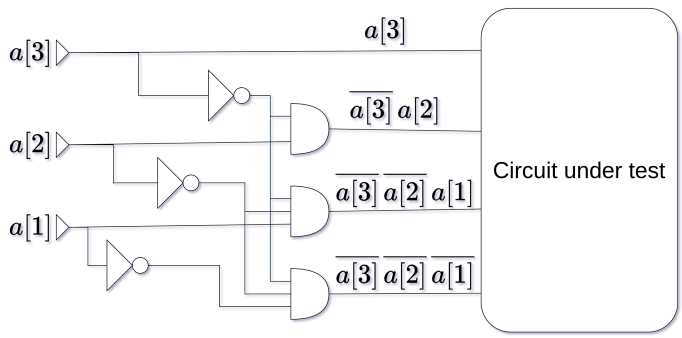


Figure 2.8: A circuit implementing a parametric substitution for the mutex condition.

2.8 Model counting

SAT addresses the question of whether there exists an assignment that satisfies a given Boolean formula, and returns such an assignment if one exists. A closely related—but more difficult—problem is model counting (also known as #SAT), which asks *how many* satisfying assignments exist. For example, there are three models to the formula $a + b$ for variables a and b. Model counting is strictly harder than SAT: if we know the total number of satisfying assignments, determining satisfiability is trivial—we simply check if the count is zero. However, the reverse does not hold. Knowing that there is at least one solution offers little insight into the total number of solutions, which could be exponentially large. Model counting is widely recognized as the prototypical #P-complete problem, serving as a central representative of the class due to its foundational role [26].

The second paper in this thesis makes substantial use of efficient model counting, which is crucial for the techniques it presents. One of the main advantages of BDDs is their support for model counting in time linear to the size of the diagram, making them especially appealing for this purpose. In the VossII system, this functionality is provided by the `truth_cover` function, which is described in detail in Section 2.8.1. An alternative approach is SAT-based solutions, explored in Section 2.8.2.

Model counting is closely related to probabilistic inference, which can often be reduced to weighted model counting (WMC) [32]. In WMC, each Boolean variable is assigned weights for T and F values, and the total weight of a formula is the sum over all models, where the weight of each model is the product of its literal weights:

$$\mathrm{WMC}(f, w) = \sum_{m \in M(f)} \prod_{v \in m} w(v), \tag{2.11}$$

where f is a Boolean formula, w maps literals to weights, and $M(f)$ is the set of models of f. As a simple example, let $f = ab + a\bar{b}$, with weights $w(a) = 2$, $w(b) = 3$, and $w(\bar{b}) = 4$, for which we get:

$$\mathrm{WMC}(f, w) = w(a)w(b) + w(a)w(\bar{b}) = 2 \cdot 3 + 2 \cdot 4 = 14 \tag{2.12}$$

We will note the similarity to WMC when discussing probabilistic techniques in Section 2.9.2. The difference is whether the weights are provided as integers (as they are here) or as probabilities.

2.8.1 Truth cover: model counting with BDDs

VossII features the built-in function `truth_cover`, which computes the number of satisfying assignments to a formula f represented as a BDD. In other words, it is model counting for BDDs. Along with the BDD, `truth_cover` requires a list of the variables the formula (nominally) depends on and returns the number of assignments to those variables that satisfy the BDD. A simple implementation is:

```
letrec tc_slow vars f =
  IF vars = [] THEN
    IF f == F THEN 0 ELSE
    IF f == T THEN 1 ELSE
    error "Undefined."
  ELSE
    val (v:vs) = vars in
    let H = substitute [(v,T)] f in
    let L = substitute [(v,F)] f in
    (tc_slow vs H) + (tc_slow vs L)
```

To illustrate its use we consider the following example, where we note the critical importance of the list of variables:

```
VARS "a b";
tc_slow ["a", "b"] (a OR b); // returns 3
tc_slow ["a", "b"] T; // returns 4
tc_slow ["a", "b"] F; // returns 0
tc_slow ["a"] b; // Undefined, crashes
```

Listing 2.5 shows a much more efficient implementation, `tc`, which incorporates memoization. It also makes use of `top_cofactor` to identify the top variable and performs a linear search through the variable list to locate it. Any variables that are skipped in this process are considered free and can be omitted in the recursive calls—each such omission requiring us to double the result to account for the free variable. Out of these optimizations, memoization remains the most critical performance enhancement. This version closely resembles the built-in `truth_cover` function, although the latter is implemented in C for increased performance.

Listing 2.5: Better implementation of truth cover.

```
cletrec tc vars f =
  IF f == F THEN 0 ELSE
  IF f == T THEN 2**(length vars) ELSE
  val (v,H,L) = top_cofactor f in
  let idx = find_first (\s. s = v) vars in
  let vars' = butfirstn idx vars in
  let m = 2**(idx-1) in
  m*((tc vars' H)+(tc vars' L));
```

Thanks to memoization, the complexity of `truth_cover` is $O(n)$ for a BDD with n nodes.

2.8.2 SAT-based model counting

SAT-based techniques have been extensively studied in the literature. While they have not yet been applied in our current work, we include them here as related methods with potential for future exploration.

One of the earliest solvers for exact model counting was sharpSAT [33]. It performs a search over the solution space similar to CDCL-based SAT solvers but continues until all satisfying assignments are enumerated. To reduce redundant computation, sharpSAT employs component caching—analogous to memoization in `truth_cover`. While the paper also presents strategies for efficient cache management, we do not explore those details here. More recently, Sharma et al. introduced the probabilistic model counter GANAK [34], which can be seen as an extension of sharpSAT with probabilistic aspects and new heuristics.

Due to the computational complexity of the problem, SAT-based approximate techniques have also been developed, most prominently the ApproxMC solver [35]–[37]. The underlying idea is to randomly partition the solutions space using hashing techniques, typically XOR constraints, and then estimating

the number of models based on how many models are in a randomly chosen subset. XOR constraints are formed by selecting a random subset of the variables, and add the constraint that their parity should be F or T. For example, if we have variables a, b, c, and d, we could randomly pick variables a and d. We then randomly pick a parity T or F, say T, resulting in the constraint $a \oplus d = T$. Each such constraint reduces the set of satisfying assignment by half on average [38].

2.9 Average power estimation

With the basic formal verification methods covered, we now return to our power model from Section 2.4, and consider the problem of computing the *average* of this formula. This is given by

$$E[P] = E\left[\sum_{g \in G} \text{fanout}_g \cdot (g(\vec{v}^0) \oplus g(\vec{v}^1)\right], \tag{2.13}$$

where E denotes the expectation. For a finite number of outcomes, it is computed by summing each outcome multiplied by its probability:

$$\begin{aligned} E[P] &= x_1 p_1 + x_2 p_2 + \ldots + x_n p_n \\ &= \Pr\{P = 0\} \cdot 0 + \Pr\{P = 1\} \cdot 1 + \ldots + \Pr\{P = M\} \cdot M \\ &= \sum_{i \in [0,M]} \Pr\{P = i\} \cdot i \end{aligned} \tag{2.14}$$

for some for some maximum power M. Clearly, this expression heavily depends on the input probabilities, that is, the probability of seeing each input vector pair $\vec{v}^0$ and $\vec{v}^1$. Note that if each vector is equally likely, this can be viewed as a model counting problem:

$$\begin{aligned} E[P] &= \sum_{i \in [0,M]} \Pr\{P = i\} \cdot i \\ &= \sum_{i \in [0,M]} \frac{MC(P = i)}{2^n} \cdot i \\ &= \frac{1}{2^n} \sum_{i \in [0,M]} MC(P = i) \cdot i \end{aligned} \tag{2.15}$$

where $MC(\cdot)$ denotes model counting and n is the input vector width. The assumption of uniform input probabilities is rarely realistic in practice. To address this, we require a way to specify input probabilities—that is, the likelihood of observing each $(\vec{v}^0, \vec{v}^1)$ pair. One practical solution is to allow the user to provide these input vectors directly. This approach is known as vector-based power estimation in tools like Cadence Genus [39]. However, it comes with significant trade-offs: the quality of the estimation heavily depends on how representative the provided vectors are. Manually creating or sourcing realistic

input vectors can be time-consuming and error-prone, and poor coverage may lead to misleading power estimates. Moreover, if exhaustive vector generation is required, this approach does not scale well.

2.9.1 Approximate approach with Monte Carlo

If an approximate solution is sufficient, one of the most straightforward methods of computing equation 2.13 is using Monte Carlo simulation. For our use case, the approach is entirely straightforward: we sample $\vec{v}^0$ and $\vec{v}^1$ according to the chosen input distribution and simulate the circuit with these scalar vectors, and obtain a sample of the power distribution. This continues until some stop condition is reached. A commonly used stopping condition, proposed in prior work, is:

$$\frac{t_{\alpha/2} \cdot \sigma}{\mu \cdot \sqrt{N}} > \epsilon, \tag{2.16}$$

where $t_{\alpha/2}$ is the critical value from the t-distribution, σ is the sample standard deviation, μ is the sample mean, and ϵ is the target relative error [18], [40]. This criterion allows users to balance accuracy and computational effort by selecting a desired error margin in advance, which they found works well in practice. T-tests of this kind assume that the underlying distribution follows a Gaussian model, which has been found to hold true in many practical cases [18].

2.9.2 Probabilistic approaches

Monte Carlo approximation offers a powerful fallback when approximate solutions are acceptable. However, in this work, we focus on exploring the limits of exact methods, which is particularly important in security-sensitive applications. To this end, we consider probabilistic approaches. Here, users can specify the probability distribution over input transitions, i.e., the likelihood of each $(\vec{v}^0, \vec{v}^1)$ pair occurring. This is known as vector-free or probability-based power estimation [14], in contrast to vector-based estimation. However, since there are 2^{2n} possible pairs for n inputs, explicitly enumerating them quickly becomes infeasible. A more scalable approach is to define the probability of each individual input bit being high, which is known as *signal probability*. For example, if input bit a has $\Pr\{a = T\} = 0.3$, then it is low with probability 0.7. This bit-level probabilistic modeling has a long history in digital circuit analysis, dating back to as early as 1975 [41]. It has proven useful not only in power estimation but also in evaluating circuit reliability under uncertainty [42].

Using basic rules from probability theory, signal probability can be propagated through the circuit. For instance, $\Pr\{\overline{a}\} = 1 - \Pr\{a\}$ and $\Pr\{ab\} = \Pr\{a\} \cdot \Pr\{b\}$. Note however that the latter only holds when expressions a and b are not *correlated*. To show this, first define a simple recursive datatype `mybool` in Listing 2.6, along with corresponding signal propagation rules.

Listing 2.6: Basic datatype for Boolean functions and propagation rules for signal probability.

```
lettype mybool = True
               | Var {n :: string}
               | Not {b :: mybool}
               | And {b1 :: mybool} {b2 :: mybool};
let False = Not True;
let Or b1 b2 = Not (And (Not b1) (Not b2));
let sigprob_propagation {sp :: (string#float) list} b =
  letrec rec True = 1.0
  /\     rec (Var v) = assoc v sp
  /\     rec (Not b) = 1.0 - (rec b)
  /\     rec (And b1 b2) = (rec b1) * (rec b2)
  in rec b;
```

This method yields correct results under the assumption that each variable appears only once in an expression. For instance:

```
let a = Var "a"; let b = Var "b";
let e1 = And (Not a) b;
let sp = [("a", 0.4), ("b", 0.3)];
sigprob_propagation sp e1;
```

This returns $(1 - 0.4) \cdot 0.3 = 0.6 \cdot 0.3 = 0.18$, as expected. However, in the expression

```
let e2 = And a (Not a);
sigprob_propagation sp e2;
```

the function returns 0.24, though the expression $e2$ is always false and should return 0. One might wonder whether the function always overapproximates, but this is not the case. Consider:

```
let e3 = Or a (Not a);
sigprob_propagation sp e3;
```

Here, the result is 0.76, though the expression $e3$ is always true and should return 1.

This example highlights the issue of *correlation* between Boolean signals. In the context of digital circuits, this arises when we assign independent signal probabilities to the inputs, yet attempt to compute the probability of a signal deeper in the circuit that depends on multiple correlated paths. Correlation in this case translates to *reconvergent fanout*—a well-known challenge in probabilistic circuit analysis where signals split and later recombine, violating the assumption of independence [43].

We can resolve the correlations and obtain an exact solution by computing every satisfying assignment (model) for the expression, compute the probability of each model independently, and return the sum of these probabilities. In other words, we convert the formula to DNF, evaluate the probability of each term, and sum the results:

$$\Pr\{f = T\} = \sum_{m \in M(f)} \prod_{v \in m} w(v) \tag{2.17}$$

We note that this is the same formula as in WMC, mentioned in Section 2.8.

For instance, the formula $f = ab + bc$ can be expanded into DNF as

$$f = abc + ab\bar{c} + \bar{a}bc$$

Given $P(a) = 0.5$, $P(b) = 0.4$, and $P(c) = 0.3$, we compute

$$\begin{aligned} \Pr\{f\} &= P(a)P(b)P(c) + P(a)P(b)(1 - P(c)) + (1 - P(a))P(b)P(c) \quad (2.18) \\ &= 0.5 \cdot 0.4 \cdot 0.3 + 0.5 \cdot 0.4 \cdot 0.7 + 0.5 \cdot 0.4 \cdot 0.3 \\ &= 0.26 \end{aligned}$$

Technically, it is sufficient to compute the disjoint cover of the formula instead of the DNF [40]. We will not explore this further, and instead consider the use of BDDs in the probability computation, as demonstrated by Ghosh et al [44]. In VossII, we can write this as:

```
cletrec bdd2p sp bdd =
  IF bdd == F THEN 0.0 ELSE
  IF bdd == T THEN 1.0 ELSE
  val (v,H,L) = top_cofactor bdd in
  let prob = assoc v sp in
  let res_L = bdd2p sp L in
  let res_H = bdd2p sp H in
  (1.0-prob)*res_L + prob*res_H;
```

Since the implementation leverages memoization, the computation runs in time linear to the size of the BDD.

We note that if all signal probabilities are 0.5, we can write this as

```
let bdd2p_tc bdd =
    let deps = depends bdd in
    let num = truth_cover deps bdd in
    let den = 2**(length deps) in
    (int2float num) / (int2float den);
```

which is equivalent to equation 2.15. With these components in place, the power model can now be evaluated exactly, using the provided input signal probabilities.

2.9.3 Symbolic power computation

Combining the above methods with BDD-based symbolic simulation, we can compute the power consumption of a circuit exactly, provided the signal probabilities for the inputs. This approach of *symbolic power analysis* has previously been explored by Monteiro et al. [43]. In this section, we demonstrate how this can be implemented in VossII, using a 4-bit RCA as an example, shown in Figure 2.9. The triangles in the diagram represent artificial phase delays, which are included to demonstrate the appearance of glitches. These delays are not meant to represent realistic timing behavior and will not be explored further; for additional details, see [45]. In our framework, a phase represents one unit of simulation time, with 1000 phases corresponding to one full clock cycle.

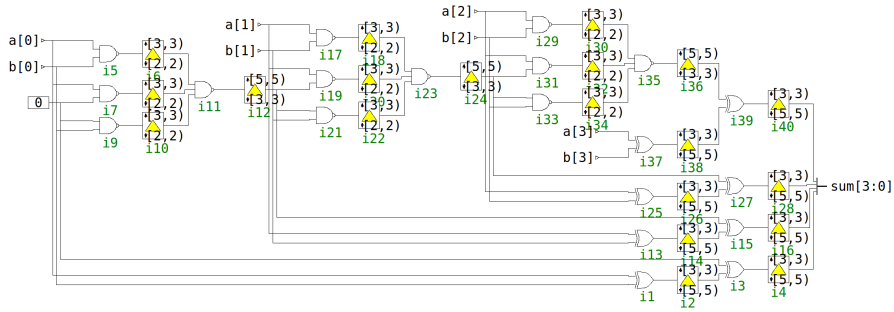


Figure 2.9: 4-bit RCA with inserted delays.

Given this circuit, we can set up the symbolic simulation as follows:

```
let all_nodes = md_expand_vectors (nodes ckt);
let input_nodes = md_expand_vectors (inputs ckt);
let ant =
  let input_stimuli nd =
    let val_before =
      variable nd in
    let val_after  =
      variable (nd^"_1") in
    [(T, nd, val_before, 0, 1000),
     (T, nd, val_after, 1000, 2000)] in
  flatmap input_stimuli input_nodes;
let ste =
  STE "-e" ckt [] ant [] [];
```

Once the STE run is complete, we implement the *nd_xors* function to obtain a list of toggle expressions for a node. We omit the implementation here, as it is lengthy and detracts from the main focus. With these expressions, we can compute the power as a bit-vector, as shown in Listing 2.7. This function implements equation 2.9.

Listing 2.7: Symbolically computing the total power.

```
let power_bv =
  let nd_power nd =
    let cap = int2bv (length (fanout ckt nd)) in
    let changes = map bdd2bv (nd_xors nd) in
    bv_mul cap (bv_sum changes) in
  bv_sum (map nd_power all_nodes);
```

For simplicity we assume uniform inputs, with which we can compute the probability of each outcome, that is, the value of the bit-vector, as shown in Listing 2.8. From this we can then compute the average power, which turns out to be 413.664. We can also plot the power as a distribution, shown in Figure 2.10.

Listing 2.8: Average power computation utilizing the bdd2p_tc from before.

```
let maximum = bv_max power_bv;
let xs = (0--maximum);
let ys = map (\x. bdd2p_tc (power_bv = int2bv x)) xs;
let products = map (\(x,y). (int2float x)*y) (zip xs ys);
let avg = accumulate fadd products;
```

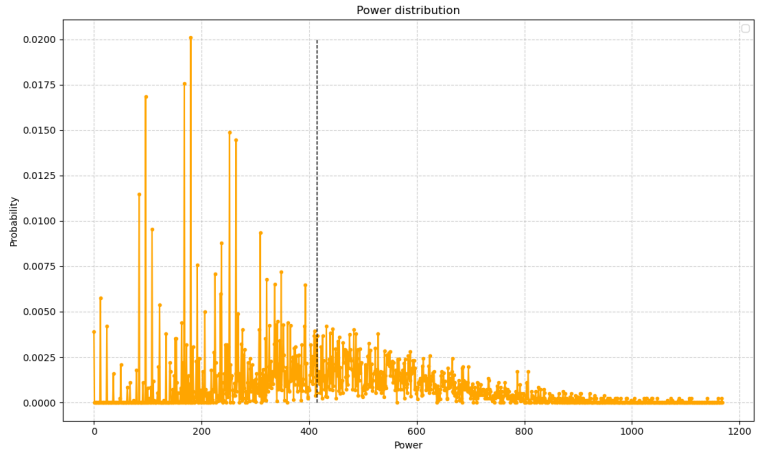


Figure 2.10: Power distribution for a 4-bit ripple-carry adder with the power on the x-axis and probability on the y-axis, assuming uniform inputs. For each value of x, the probability $y(x)$ is the number of is the vectors that makes the circuit use x power, divided by the total number of input vectors. The average is marked.

An important observation is that if we are only interested in the average power, then we do not need to compute the *power_bv* expression due to the linearity of expectation. Analytically, with a zero-delay model, we have

$$\begin{aligned} E[P] &= E\left[\sum_{g \in G} \text{fanout}_g \cdot \left(g(\vec{v}^0) \oplus g(\vec{v}^1)\right)\right] \\ &= \sum_{g \in G} \text{fanout}_g \cdot E\left[\left(g(\vec{v}^0) \oplus g(\vec{v}^1)\right)\right] \end{aligned} \tag{2.19}$$

meaning we can compute the activity for each node separately, and compute the sum and get the same result. Furthermore, if we have glitches, this holds

for each *transition expression*:

$$E[P] = E\left[\sum_{g \in G} \text{fanout}_g \cdot \left((g(\vec{v}^0) \oplus g^a) + (g^a \oplus g(\vec{v}^0))\right)\right] \tag{2.20}$$
$$= \sum_{g \in G} \text{fanout}_g \cdot \left(E\left[(g(\vec{v}^0) \oplus g^a)\right] + E\left[(g^a \oplus g(\vec{v}^0)\right]\right)$$

This lets us compute the average as follows, which is significantly faster:

```
let avg =
  let nd_avg nd =
    let cap = int2float (length (fanout ckt nd)) in
    let toggle_probs = map bdd2p_tc (nd_xors nd) in
    cap * (fsum toggle_probs) in
  fsum (map nd_avg all_nodes);
```

2.9.4 An aside: sequential circuits

While not something we will pursue further in this thesis, a natural and necessary extension of our work is to expand beyond purely combinational circuits and consider sequential circuits. The simplest such circuits, deterministic finite state machines, can be modeled by augmenting combinational logic with D flip-flops. The behavior of such circuits can be captured by Boolean functions $Output(\vec{v}, \sigma)$ and $NextState(\vec{v}, \sigma)$, where $\vec{v}$ denotes primary inputs and σ is the current state. These functions typically share logic and together form the circuit's combinational block, as visualized in Figure 2.11.

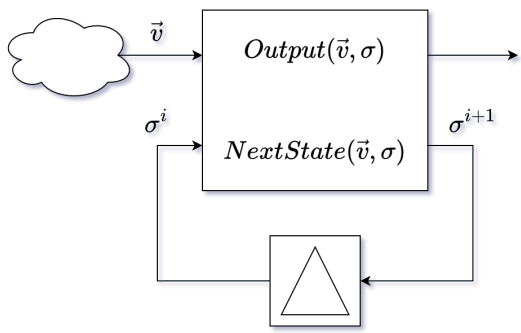


Figure 2.11: A deterministic finite state machine.

The average power consumption of this *combinational block* is given by:

$$\sum_{\sigma \in S} \Pr\{\text{in } \sigma\} \cdot E[P(\vec{v}, \sigma)],$$

where S is the set of all possible states and $E[P(\vec{v}, \sigma)]$ is the expected power in state σ. This analysis assumes no feedback from state to input, a simplifying

but significant limitation. To solve this, we need to compute the probability of being in each state. An exact solution can be obtained by solving the Chapman-Kolmogorov equations, solving for a steady-state distribution $\vec{v}$ satisfying $\vec{v} \cdot P = \vec{v}$ for stochastic next-state matrix P [40].

This warrants an example, for which we will use a tiny two-bit branch predictor circuit, whose behavior is captured in Figure 2.12. The circuit has a

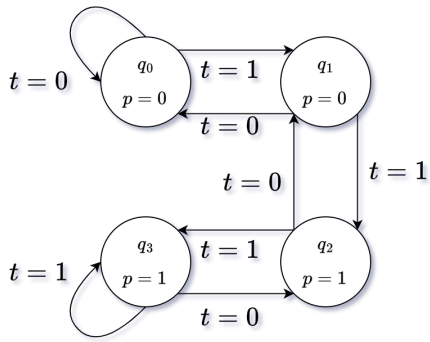


Figure 2.12: State transition graph for a two-bit branch predictor.

single input bit t (taken) and a single output bit p (prediction). The state is captured in two counter bits, increased when t is high and decreased when low. Assuming the input bit has signal probability p_t, the state transitions can be captured with the stochastic matrix

$$P = \begin{bmatrix} 1-p_t & p_t & 0 & 0 \\ 1-p_t & 0 & p_t & 0 \\ 0 & 1-p_t & 0 & p_t \\ 0 & 0 & 1-p_t & p_t \end{bmatrix} \tag{2.21}$$

where we interpret the row as the starting state and the column as the next state. Multiplying a current-state probability vector with P gives the probabilities for the next state, for example:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \cdot P = \begin{bmatrix} 1-p_t & p_t & 0 & 0 \end{bmatrix} \tag{2.22}$$

Solving for fix-point vector $\vec{v}$ in $\vec{v}P = \vec{v}$ (or $\vec{v}(P - I) = 0$) gives us a vector whose elements represent the probability of being in each state. This is a linear equation and can be solved with straightforward Gaussian elimination for small examples. With input probability $p_t = \frac{1}{2}$, we obtain $\vec{v} = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{bmatrix}$, i.e. each state is equally likely. If however the branch is taken often, say $p_t = 0.8$, we get $\begin{bmatrix} \frac{1}{85} & \frac{4}{85} & \frac{16}{85} & \frac{64}{85} \end{bmatrix}$, skewing the distribution towards the taken side.

A major challenge is scalability: with n flip-flops, the state space grows exponentially to 2^n, such that P becomes a $2^n \times 2^n$ matrix. Previous work has addressed this through approximation techniques [40], but we theorize that with symbolic methods, we can push the limits of an exact solution. We discuss this as a future work in Chapter 4.

2.9.5 The problem of correlations

Returning to our symbolic power computation, one key limitation of the approach so far is the treatment of input signal probabilities. While BDDs effectively model correlations within the circuit itself, we assume that the input signals are independent. In practice, this assumption often breaks down due to two types of correlation: *temporal* correlation—dependencies between the values of the same signal across clock cycles—and *spatial* correlation—dependencies between different input signals at a given time [40]. Fundamentally, both types of correlation reduce to the problem of accurately capturing joint probability distributions.

To illustrate temporal correlation, consider a signal a in two consecutive cycles: we denote its value in the first cycle as a, and in the second as A. We can assign signal probabilities $\Pr\{a = T\}$ and $\Pr\{A = T\}$ independently, but this ignores any statistical relationship between them. In real systems, these values may be strongly correlated. Empirical results show that neglecting temporal correlation can lead to power estimation errors ranging from 15 % to 50 % [46].

To address temporal correlation, *transition probabilities* can be used. For each bit, four probabilities P_{00}, P_{01}, P_{10}, and P_{11} are specified to describe the likelihood of each possible transition from one cycle to the next. Incorporating this into our symbolic method is straightforward: we need only adapt the `bdd2p` function to handle pairs of Boolean variables representing transitions. While this modification is conceptually simple, we omit the implementation details here.

While modeling temporal correlation improves our control over signal behavior across time, it does not capture spatial correlation—dependencies between different input signals in the same cycle, which are also important for accurate power estimation. One way to approximate spatial correlation is through correlation coefficients [47], which quantify pairwise dependencies between input signals. Correlation coefficients are also used in the field of reliability [42].

Recall from our earlier discussion on sequential circuits that the inputs to the combinational logic consist of both the primary inputs and the current state. Estimating power in this setting requires computing the probability distribution over all state bits—effectively modeling spatial correlation among them—before applying symbolic power computation.

The second paper of this thesis addresses techniques for managing these joint distributions, applicable to both scalar and symbolic simulation, as summarized in Section 3.2.

Chapter 3

Paper Summaries

3.1 Higher-order Hardware: Implementation and Evaluation of the Cephalopode Graph Reduction Processor

3.1.1 Motivation

The paper focuses on the domain of resource-constrained environments, such as those found in the Internet of Things (IoT). IoT devices, like other embedded systems, are typically designed with strict resource limitations. Energy efficiency is especially critical, as these devices are often powered by batteries and are expected to function for extended periods – often several years – without the need for battery replacement. Due to these strict energy consumption constraints, the compute power and memory available on these chips are highly limited. As a result, they are typically programmed using low-level languages such as C and C++, which offer minimal runtime overhead and direct access to hardware.

This simplicity comes at the expense of the abstractions and safety features provided by higher-level languages, particularly memory safety. This trade-off has already led to security vulnerabilities in IoT devices, with incidents including smart fridges exposing Gmail credentials [48], cars being remotely hijacked [49], and household gadgets being exploited to launch large-scale DDoS attacks [50].

As an initial illustrative example of how such issues can arise, we consider the small C program shown in Listing 3.1. This program contains a precomputed array, `primes`, which encodes the primality status of the first 256 natural numbers. When the array is indexed with a number a, the function `is_prime` returns 1 if a is prime and 0 otherwise. For instance, the first four entries of the array are $[0, 0, 1, 1]$, accurately reflecting the primality of the numbers $[0, 1, 2, 3]$.

In the `main` function, we try this for the number 131. We compile and run the program, and get

Listing 3.1: Prime number example.

```
static WORD primes[] = {
  0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0,
  0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1,
  0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1,
  0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0,
  0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1,
  0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
  0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0,
  0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
  0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0,
  0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0,
  0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
  0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
  0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1,
  0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0
};
WORD is_prime(WORD nr) {return primes[nr];}
void main(void) {
  WORD number = 131;
  if (is_prime[number])
    printf("131 is a prime.\n");
  else
    printf("131 is not a prime.\n");
}
```

```
131 is not a prime.
```

which is wrong. What caused the issue? It turns out that `WORD` is defined as a signed `char`, which is an 8-bit integer type capable of representing values in the range $[-128, 127]$. As a result, the value 131 is interpreted as -125, causing the program to index the array at position -125. This accesses a memory location outside the bounds of the array, which in this case happened to contain the value 0. Fortunately, this particular memory region did not contain any sensitive data, such as a password.

Simple mistakes like this can be detected by the compiler, provided that warnings are enabled. In this case, the variable `number` is assigned a value that exceeds the representable range of its declared type, allowing the compiler to identify the issue through static analysis and issue a warning. However, with more subtle or complex bugs, we may not be as fortunate.

For example, consider the program presented in Listing 3.2. It includes a function, `avg`, that calculates the average of an array of numbers, where the end of the array is indicated by the value 0. This time, we also incorporate a set of tests to help detect potential errors. In the `main` function, these tests are executed first, and the program will terminate if any of them fail. If all tests pass, execution proceeds to the primary logic, where we process a set of temperature readings from a hypothetical boiler. We want the program to shut down the boiler if the average temperature exceeds 55 degrees, which we

anticipate will occur for the given readings $(50, 60, 70)$. The expected average, 60, is also well within the representable range of a signed `char`. As before, we

Listing 3.2: Mean value example.

```
WORD avg(WORD* items) {
  WORD sum = 0, cnt = 0;
  while (*items != 0) {
    sum += *items;
    items++; cnt++;
  }
  if (cnt != 0) return sum / cnt;
  return 0;
}
void test_avg(void) {
  WORD arr[10];
  arr[0]=5; arr[1]=0;
  if (avg(arr) != 5) panic("FAIL!");
  arr[0]=20; arr[1]=30; arr[2]=40; arr[3]=0;
  if (avg(arr) != 30) panic("FAIL!");
  arr[0] = -20; arr[1] = -30; arr[2] = 10;
  arr[3] =  40; arr[4] =  30; arr[5] =  0;
  if (avg(arr) != 6) panic("FAIL!");
}
void main(void) {
  test_avg();
  printf("All tests passed.\n");
  // "Real" program starts here
  WORD temp[10];
  temp[0] = 50; temp[1] = 60; temp[2] = 70; temp[3] = 0;
  if (avg(temp) > 55)
    printf("Turn off boiler.\n");
  else
    printf("Keep boiler running.\n");
}
```

compile and run the program, which outputs

```
All tests passed.
Keep boiler running.
```

which, once again, is wrong. And the kitchen is on fire.

In this case, the issue is an integer overflow. Although the expected average value, 60, falls within the representable range, the intermediate value `sum` computed in the `avg` function becomes $50 + 60 + 70$, which exceeds the allowable range and causes an overflow. Errors like this are dynamic in nature and typically cannot be detected by the compiler.

In high-level languages, such errors are typically prevented by the language's built-in safeguards. Most of these languages raise an exception when attempting to access elements outside the bounds of an array, thereby preventing errors like the one in the first program. Additionally, languages such as Python and Haskell use arbitrary-precision arithmetic, which helps avoid issues like integer overflow.

High-level *functional* languages such as Haskell present a compelling option for programming embedded devices. They have demonstrated strong potential in enabling programming environments that support security and privacy guarantees without placing significant burdens on the developer [51], [52]. Like most high-level languages, Haskell offers memory safety, which helps prevent low-level programming errors. Its robust type system further aids in catching many mistakes at compile time. However, Haskell's particular emphasis on higher-order functions and polymorphism enhances opportunities for code reuse, and less code generally translates to a reduced likelihood of errors.

Unfortunately, there is a clear reason why microcontrollers are not typically programmed in Haskell: the size and complexity of its runtime system. In practice, this refers to the runtime system of the Glasgow Haskell Compiler (GHC), which is effectively the only widely used Haskell compiler. The GHC runtime system has proven challenging to deploy in resource-constrained environments, rendering it infeasible for use on small embedded devices [53]. Moreover, it relies on garbage collection, which complicates the ability to deliver the strict performance guarantees often required by real-time systems—a common demand in many embedded applications.

Instead of depending on a full-featured software runtime system, another approach is to design a specialized hardware architecture that can natively execute a high-level language, incorporating essential features like garbage collection directly into the hardware. The Cephalopode processor exemplifies this strategy, aiming to combine the safety and ease-of-use of high-level languages with the stringent resource limitations of embedded systems—prioritizing minimal energy consumption as its central design objective.

3.1.2 Design

Although the primary focus of the author was on the evaluation, we will briefly outline the design and implementation for completeness. For more information on the design of the processor, we refer to the PhD thesis of the first author [54].

The program execution model in Cephalopode is based on *combinator graph reduction*. In this approach, a program is compiled into a small set of primitive functions known as combinators, which are used to build a graph representing the program. Computation proceeds by systematically applying these functions—*reducing* the graph—until no further reductions are possible. For a detailed explanation, we refer the reader to Turner's original work [55]. A common notion regarding combinator graph reduction is that it is memory-intensive: a topic we will revisit later.

A helpful analogy is to think of combinators as opcodes: the high-level language is compiled into a sequence of these primitive operations, and execution involves examining the next opcode along with its arguments and performing the corresponding action.

Figure 3.1 provides a high-level overview of the Cephalopode architecture. At its core is the graph reduction engine, which traverses the graph and performs reduction steps, delegating arithmetic and logic operations to the ALU as needed. The combinator graph is stored in RAM and updated in-place

during execution, with memory management handled by the allocator.

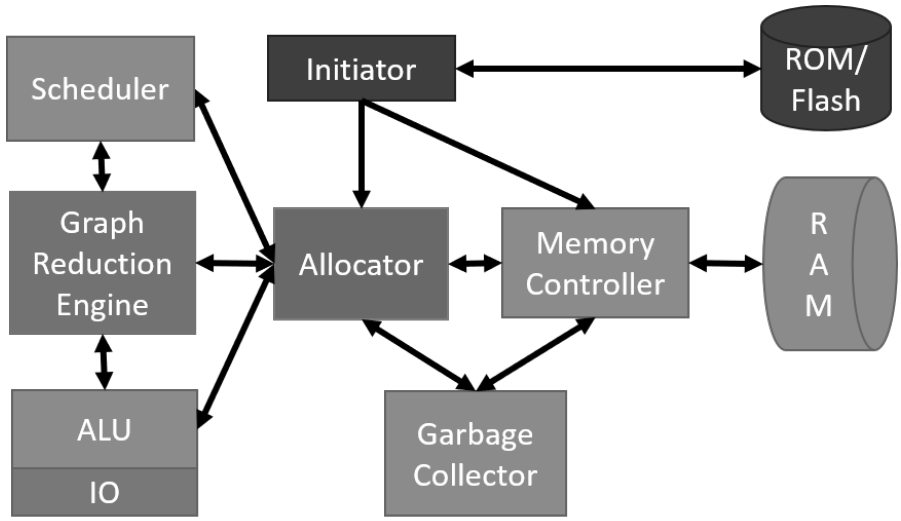


Figure 3.1: The architecture of Cephalopode.

One of the most notable components is the hardware-based garbage collection unit, which uses a tracing approach. Instead of employing a traditional stop-the-world collector—unsuitable for real-time embedded systems—Cephalopode performs garbage collection concurrent with program execution. This is made possible by the key insight that garbage remains garbage until it is freed [56], allowing collection to safely operate on a snapshot of the graph, with any newly created garbage handled in subsequent passes. Because of the snapshotting approach, memory usage is effectively doubled. However, we mitigate this overhead using a copy-on-write strategy to minimize unnecessary duplication. Further details can be found in the paper.

The final design point to mention is that of arithmetic precision. We recall from the example in Listing 3.2 that arithmetic overflow errors can be subtle, and that many high-level languages provide arbitrary-precision capabilities for avoiding this. In Cephalopode, this capability is provided in the hardware. All numbers are fixed-precision by default, becoming a arbitrary-precision automatically when needed. Fixed-precision numbers fit into one graph node, while arbitrary-precision integers are represented as a list of chunks. Essentially, this approach combines the speed of fixed-precision with the safety of arbitrary-precision. In the paper, we referred to this mechanism as multiple-precision, though we later adopted the term dynamic precision.

3.1.3 Evaluation

The design and evaluation flow for Cephalopode is illustrated in Figure 3.2. The processor is developed using the Bifröst system [57], whose output is passed to VossII to generate a Verilog file (`ceph.v`) for evaluation. Note that this

process covers only the core; the memories are not synthesized, a point we will revisit later. This Verilog description is then synthesized into a netlist using industrial-grade synthesis tools. Our analysis is based on post-synthesis results rather than post-place-and-route, as the latter process is significantly more complex and was deemed outside the scope of this work. After synthesis, we can

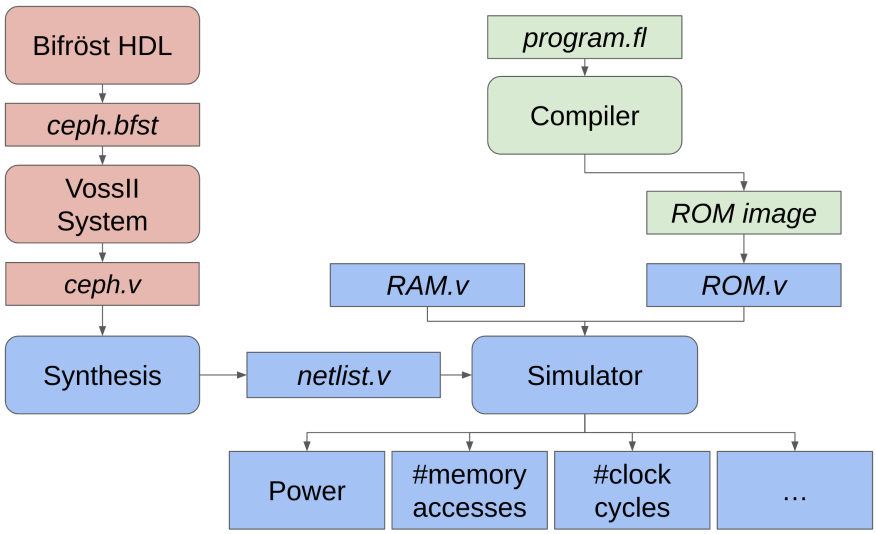


Figure 3.2: High-level overview of the evaluation for Cephalopode. The corresponding flow for MicroHs is in Figure 3.3.

extract figures for the maximum clock frequency and gate count. Additionally, we obtain a preliminary power estimate using vector-free power estimation techniques, which report an approximate consumption of 2.2 mW. However, we know from Section 2.9 that vector-free estimates can be highly inaccurate; to achieve more reliable results, it is necessary to run benchmark programs on the processor.

To assemble a representative benchmark suite for IoT applications, we selected one program from each class in IoTBench [58]. These are small programs, ranging from a recursive computation of 10! to a 10×10 matrix multiplication. The programs are compiled using a basic compiler that produces a ROM image containing the corresponding combinator graph, which can be loaded into the ROM memory for simulation on the netlist. We do not elaborate on the compiler here and refer the reader to the paper for details, noting only that it applies little to no optimization.

Using vector-based power estimation, we observe a power consumption ranging from 300 μW to 400 μW across different programs, with an average of approximately 330 μW. This represents an 85 % reduction compared to the 2.2 mW estimated using vector-free methods.

Up to this point, our analysis has focused solely on the processor core, excluding the memories. The memories are modeled in behavioral Verilog and are not synthesized, requiring us to develop a separate power model for

them based on the target implementation technology. In desktop systems, main memories are typically built using DRAM cells due to their low cost per bit. However, DRAM exhibits higher power consumption than SRAM because it must be periodically refreshed. Additionally, DRAM is significantly slower than SRAM, and since graph reduction is highly memory-intensive, fast memory access is critical. For these reasons, we assume that the memories are implemented using SRAM cells. Accordingly, we model memory accesses such that each read or write operation completes in a single phase, making the memories functionally similar to L1 caches in high-performance systems [4].

With the decision to use SRAM cells, we require an energy consumption model. Since the processor is synthesized using the ASAP7 standard cell library, we draw on prior experimental data from this library, which reports energy usage of approximately 0.25 pJ per write and 0.23 pJ per read [59]. Based on this, our model scales the energy per access by the memory width, yielding:

$$E = mem\_width \cdot (0.25 \cdot \#writes + 0.23 \cdot \#reads).$$

The static power consumption of SRAM is near-zero, so we exclude it from the analysis [60]. Tracking memory accesses is straightforward using counters in the behavioral memory module that are incremented for each read and write.

For evaluation, we need to compare the processor to another system setup (hardware and software) that one may reasonably use for embedded devices. For hardware, we use a small RISC-V processor [61], [62]. In terms of software, we use two setups, seeking to answer two different questions. First, we compare with handwritten C code, comparing the energy consumption while accepting the risks of low-level languages. Secondly, we aim to investigate alternative options for programming IoT devices using high-level functional languages. At first glance, this seems challenging, as we cannot rely on GHC—its large runtime system was the very reason this project was initiated. Fortunately, an alternative exists: in spring 2023, Augustsson began developing MicroHs, a lightweight Haskell compiler that translates Haskell programs into combinator graphs [63]. These graphs, which represent the compiled programs, are executed by the minimalist MicroHs runtime system. This is written in C, and is small enough to fit within the memory constraints of typical IoT devices.

For both approaches, we need a way to compile and execute C code on the simulated RISC-V core. For this, we use the GNU GCC RISC-V toolchain [64] to create memory images for the ROM and RAM memories. These are then loaded into the memories for simulation.

For MicroHs, there are some prior steps to consider. The MicroHs compiler *mhs* is first compiled normally, on a regular desktop machine. The resulting binary is then used to compile the Haskell program into a combinator graph file, which essentially is a large C array containing the combinator graph. This is bundled together with the MicroHs runtime and compiled to ROM and RAM images as before. Figure 3.3 illustrates the evaluation flow, which we note is very similar to Cephalopode.

As a final note before we show the results, recall that Cephalopode uses what we call *dynamic* precision. In MicroHs, at the time the paper was

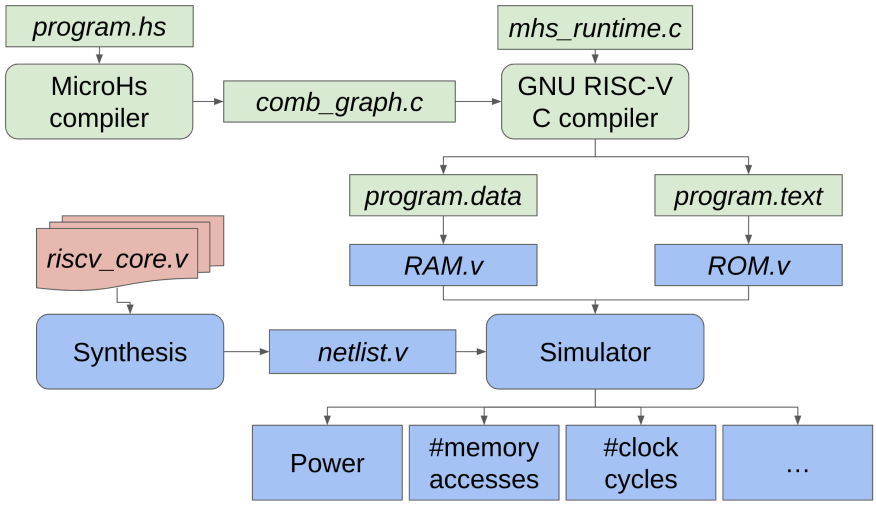


Figure 3.3: Evaluation flow for MicroHs on the RISC-V core.

written[1], arbitrary precision was implemented as a library rather than runtime system primitives. As a result, its performance is significantly worse than it would be in a lower level implementation. Therefore, we compare both with fixed-precision and arbitrary-precision MicroHs, concluding that the "real" number is somewhere in between. We do the same for C, where we use the arbitrary-precision arithmetic package from VossII [65].

We now present the results. Figure 3.4 shows the energy consumption comparisons relative to Cephalopode. First, we examine the comparison between Cephalopode and MicroHs, addressing the question of alternative options for executing high-level functional languages on low-level platforms. The geometric mean (GM), is shown to the far right. The data reveals that Cephalopode consumes 17 times less energy than fixed-precision MicroHs and 174 times less than arbitrary-precision MicroHs running on the RISC-V core. To provide some context, imagine a battery that could power Cephalopode for 10 years. That same battery would only last 7 months for the fixed-precision MicroHs implementation, and just 3 weeks for the arbitrary-precision version. In conclusion, we conclude that Cephalopode is an energy-efficient hardware execution engine for high-level functional languages. We then evaluate the results of handwritten C code running on a RISC-V platform, thereby addressing the question of Cephalopode's energy efficiency in comparison to low-level implementations, while acknowledging the inherent trade-offs and risks of working in low-level languages. As expected, the fixed-precision C version achieves the highest efficiency—approximately 50 times more memory efficient than Cephalopode—while the arbitrary-precision C version performs about twice as efficiently. However, it is noteworthy that Cephalopode, despite

[1]The arbitrary-precision implementation in MicroHs has been improved since the paper was published. Private communication, Augustsson, 2024.

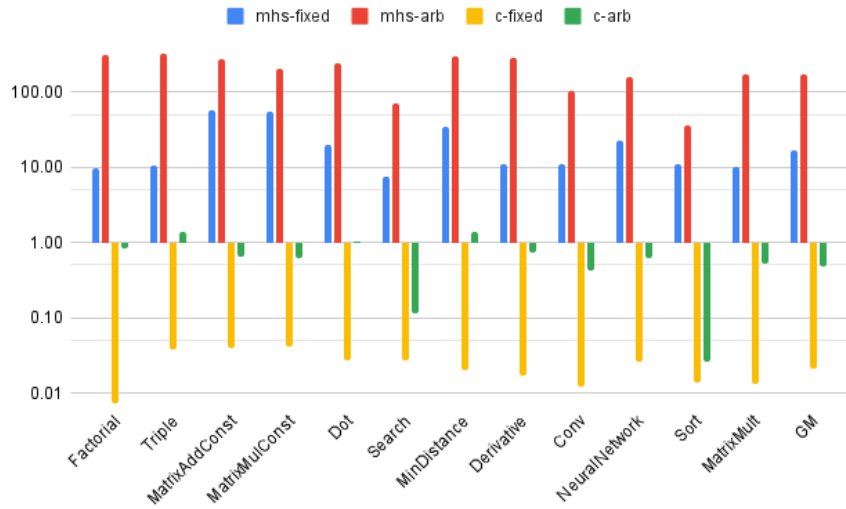


Figure 3.4: Energy consumption of the systems relative to Cephalopode. GM, to the far right, is the geometric mean of the results.

being a high-level and type-safe system, remains remarkably competitive with the arbitrary-precision C code. This demonstrates the ability of Cephalopode to deliver strong performance without sacrificing the benefits of abstraction and safety.

A deeper analysis of power consumption reveals interesting insights. Focusing solely on the processor cores and excluding memory, Cephalopode draws just 336 μW, outperforming the RISC-V core, which consumes 445 μW—despite Cephalopode being approximately 3.6 times larger in terms of gate count (49,000 vs. 13,500 gates). This efficiency can be largely attributed to the inherently low-power nature of the combinator graph reduction execution model. Another key contributor is Cephalopode's use of clock gating, which is a technique for minimizing dynamic power consumption by disabling the clock from unused parts of the circuit [4]. When clock gating was first introduced in Cephalopode, core power dropped by approximately 40 %. Cephalopode operates with 18 distinct clock domains, allowing most of the processor to remain inactive during execution.

3.1.4 Related work and conclusions

Cephalopode introduces a new design point within the field of custom processors for functional languages. This area of research was particularly active in the 1980s, spurred by Turner's 1979 paper on using combinators for program evaluation [66]. Notable historical examples include SKIM [67], NORMA [68], TIGRE [69], and the G-machine [70]. The G-machine later influenced the development of the Spineless Tagless G-machine [71], which remains the foundation

of GHC today, largely because of its efficiency on conventional computer architectures, especially in minimizing memory accesses. More recent contributions to the field include the Reduceron [72] and the Heron processor [73]. To the best of our knowledge, however, Cephalopode is the first design specifically aimed at resource-constrained devices, with low energy consumption as the primary objective.

In conclusion, Cephalopode demonstrates that specialized hardware for executing functional languages in resource-limited environments is both a practical and efficient solution. It adds a new option in the design space where the trade-offs among hardware resources, energy efficiency, programming abstraction, and security must be carefully considered.

3.2 BDD-Based Methods for Constrained and Biased Simulation-Vector Generation

This paper introduces methods for generating input vectors for hardware simulation that conform to arbitrary input distributions, using BDDs as the core computational tool. These methods have broad applicability in the domain of verification, where properties are often formulated as implications—that is, a property must hold only whenever a specified precondition is satisfied. If the precondition does not hold, the test case is considered irrelevant and can be discarded. This concept forms the basis of constrained vector simulation, which focuses on generating only those input patterns that meet the specified constraints.

In Section 2.9 we detailed how to compute the power of a combinational circuit as a function of its input vectors. This can be done using either scalar inputs (useful in approximate solutions, e.g. Monte Carlo) or symbolic inputs (for exact solutions). We also have seen how signal probabilities can bias the input distribution, and then how transition probabilities give further control of the input distribution.

However, methods for defining and generating arbitrary input vector distributions have not yet been addressed. This work introduces techniques that enable symbolic power estimation under arbitrary input distributions. We present methods that generate scalar vectors, and also techniques that work in symbolic simulation.

3.2.1 Motivation and summary

We introduce the motivation for the problem at hand by gradually building up its complexity. Consider a small example circuit with four input bits, and suppose we want to generate input vectors of the form $\langle a, b, c, d\rangle$. These vectors must satisfy a Boolean constraint $C : \mathbb{B}^4 \to \mathbb{B}$, which we assume is satisfiable. In other words, every generated vector $\vec{v}$ must satisfy $C(\vec{v}) = 1$. To illustrate, we reuse the mutex condition introduced in Section 2.7.5, which requires that exactly one of four bits is set to true. Thus, $\langle F, F, F, T\rangle$ satisfies the condition, while $\langle F, T, F, T\rangle$ violates it.

If we only require **soundness**—that is, all generated vectors satisfy the constraint—then, technically, we could repeatedly produce the same valid vector (e.g., $\langle F, T, F, F\rangle$). While clearly insufficient for most purposes, this still presents a computational challenge: even finding one satisfying vector is generally NP-hard, as it involves solving the SAT problem for C.

However, we typically also want **completeness**: every vector satisfying C should have a non-zero probability of being generated. This ensures coverage of the entire solution space. This problem was addressed earlier in Section 2.7.5, where we used `param` to create a parametric substitution for the example above. Substitutions generated using `param` ensure soundness and completeness.

While soundness and completeness are often sufficient, there are many cases where we also care about the distribution of the generated vectors. Typically,

we want a **uniform distribution**, where each satisfying vector is equally likely to be generated.

One straightforward approach to achieve uniformity is rejection sampling, where we sample vectors uniformly from the full input space and test them against the constraint C. Vectors that do not satisfy C are discarded, and the process is repeated. This guarantees uniformity among the satisfying vectors, assuming uniform initial sampling. While rejection sampling guarantees uniformity, it becomes highly inefficient when the constraint is too restrictive. For instance, if we consider the mutex constraint for vectors of length n, only $\frac{n}{2^n}$ of the possible input vectors satisfy C, making the approach impractical for large n given this constraint. Clearly, we also need to consider the sampling **efficiency**.

If the constraint C can be encoded as a BDD, exact uniform sampling can be achieved with the following baseline algorithm. We first compute the truth cover (introduced in Section 2.8.1) of C—we denote this by $tc(C)$. To generate a vector, we then perform a *walk* down the BDD, branching to the low or high child depending on their respective truth covers, as illustrated in Figure 3.5. In the figure, we are about to pick the value for variable a, knowing that if we set $a = F$ there are 3 vectors that satisfy C, and 5 if $a = T$. We should therefore set $a = F$ with probability $\frac{3}{3+5}$.

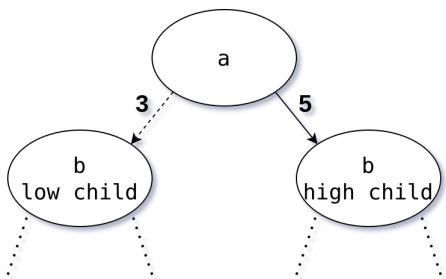


Figure 3.5: Variable a should be set to F with probability $\frac{3}{3+5}$.

This process can be implemented by generating a new random value at each decision node. However, a more efficient alternative is to randomize a single value once and then follow a deterministic path based on that value. Specifically, we sample an index i uniformly from the range $[0, tc(C) - 1]$, where $tc(C)$ is the truth cover for the constraint.

The walk proceeds as follows: at each node, we compare i with the truth cover of the left child, $tc(L)$. If $i < tc(L)$, we continue to the left child L. Otherwise, we move to the right child H and update the index: $i' := i - tc(L)$. For example, in the figure above, suppose we sample $i = 6$. We check whether $6 < 3$. This is false, so we move to the right and update the index to $6 - 3 = 3$. We are now looking for the 3rd solution in a the H BDD, which we know has 5 solutions. This continues until all variables are assigned.

In the paper, we refer to this baseline method as `tcwalk`: truth cover-guided walk. Thanks to memoization of truth cover, its complexity is $O(n)$, where n

is the number of variables in the BDD.

For our purposes in symbolic analysis of power consumption, we also want the results to be **usable in symbolic simulation**. While methods like rejection sampling generate individual vectors which cannot be used for this, parametric substitutions can. Unfortunately, `param` does not not take the distribution of values into account, and thus violate our uniformity requirement. We recall the parametric substitution for the mutex condition that we computed in Section 2.7.5:

$$\langle a[3],\ \overline{a[3]}\,a[2],\ \overline{a[3]}\,\overline{a[2]}\,a[1],\ \overline{a[3]}\,\overline{a[2]}\,\overline{a[1]}\rangle \tag{3.1}$$

There are eight possible assignments to variables $a[3]$, $a[1]$, and $a[1]$. This substitution maps four of them to $\langle T, F, F, F\rangle$, two to $\langle F, T, F, F\rangle$, and the remaining solutions have one mapping each. In other words, applying uniform input probabilities results in a highly skewed distribution.

As an alternative, consider the substitution

$$\langle \overline{a[1]}\,\overline{a[0]},\ \overline{a[1]}\,a[0],\ a[1]\,\overline{a[0]},\ a[1]\,a[0]\rangle \tag{3.2}$$

in which every mutex vector has equal probability of being generated. Computing substitutions like this is the first major contribution of the paper. To this end, we introduce the function `uparam`, short for *uniform parametric substitution*. Given a constraint C, `uparam` generates a parametric substitution that ensures: (1) all generated vectors satisfy C (soundness), (2) every vector satisfying C can be generated (completeness), (3) all satisfying vectors are produced with approximately equal probability (uniformity). Most importantly, the results can be used in symbolic simulation, facilitating their use in our symbolic analysis of power consumption.

A straightforward way to implement `uparam` is as a symbolic counterpart to `tcwalk`. This involves first creating a symbolic version of the function that finds the i-th solution—achieved simply by changing the index i from an integer to a symbolic bit-vector. The randomization step at the top level is then handled by introducing a bit-vector u that is large enough to represent any value in the range $[0, tc(C) - 1]$. We compute the symbolic index as $i = u \bmod tc(C)$, which effectively maps u into the desired range as uniformly as possible. Although the implementation in the paper is more sophisticated—mainly to better leverage memoization—it follows the same core idea.

With `uparam` covered, we drop the requirement of usability in symbolic simulation at the moment. In some cases, we want to go beyond a uniform distribution and allow for user-defined **arbitrary distributions** of the inputs. For instance, if analyzing an adder, we may want to generate small inputs more frequently than larger ones. In our work, we allow the user to provide the distribution using a symbolic bit-vector, where substitution of vector $\vec{v}$ yields the frequency of $\vec{v}$. We call these FBVs, short for frequency bit-vectors. Conceptually, an FBV can be viewed as a multiset (or "bag") of vectors, where the frequency of a vector indicates how many times it appears in the bag. Sampling from the FBV involves selecting a vector uniformly at random from this multiset. For example, we have

```
let a = bv_unsigned "a" 4;
let {fbv1::bv} =
  IF a = '0b0001 THEN '5 ELSE
  IF a = '0b0010 THEN '1 ELSE
  IF (mutex cmd) THEN '2 ELSE '0;
```

This bit-vector effectively represents a frequency table for each value. By substituting an input vector, we can retrieve its corresponding frequency:

```
let subst [v3,v2,v1,v0] =
  substitute [("a[3]", v3), ("a[2]", v2),
              ("a[1]", v1), ("a[0]", v0)] fbv1;
subst [F,T,F,F]; // <F,T,F>, i.e. 2
subst [F,F,F,T]; // <F,T,F,T>, i.e. 5
subst [F,F,F,F]; // <F>, i.e. 0
```

The probability of selecting a specific vector $\vec{v}$ is given by:

$$\Pr\{\text{Picking } \vec{v}\} = \frac{\text{frequency of } \vec{v}}{\text{sum of all frequencies in the FBV}}. \tag{3.3}$$

For example,

$$\Pr\{\text{Picking } \langle F, F, T, F\rangle\} = \frac{1}{5+1+2+2} = 10\%.$$

To simplify the definition of such FBVs, we introduce the `weighted_switch` function, used as follows:

```
let a = bv_unsigned "a" 16;
let b = bv_unsigned "b" 16;
let adder_inputs_small = weighted_switch [
  (30, (a = '0) OR (b = '0)),
  (60, (a < '16) AND (b < '16)),
  (10, T)];
```

This FBV specifies the distribution for generating bit-vectors a and b. In 30 % of the cases, either $a = 0$ or $b = 0$ is produced. In 60 % of the cases, both a and b are generated with small values (less than 16). The remaining 10 % covers all other values that do not meet the previous conditions.

We introduce the `bv_tcwalk` algorithm for sampling from FBVs. Conceptually, it is a variant of `tcwalk` where we replace the calls to `tc` with a new function `bv_tc`. This function computes the sum of all values that a bit-vector may represent, that is, the denominator in equation 3.3. We leave its implementation, and that of `bv_tcwalk` itself, for the paper.

Our final contribution is `nuparam` (non-uniform param), a function that computes a parametric substitution for arbitrary distributions under certain simplifying assumptions. Instead of operating on FBVs, `nuparam` directly takes a list of (weight, condition) pairs—similar in form to the arguments used in `weighted_switch`. This approach assumes that the weights sum to a power of two, and that the conditions are mutually exclusive. If these assumptions are not initially met, simple preprocessing steps can be applied to enforce them. Further implementation details and discussion are provided in the paper.

Table 3.1 summarizes our main contributions[2] alongside related work, which we will discuss shortly. A detailed performance comparison is not included at this stage, as we have not yet conducted a thorough evaluation.

Table 3.1: Overview of contributed methods (top) and related work (bottom). `nuparam` requires some simplification of its input, hence the asterisk.

Method	**Distribution**	**Supports symbolic simulation**
tcwalk	Uniform	No
bv_tcwalk	Arbitrary	No
uparam	Near-uniform	Yes
nuparam	Arbitrary*	Yes
`param`	No guarantees	Yes
Relation circuits [74]	No guarantees	Yes
SAT-based [38][75][76][77][78]	Near-uniform	No
MCMC [79][80]	Near-uniform	No
Weighted BDDs [81][82]	Bitwise biasing	No

3.2.2 Related work

`param` was covered in Section 2.7.5. It computes parametric substitutions for a constraint, meaning soundness and completeness are satisfied, and its output can be used in symbolic simulation. However the distribution of generated values is not considered.

Kukula et al. propose a method that can be seen as a generalization of `param` [74]. Their method allows the the construction of a combinational circuit satisfying a relation $\mathrm{T}(x,y)$ between inputs x and outputs y. However, much like `param`, their method does not take distributions into account.

The following methods focus on the problem of constrained random simulation—that is, generating individual satisfying assignments for a constraint C in a manner that balances sampling efficiency and uniformity. Note that they produce concrete vectors rather than symbolic representations, and thus are not suitable for symbolic simulation.

SAT-based techniques have been widely used for this task. The general strategy closely follows the approach used in approximate model counting, as discussed in Section 2.8.2. Put simply, randomly generated XOR constraints are added to the original formula to shrink the solution space to a manageable size. Once the space is small enough, all satisfying assignments can be enumerated, and one can be selected uniformly at random. The primary difficulty—and the source of approximation—lies in how the constraints are chosen to ensure the reduced space is neither too large nor empty [38], [75]–[78]. Thanks to this reduction of the solution space, this approach can scale to problem sizes of hundreds of thousands of variables.

[2] Referring to `tcwalk` as a contribution is generous, will be made clear in the related work on weighted BDDs. Nevertheless, we include it for completeness.

If a vector $\vec{v}$ is known to satisfy constraint C, then vectors similar to $\vec{v}$ are likely to also satisfy C. Kitchen et al. applied this intuition in constrained random simulation by applying MCMC (Markov Chain Monte Carlo) methods [79]. They use Gibbs sampling and generate each new sample by mutating the previous one, with a recovery strategy in place for samples that violate C—ultimately falling back on SAT solving when needed. Their system supports both integer and Boolean constraints, enabling the use of operations like multiplication without the exponential blowup typically encountered with BDDs. However, a known drawback of MCMC methods, Gibbs sampling in particular, is that consecutive samples are highly correlated. To address this, they maintain a "pool" of previously generated samples and randomly select one to mutate, reducing correlation to some extent. Their evaluation shows promising results, with performance on par with BDD-based techniques, but without the risk of exponential state space growth. Further details can be found in Kitchen's PhD thesis [80].

Finally, Yuan et al. present BDD-based method, which, alongside `param`, is the closest to our work [81], [82]. Their algorithm consists of sampling vectors that satisfy a constraint C provided as a BDD. Conceptually, it is a generalization of `tcwalk`—which we originally implemented as a simplification of their approach. Unlike `tcwalk` and the SAT and MCMC methods from before, their technique allows for non-uniform sampling and by imposing *weights*—which work like signal probabilities—on individual bits.

To clarify this, we again consider the 4-bit mutex example for vector $\langle a[3], a[2], a[1], a[0]\rangle$, where we now apply the biases:

$$\Pr\{a[3]\} = \frac{1}{2}, \Pr\{a[2]\} = \frac{1}{3}, \Pr\{a[1]\} = \frac{1}{4}, \Pr\{a[0]\} = \frac{1}{5} \tag{3.4}$$

Given this, the weight of each vector is the product of the individual bit biases, for instance:

$$w(\langle T, F, F, F\rangle) = \left(\frac{1}{2}\right) \cdot \left(1 - \frac{1}{3}\right) \cdot \left(1 - \frac{1}{4}\right) \cdot \left(1 - \frac{1}{5}\right) = \frac{24}{120} \tag{3.5}$$

The probability of generating a particular vector is the weight of the vector divided by all weights. for instance the probability $\Pr\{\langle T, F, F, F\rangle\}$ is given by

$$\frac{w(\langle T, F, F, F\rangle)}{w(\langle T, F, F, F\rangle) + w(\langle F, T, F, F\rangle) + w(\langle F, F, T, F\rangle) + w(\langle F, F, F, T\rangle)} \tag{3.6}$$

Which for our biasing is

$$\frac{\frac{24}{120}}{\frac{24}{120} + \frac{12}{120} + \frac{8}{120} + \frac{6}{120}} = \frac{24}{50} \tag{3.7}$$

This simplifies to a uniform distribution if no biasing is applied.

In `tcwalk`, we generate vectors using random walks down the BDD based on truth cover. In their work, the role of truth cover is the placed by the `weight` function, which also taking the biases into account. The `weight` function is computed bottom-up as:

$$\mathit{weight}(n) = \Pr\{n\} \cdot \mathit{weight}(H) + (1 - \Pr\{n\}) \cdot \mathit{weight}(L)$$

for node n with biasing $\Pr\{n\}$ and children H and L. Figure 3.6 shows the constraint BDD for the 4-bit mutex example where the nodes have been annotated by their weights, biased according to equation 3.4. To exemplify a

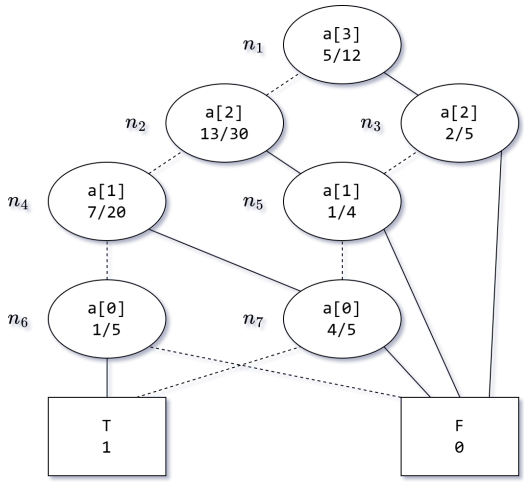


Figure 3.6: Constraint BDD with weights.

weight computation, the weight of node n_4 would be computed as:

$$\Pr\{a[1]\} \cdot weight(n_7) + (1 - \Pr\{a[1]\}) \cdot weight(n_6) = \frac{1}{4} \cdot \frac{4}{5} + \frac{3}{4} \cdot \frac{1}{5} = \frac{7}{20} \quad (3.8)$$

When walking down the data structure, we branch to H with probability $\frac{\Pr\{n\} \cdot weight(H)}{weight(n)}$, and to L otherwise.

To demonstrate correctness, we consider the probability of generating the vector $\langle 1, 0, 0, 0 \rangle$, which corresponds to a walk though nodes n_1, n_3, n_5, n_7, T:

$$\frac{\Pr\{a[3]\} w(n_3)}{w(n_1)} \frac{\Pr\{\overline{a[2]}\}) w(n_5)}{w(n_3)} \frac{\Pr\{\overline{a[1]}\}) w(n_7)}{w(n_5)} \frac{\Pr\{\overline{a[0]}\}) w(T)}{w(n_7)} = \quad (3.9)$$

$$\frac{w(T)}{w(n_1)} (\Pr\{a[3]\} \Pr\{\overline{a[2]}\} \Pr\{\overline{a[1]}\} \Pr\{\overline{a[0]}\}) = \quad (3.10)$$

$$\frac{1}{\frac{5}{12}} (\frac{1}{2} \frac{2}{3} \frac{3}{4} \frac{4}{5}) = \frac{24}{50} \quad (3.11)$$

Which is the same probability that we computed in equation 3.7.

As previously mentioned, biasing individual input bits corresponds to signal probabilities, in that we cannot take correlations between the bits into account.

3.2.3 Conclusions and future work

In summary, this paper introduces new techniques for generating vectors under constraints and specified input distributions, applicable to both scalar and symbolic simulation.

There are several promising directions for future work. A key one is conducting a thorough performance evaluation of the proposed algorithms, with comparisons to existing approaches. This is particularly challenging for the `uparam` and `nuparam` algorithms, as there is limited prior work to benchmark against. A starting point would be to analyze their computational complexity, which has not yet been determined. Preliminary findings suggest that while these algorithms generally scale less favorably than `param`, the difference is surprisingly small.

Related work on weighted BDDs highlights methods such as constraint partitioning and simplification to improve scalability for larger problem instances [81], [82]. Additionally, their framework supports varying constraints and bias based on the current state. Neither of these are features we have included in our methods, but would be very good to have.

Another avenue worth exploring is the use of binary moment diagrams (BMDs) for specifying input distributions [83]. BMDs generalize BDDs by supporting linear functions over integers or real numbers, offering a more expressive representation framework.

Finally, and perhaps most significantly in the context of this thesis, is the application of these methods to our ongoing work in symbolic power analysis. While the paper includes some initial examples, the most compelling and impactful developments are yet to come.

Chapter 4

Conclusions and future work

In this thesis, we have explored the intersection of power consumption and hardware security. We began by presenting the implementation and evaluation of the Cephalopode processor—a custom, low-power processor designed to execute high-level functional languages. The motivation for this stems from the inherent security vulnerabilities of low-level languages, as demonstrated by several practical examples we have examined. The energy efficiency of the processor was compared against a low-power RISC-V core executing a similar high-level language in software. The evaluation showed that Cephalopode achieves strong energy performance, highlighting the effectiveness of custom hardware designs in providing high security without sacrificing energy efficiency.

In the core of the thesis we have investigated the application of formal verification tools in the area of power analysis in hardware circuits. This area involves computing the power consumption of a circuit as a function of its input vectors. The power consumption therefore becomes a function of the *distribution* of the input vectors, adding the additional dimension of probability to the problem. Whereas previous work only allowed for limited control of the input distribution, we have seen how the second paper in this thesis allows for efficiently describing and generating arbitrary distributions of input vectors, for both scalar and symbolic simulation.

We now turn to directions for future work. It is helpful to frame this discussion across three time horizons: short-term (weeks to months, or the next paper), medium-term (spanning a few years or several publications ahead), and long-term (extending over many years).

We first revisit the central research question: *Is it possible to design circuits that are provably secure against power side-channel attacks?* This question can be approached at three levels of abstraction:

- **Analysis** - What does it mean for a circuit to be secure against power attacks? How should power be modeled?
- **Verification** - Given a specific circuit, how can we determine or prove that it is secure?
- **Design** - Can we create circuits that are provably secure from the outset, and can this be incorporated into a broader design methodology?

This thesis has concentrated solely on the **analysis** level. We have examined power models that abstract the complex behavior of CMOS circuits into forms amenable to formal analysis. We demonstrated how BDD-based symbolic simulation can be used to represent power as a function of the primary circuit inputs. Our main contribution lies in extending this analysis by enabling the use of symbolic input distributions, thus allowing power consumption to be studied as a posterior distribution determined by the input vector distribution.

Work remains with regards to bringing our tools into the verification stage. The most important next step is to expand beyond purely combinational circuits and consider sequential circuits. Since most real-world digital systems are sequential in nature, this step is crucial for the broader applicability of our methods. We discussed in Section 2.9.4 that the average power consumption of the simplest sequential circuits, deterministic finite state machines, can be computed by finding the probability of being in each state, which can be found by solving a linear equation. The problem is state-explosion: with n flip flops, there are 2^n states. In future work, we aim to investigate the use of symbolic methods to mitigate this. We anticipate that extending our analysis to sequential circuits will be achievable within a short- to medium-term timeframe.

Another important consideration is the accuracy of the employed power model. In our work, we have focused on a simple, deterministic model that considers only dynamic power. As discussed in the introduction, proving security under such a model may fail to account for "higher-order" power effects that a skilled attacker could potentially exploit. For example, the model treats transitions like $T \to F$ and $F \to T$ as equivalent, which is not entirely accurate. Similarly, even for gates implementing commutative Boolean functions—such as NAND—the actual power consumption may differ slightly depending on the input ordering, though our model does not distinguish between them. Additionally, the influence of interconnects (wires), which can represent a significant portion of total power consumption [4], is entirely omitted. Naturally, incorporating more detailed power models would increase computational complexity and may end up modeling variations too subtle to be exploitable in practical attacks. While this can be worked on in the short timeframe, it is likely more valuable to revisit once we have more insight into assumptions about the attacker capability.

Also in the short to medium term, we aim to develop a tool for identifying vulnerabilities to power side-channel attacks, thereby progressing into the **verification** phase. This tool would take a hardware circuit and corresponding input specifications as input, and automatically flag potential security weaknesses. We can draw significant inspiration from existing tools designed for

timing side-channel analysis, such as Iodine [84] and WhisperFuzz [85]. It is likely that we will build such a tool ourselves—possibly even one that integrates both power and timing side-channel analysis. Once the tool is developed and released, we plan to evaluate its usability, again looking to the usability studies of timing-analysis tools for guidance [86].

In the long term, we can eventually consider the **design** stage—designing circuits are provably secure from side-channel attacks. Here, we can draw inspiration from functional verification: Can we design circuits that are provably secure from the ground up, in the same way that we design circuits that are provably correct from the ground up? This will invariably lead to trade-offs between security and efficiency, requiring methods for quantifying leakage.

To explore the idea of quantifying leakage, we consider the following. The contribution of our second paper enables the analysis of power consumption as a posterior distribution derived from a given prior input distribution. This opens the door to situating our work within the broader framework of quantitative information flow [87]. In essence, this field models systems as mappings from input distributions to output distributions, providing a foundation for quantifying the likelihood of information—such as secrets—being leaked. We speculate that the insight this brings will be highly useful in the case where we allow some information to be leaked, but want to quantify the risk in order to make informed trade-offs between security and efficiency. Like more advanced power models, this may be more beneficial to return to later rather than considering now.

One important direction we should pursue in the near future is the integration of alternative decision procedures—such as SAT and SMT solvers—alongside BDDs. Recent advances in SMT technology [88] create promising opportunities for experimentation and enhancements to our toolchain. Furthermore, VossII has recently integrated an SMT interface, making experimentation easier. Regarding our work in symbolic constrained vector generation, we see particular potential in SAT/SMT-based model counting techniques. Additionally, the floating-point capabilities of modern SMT solvers could prove highly effective for probabilistic computations—for example, calculating the probability distribution over states in sequential circuits. In this setting, formal methods for linear algebra may also become increasingly relevant, as discussed in [89]. We ourselves have done some preliminary work on symbolic linear algebra, but this is still in the early stages.

Future work related to the input generation paper includes exploring approximate techniques, particularly those used in approximate SAT-based model counting solvers. This is especially relevant for the `uparam` and `nuparam` algorithms, whose computational complexity also remains to be fully analyzed. Another promising direction is improving usability—specifically, identifying convenient yet efficient methods for users to define input distributions. One possible approach is developing a domain-specific language (DSL) for this purpose, taking inspiration from tools like QuickCheck [90].

We conclude by outlining potential future directions for the evaluation of Cephalopode. In our current evaluation, we synthesized the design and performed all analysis on the post-synthesis results. Future work could take

the processor closer to physical implementation, potentially mapping it onto an FPGA. This would enable the collection of more realistic power measurements and eliminate the need for a simulated memory model. Additionally, it would facilitate more direct comparisons with related projects such as Heron [73].

Bibliography

[1] TOP500 Project, *Top500 list - november 2024*, Accessed: 2025-04-25, 2024. [Online]. Available: https://top500.org/lists/top500/2024/11/ (cit. on p. 3).

[2] U.S. Energy Information Administration, *How much electricity does an american home use?* Accessed: 2025-04-25, 2024. [Online]. Available: https://www.eia.gov/tools/faqs/faq.php?id=97&t=3 (cit. on p. 3).

[3] P. Insights. "Bitcoin electricity consumption comparable to that of poland." Accessed: 2025-04-25. (2023), [Online]. Available: https://www.polytechnique-insights.com/en/columns/energy/bitcoin-electricity-consumption-comparable-to-that-of-poland/ (cit. on p. 3).

[4] S. Kaxiras and M. Martonosi, *Computer Architecture Techniques for Power-Efficiency* (Synthesis Lectures on Computer Architecture), en. Cham: Springer International Publishing, 2008, ISBN: 978-3-031-00593-0 978-3-031-01721-6. DOI: 10.1007/978-3-031-01721-6. [Online]. Available: https://link.springer.com/10.1007/978-3-031-01721-6 (visited on 02/02/2023) (cit. on pp. 3, 9, 10, 41, 43, 54).

[5] M. Själander, M. Martonosi and S. Kaxiras, *Power-Efficient Computer Architectures: Recent Advances* (Synthesis Lectures on Computer Architecture), en. Cham: Springer International Publishing, 2015, ISBN: 978-3-031-00617-3 978-3-031-01745-2. DOI: 10.1007/978-3-031-01745-2. [Online]. Available: https://link.springer.com/10.1007/978-3-031-01745-2 (visited on 24/04/2025) (cit. on p. 3).

[6] Cybersecurity and Infrastructure Security Agency, *Product security: Bad practices*, Accessed: 2025-04-25, 2024. [Online]. Available: https://www.cisa.gov/resources-tools/resources/product-security-bad-practices (cit. on p. 4).

[7] P. Kocher, J. Jaffe and B. Jun, "Differential Power Analysis," en, in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1999, pp. 388–397, ISBN: 978-3-540-48405-9. DOI: 10.1007/3-540-48405-1_25 (cit. on p. 4).

[8] P. Kocher, J. Jaffe, B. Jun and P. Rohatgi, "Introduction to differential power analysis," en, *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, Apr. 2011, ISSN: 2190-8516. DOI: 10.1007/s13389-011-0006-y. [Online]. Available: https://doi.org/10.1007/s13389-011-0006-y (visited on 10/04/2024) (cit. on p. 4).

[9] Y. Wang, R. Paccagnella, E. T. He, H. Shacham, C. W. Fletcher and D. Kohlbrenner, "Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86," en, *IEEE Micro*, vol. 43, no. 4, pp. 19–27, Jul. 2023, ISSN: 0272-1732, 1937-4143. DOI: 10.1109/MM.2023.3274619. [Online]. Available: https://ieeexplore.ieee.org/document/10122602/ (visited on 20/11/2023) (cit. on p. 4).

[10] Y. Wang, R. Paccagnella, A. Wandke *et al.*, "DVFS Frequently Leaks Secrets: Hertzbleed Attacks Beyond SIKE, Cryptography, and CPU-Only Data," en, in *2023 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, May 2023, pp. 2306–2320, ISBN: 978-1-66549-336-9. DOI: 10.1109/SP46215.2023.10179326. [Online]. Available: https://ieeexplore.ieee.org/document/10179326/ (visited on 20/11/2023) (cit. on p. 4).

[11] E. Dubrova, K. Ngo, J. Gärtner and R. Wang, "Breaking a Fifth-Order Masked Implementation of CRYSTALS-Kyber by Copy-Paste," en, in *Proceedings of the 10th ACM Asia Public-Key Cryptography Workshop*, Melbourne VIC Australia: ACM, Jul. 2023, pp. 10–20, ISBN: 9798400701832. DOI: 10.1145/3591866.3593072. [Online]. Available: https://dl.acm.org/doi/10.1145/3591866.3593072 (visited on 08/03/2024) (cit. on p. 5).

[12] K. Ngo, E. Dubrova, Q. Guo and T. Johansson, "A Side-Channel Attack on a Masked IND-CCA Secure Saber KEM Implementation," en, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 676–707, Aug. 2021, ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i4.676-707. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9079 (visited on 10/12/2023) (cit. on p. 5).

[13] Y. Yu, F. Marranghello, V. D. Teijeira and E. Dubrova, "One-Sided Countermeasures for Side-Channel Attacks Can Backfire," in *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, ser. WiSec '18, New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 299–301, ISBN: 978-1-4503-5731-9. DOI: 10.1145/3212480.3226104. [Online]. Available: https://dl.acm.org/doi/10.1145/3212480.3226104 (visited on 30/11/2023) (cit. on p. 5).

[14] Y. Nasser, J. Lorandel, J.-C. Prévotet and M. Hélard, "RTL to Transistor Level Power Modeling and Estimation Techniques for FPGA and ASIC: A Survey," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 3, pp. 479–493, Mar. 2021, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Cir-

cuits and Systems, ISSN: 1937-4151. DOI: 10.1109/TCAD.2020.3003276 (cit. on pp. 5, 10, 27).

[15] L. T. Clark, V. Vashishtha, L. Shifren *et al.*, "ASAP7: A 7-nm finFET predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, Jul. 2016, ISSN: 0026-2692. DOI: 10.1016/j.mejo.2016.04.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002626921630026X (visited on 14/11/2023) (cit. on p. 10).

[16] P. Gupta and A. Kahng, "Quantifying error in dynamic power estimation of CMOS circuits," in *Fourth International Symposium on Quality Electronic Design, 2003. Proceedings.*, Mar. 2003, pp. 273–278. DOI: 10.1109/ISQED.2003.1194745. [Online]. Available: https://ieeexplore.ieee.org/document/1194745 (visited on 24/04/2025) (cit. on p. 10).

[17] F. Najm, "Transition density, a stochastic measure of activity in digital circuits," in *28th ACM/IEEE Design Automation Conference*, Jun. 1991, pp. 644–649. DOI: 10.1145/127601.127744 (cit. on p. 11).

[18] R. Burch, F. Najm, P. Yang and T. Trick, "A Monte Carlo approach for power estimation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 1, pp. 63–71, Mar. 1993, Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ISSN: 1557-9999. DOI: 10.1109/92.219908. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/219908 (visited on 03/03/2025) (cit. on pp. 11, 27).

[19] C.-Y. Tsui, M. Pedram and A. Despain, "Exact and Approximate Methods for Calculating Signal and Transition Probabilities in FSMs," in *31st Design Automation Conference*, ISSN: 0738-100X, Jun. 1994, pp. 18–23. DOI: 10.1109/DAC.1994.204066 (cit. on p. 11).

[20] C.-Y. Tsui, J. Monteiro, M. Pedram, S. Devadas, A. Despain and B. Lin, "Power estimation methods for sequential logic circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, no. 3, pp. 404–416, Sep. 1995, Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ISSN: 1557-9999. DOI: 10.1109/92.406998. [Online]. Available: https://ieeexplore.ieee.org/document/406998 (visited on 12/11/2024) (cit. on p. 11).

[21] S. Bhanja and N. Ranganathan, "Switching activity estimation of VLSI circuits using Bayesian networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 4, pp. 558–567, Aug. 2003, Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2003.816144. [Online]. Available: https://ieeexplore.ieee.org/document/1229864 (visited on 13/03/2025) (cit. on p. 11).

[22] T.-L. Chou and K. Roy, "Accurate estimation of power dissipation in CMOS sequential circuits," in *Proceedings of Eighth International Application Specific Integrated Circuits Conference*, ISSN: 1063-0988, Sep. 1995, pp. 285–288. DOI: 10.1109/ASIC.1995.580733 (cit. on p. 11).

[23] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, ser. STOC '71, New York, NY, USA: Association for Computing Machinery, 1971, pp. 151–158, ISBN: 978-1-4503-7464-4. DOI: 10.1145/800157.805047. [Online]. Available: https://dl.acm.org/doi/10.1145/800157.805047 (visited on 24/04/2025) (cit. on p. 13).

[24] M. Davis, G. Logemann and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962, ISSN: 0001-0782. DOI: 10.1145/368273.368557. [Online]. Available: https://dl.acm.org/doi/10.1145/368273.368557 (visited on 24/04/2025) (cit. on p. 14).

[25] J. Marques-Silva and K. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, May 1999, ISSN: 1557-9956. DOI: 10.1109/12.769433. [Online]. Available: https://ieeexplore.ieee.org/document/769433 (visited on 07/04/2025) (cit. on p. 14).

[26] A. Biere, A. Biere, M. Heule, H. van Maaren and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. NLD: IOS Press, Jan. 2009, ISBN: 978-1-58603-929-5 (cit. on pp. 14, 23).

[27] Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986, Conference Name: IEEE Transactions on Computers, ISSN: 1557-9956. DOI: 10.1109/TC.1986.1676819. [Online]. Available: https://ieeexplore.ieee.org/document/1676819 (visited on 26/03/2025) (cit. on p. 14).

[28] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, Nov. 1993, pp. 42–47. DOI: 10.1109/ICCAD.1993.580029. [Online]. Available: https://ieeexplore.ieee.org/document/580029 (visited on 24/04/2025) (cit. on p. 15).

[29] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518, ISBN: 978-3-540-24605-3 (cit. on p. 18).

[30] C.-j. Seger and R. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, vol. 6, Mar. 1994. DOI: 10.1007/BF01383966 (cit. on p. 19).

[31] M. D. Aagaard, R. B. Jones and C.-J. H. Serger, "Formal verification using parametric representations of boolean constraints," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, 1999, pp. 402–407 (cit. on p. 22).

[32] M. Chavira and A. Darwiche, "On probabilistic inference by weighted model counting," *Artificial Intelligence*, vol. 172, no. 6, pp. 772–799, Apr. 2008, ISSN: 0004-3702. DOI: 10.1016/j.artint.2007.11.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370207001889 (visited on 12/03/2025) (cit. on p. 24).

[33] M. Thurley, "sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP," en, in *Theory and Applications of Satisfiability Testing - SAT 2006*, D. Hutchison, T. Kanade, J. Kittler *et al.*, Eds., vol. 4121, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 424–429, ISBN: 978-3-540-37206-6 978-3-540-37207-3. DOI: 10.1007/11814948_38. [Online]. Available: http://link.springer.com/10.1007/11814948_38 (visited on 07/05/2025) (cit. on p. 25).

[34] S. Sharma, S. Roy, M. Soos and K. S. Meel, "GANAK: A Scalable Probabilistic Exact Model Counter," en, in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, Macao, China: International Joint Conferences on Artificial Intelligence Organization, Aug. 2019, pp. 1169–1176, ISBN: 978-0-9992411-4-1. DOI: 10.24963/ijcai.2019/163. [Online]. Available: https://www.ijcai.org/proceedings/2019/163 (visited on 12/03/2025) (cit. on p. 25).

[35] M. Soos and K. S. Meel, "BIRD: Engineering an Efficient CNF-XOR SAT Solver and Its Applications to Approximate Model Counting," en, *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 1592–1599, Jul. 2019, ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v33i01.33011592. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/3974 (visited on 07/05/2025) (cit. on p. 25).

[36] M. Soos, S. Gocht and K. S. Meel, "Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling," en, in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds., vol. 12224, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 463–484, ISBN: 978-3-030-53287-1 978-3-030-53288-8. DOI: 10.1007/978-3-030-53288-8_22. [Online]. Available: http://link.springer.com/10.1007/978-3-030-53288-8_22 (visited on 11/04/2025) (cit. on p. 25).

[37] J. Yang and K. S. Meel, *Rounding Meets Approximate Model Counting*, arXiv:2305.09247 [cs], May 2023. DOI: 10.48550/arXiv.2305.09247. [Online]. Available: http://arxiv.org/abs/2305.09247 (visited on 07/05/2025) (cit. on p. 25).

[38] C. P. Gomes, A. Sabharwal and B. Selman, "Near-uniform sampling of combinatorial spaces using XOR constraints," in *Proceedings of the 20th International Conference on Neural Information Processing Systems*, ser. NIPS'06, Cambridge, MA, USA: MIT Press, Dec. 2006, pp. 481–488. (visited on 12/03/2025) (cit. on pp. 26, 49).

[39] Cadence Design Systems, *Genus Synthesis Solution*, https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html, Accessed: 2025-05-05, 2025 (cit. on p. 26).

[40] F. Najm, "A survey of power estimation techniques in VLSI circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 4, pp. 446–455, Dec. 1994, Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, ISSN: 1557-9999. DOI: 10.1109/92.335013 (cit. on pp. 27, 29, 33, 34).

[41] K. Parker and E. McCluskey, "Probabilistic Treatment of General Combinational Networks," *IEEE Transactions on Computers*, vol. C-24, no. 6, pp. 668–670, Jun. 1975, Conference Name: IEEE Transactions on Computers, ISSN: 1557-9956. DOI: 10.1109/T-C.1975.224279 (cit. on p. 27).

[42] Z. Wang, G. Zhang, J. Ye and J. Jiang, "Reliability Evaluation of Approximate Arithmetic Circuits Based on Signal Probability," in *2021 IEEE International Test Conference in Asia (ITC-Asia)*, ISSN: 2768-069X, Aug. 2021, pp. 1–6. DOI: 10.1109/ITC-Asia53059.2021.9808704 (cit. on pp. 27, 34).

[43] J. Monteiro, S. Devadas, A. Ghosh, K. Keutzer and J. White, "Estimation of average switching activity in combinational logic circuits using symbolic simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 1, pp. 121–127, Jan. 1997, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ISSN: 1937-4151. DOI: 10.1109/43.559336 (cit. on pp. 28, 29).

[44] A. Ghosh, S. Devadas, K. Keutzer and J. White, "Estimation of average switching activity in combinational and sequential circuits," in *[1992] Proceedings 29th ACM/IEEE Design Automation Conference*, ISSN: 0738-100X, Jun. 1992, pp. 253–259. DOI: 10.1109/DAC.1992.227826. [Online]. Available: https://ieeexplore.ieee.org/document/227826 (visited on 20/11/2024) (cit. on p. 29).

[45] J. A. Brzozowski and C.-J. H. Seger, *Asynchronous Circuits* (Monographs in Computer Science), D. Gries and F. B. Schneider, Eds. New York, NY: Springer, 1995, ISBN: 978-1-4612-8698-1 978-1-4612-4210-9. DOI: 10.1007/978-1-4612-4210-9. [Online]. Available: http://link.springer.com/10.1007/978-1-4612-4210-9 (visited on 13/05/2025) (cit. on p. 29).

[46] P. Schneider and S. Krishnamoorthy, "Effects of correlations on accuracy of power analysis-an experimental study," in *Proceedings of 1996 International Symposium on Low Power Electronics and Design*, Aug. 1996, pp. 113–116. DOI: 10.1109/LPE.1996.547490 (cit. on p. 34).

[47] C.-Y. Tsui, M. Pedram and A. Despain, "Efficient estimation of dynamic power consumption under a real delay model," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, Nov. 1993, pp. 224–228. DOI: 10.1109/ICCAD.1993.580061 (cit. on p. 34).

[48] J. Leyden, *Samsung smart fridge leaves Gmail logins open to attack*, https://www.theregister.com/2015/08/24/smart_fridge_security_fubar/, Accessed: 2023-11-20, Aug. 2015 (cit. on p. 35).

[49] A. Greenberg, "The Jeep Hackers Are Back to Prove Car Hacking Can Get Much Worse," en-US, *Wired*, Section: tags, ISSN: 1059-1028. [Online]. Available: https://www.wired.com/2016/08/jeep-hackers-return-high-speed-steering-acceleration-hacks/ (visited on 20/11/2023) (cit. on p. 35).

[50] Twitter, Email and Facebook, *Hackers tapping home appliances to launch attacks*, en-US, Section: Science, Oct. 2016. [Online]. Available: https://www.sandiegouniontribune.com/news/science/sd-me-hackable-home-20161003-story.html (visited on 20/11/2023) (cit. on p. 35).

[51] S. Marlow *et al.*, "Haskell 2010 language report," *Available on: http://www. haskell. org/onlinereport/haskell2010*, 2010 (cit. on p. 38).

[52] P. Li and S. Zdancewic, "Encoding information flow in Haskell," in *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, ISSN: 2377-5459, Jul. 2006, 12 pp.–16. DOI: 10.1109/CSFW.2006.13. [Online]. Available: https://ieeexplore.ieee.org/document/1648705 (visited on 22/05/2024) (cit. on p. 38).

[53] J. Pope, J. Saget and C.-J. H. Seger, "Cephalopode: A custom processor aimed at functional language execution for IoT devices," en, in *2020 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, Jaipur, India: IEEE, Dec. 2020, pp. 1–6, ISBN: 978-1-72819-148-5. DOI: 10.1109/MEMOCODE51338.2020.9315094. [Online]. Available: https://ieeexplore.ieee.org/document/9315094/ (visited on 25/01/2023) (cit. on p. 38).

[54] J. Pope, "Once More, With Combinators: Designing a Low-Power Architecture for Functional Programming," en, ISBN: 9789181030624, Ph.D. dissertation, Chalmers University of Technology, 2024. [Online]. Available: https://research.chalmers.se/en/publication/541212 (visited on 24/04/2025) (cit. on p. 38).

[55] D. A. Turner, "A new implementation technique for applicative languages," en, *Software: Practice and Experience*, vol. 9, no. 1, pp. 31–49, 1979, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380090105, ISSN: 1097-024X. DOI: 10.1002/spe.4380090105. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380090105 (visited on 28/04/2025) (cit. on p. 38).

[56] T. Yuasa, "Real-time garbage collection on general-purpose machines," *Journal of Systems and Software*, vol. 11, no. 3, pp. 181–198, 1990 (cit. on p. 39).

[57] J. Pope and C.-J. H. Seger, "Bifro¨st: Creating Hardware With Building Blocks," en, (cit. on p. 39).

[58] S. Chen, C. Luo, W. Gao and L. Wang, "Iotbench: A data centrical and configurable iot benchmark suite," *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, vol. 2, no. 4, p. 100 091, 2022 (cit. on p. 40).

[59] V. Vashishtha, M. Vangala, P. Sharma and L. T. Clark, "Robust 7-nm SRAM design on a predictive PDK," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, ISSN: 2379-447X, May 2017, pp. 1–4. DOI: 10.1109/ISCAS.2017.8050316. [Online]. Available: https://ieeexplore.ieee.org/document/8050316 (visited on 10/11/2023) (cit. on p. 41).

[60] P. Sandeep, P. A. Harsha Vardhini and V. Prakasam, "SRAM Utilization and Power Consumption Analysis for Low Power Applications," in *2020 International Conference on Recent Trends on Electronics, Information, Communication & Technology (RTEICT)*, Nov. 2020, pp. 227–231. DOI: 10.1109/RTEICT49044.2020.9315558. [Online]. Available: https://ieeexplore.ieee.org/document/9315558 (visited on 13/11/2023) (cit. on p. 41).

[61] A. Waterman and K. Asanovic. "The risc-v instruction set manual, volume i: User-level isa, document version 2.2." (2017) (cit. on p. 41).

[62] "Risc-v." (), [Online]. Available: https://github.com/ultraembedded/riscv (visited on 16/05/2013) (cit. on p. 41).

[63] Augustsson, Lennart, *Microhs: Haskell implemented with combinators*, 2024. [Online]. Available: https://github.com/augustss/MicroHs (cit. on p. 41).

[64] RISC-V Collaborative, *Risc-v gnu toolchain*, 2023. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain (cit. on p. 41).

[65] C.-J. Seger, *The {VossII} Hardware Verification Suite*, 2020. [Online]. Available: https://github.com/TeamVoss/VossII (cit. on p. 42).

[66] D. A. Turner, "A new implementation technique for applicative languages," *Software: Practice and Experience*, vol. 9, no. 1, pp. 31–49, 1979 (cit. on p. 43).

[67] W. R. Stoye, "The implementation of functional languages using custom hardware," University of Cambridge, Computer Laboratory, Tech. Rep., 1985 (cit. on p. 43).

[68] M. Scheevel, "NORMA: A graph reduction processor," en, in *Proceedings of the 1986 ACM conference on LISP and functional programming - LFP '86*, Cambridge, Massachusetts, United States: ACM Press, 1986, pp. 212–219, ISBN: 978-0-89791-200-6. DOI: 10.1145/319838.319864. [Online]. Available: http://portal.acm.org/citation.cfm?doid=319838.319864 (visited on 20/11/2023) (cit. on p. 43).

[69] P. J. Koopman Jr and P. Lee, "A fresh look at combinator graph reduction," *ACM SIGPLAN Notices*, vol. 24, no. 7, pp. 110–119, 1989 (cit. on p. 43).

[70] L. Augustsson and T. Johnsson, "The chalmers lazy-ml compiler," *The computer journal*, vol. 32, no. 2, pp. 127–141, 1989 (cit. on p. 43).

[71] S. L. P. Jones, "Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine," *Journal of functional programming*, vol. 2, no. 2, pp. 127–202, 1992 (cit. on p. 43).

[72] M. Naylor and C. Runciman, "The reduceron reconfigured," in *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ser. ICFP '10, New York, NY, USA: Association for Computing Machinery, Sep. 2010, pp. 75–86, ISBN: 978-1-60558-794-3. DOI: 10.1145/1863543.1863556. [Online]. Available: https://dl.acm.org/doi/10.1145/1863543.1863556 (visited on 20/11/2023) (cit. on p. 44).

[73] C. Ramsay and R. Stewart, "Heron: Modern Hardware Graph Reduction," in *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*, ser. IFL '23, New York, NY, USA: Association for Computing Machinery, Jun. 2024, pp. 1–12, ISBN: 9798400716317. DOI: 10.1145/3652561.3652564. [Online]. Available: https://dl.acm.org/doi/10.1145/3652561.3652564 (visited on 28/04/2025) (cit. on pp. 44, 56).

[74] J. H. Kukula and T. R. Shiple, "Building Circuits from Relations," en, in *Computer Aided Verification*, E. A. Emerson and A. P. Sistla, Eds., Berlin, Heidelberg: Springer, 2000, pp. 113–123, ISBN: 978-3-540-45047-4. DOI: 10.1007/10722167_12 (cit. on p. 49).

[75] C. P. Gomes, W.-J. van Hoeve, A. Sabharwal and B. Selman, "Counting CSP Solutions Using Generalized XOR Constraints," en, (cit. on p. 49).

[76] S. M. Plaza, I. L. Markov and V. Bertacco, "Random Stimulus Generation using Entropy and XOR Constraints," in *2008 Design, Automation and Test in Europe*, ISSN: 1558-1101, Mar. 2008, pp. 664–669. DOI: 10.1109/DATE.2008.4484754. [Online]. Available: https://ieeexplore.ieee.org/document/4484754 (visited on 13/03/2025) (cit. on p. 49).

[77] S. Deng, Z. Kong, J. Bian and Y. Zhao, "Self-adjusting constrained random stimulus generation using splitting evenness evaluation and XOR constraints," in *2009 Asia and South Pacific Design Automation Conference*, ISSN: 2153-697X, Jan. 2009, pp. 769–774. DOI: 10.1109/ASPDAC.2009.4796573. [Online]. Available: https://ieeexplore.ieee.org/document/4796573/?arnumber=4796573 (visited on 13/03/2025) (cit. on p. 49).

[78] S. Chakraborty, K. S. Meel and M. Y. Vardi, "Balancing Scalability and Uniformity in SAT Witness Generator," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14, New York, NY, USA: Association for Computing Machinery, Jun. 2014, pp. 1–6, ISBN: 978-1-4503-2730-5. DOI: 10.1145/2593069.2593097. [Online]. Available: https://dl.acm.org/doi/10.1145/2593069.2593097 (visited on 13/03/2025) (cit. on p. 49).

[79] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random simulation," in *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '07, San Jose, California: IEEE Press, Nov. 2007, pp. 258–265, ISBN: 978-1-4244-1382-9. (visited on 13/03/2025) (cit. on pp. 49, 50).

[80] N. Kitchen, "Markov Chain Monte Carlo Stimulus Generation for Constrained Random Simulation," en, Ph.D. dissertation, UC Berkeley, 2010. [Online]. Available: https://escholarship.org/uc/item/6gp3z1t0 (visited on 13/03/2025) (cit. on pp. 49, 50).

[81] J. Yuan, K. Shultz, C. Pixley, H. Miller and A. Aziz, "Modeling design constraints and biasing in simulation using BDDs," in *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*, ISSN: 1092-3152, Nov. 1999, pp. 584–589. DOI: 10.1109/ICCAD.1999.810715. [Online]. Available: https://ieeexplore.ieee.org/document/810715 (visited on 13/03/2025) (cit. on pp. 49, 50, 52).

[82] J. Yuan, K. Albin, A. Aziz and C. Pixley, "Simplifying Boolean constraint solving for random simulation-vector generation," in *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.*, ISSN: 1092-3152, Nov. 2002, pp. 123–127. DOI: 10.1109/ICCAD.2002.1167523. [Online]. Available: https://ieeexplore.ieee.org/document/1167523 (visited on 13/03/2025) (cit. on pp. 49, 50, 52).

[83] Y.-A. C. Randal E. Bryant, "Verification of arithmetic circuits with binary moment diagrams," in *32nd Design Automation Conference*, 1995, pp. 535–541. DOI: 10.1109/DAC.1995.250005 (cit. on p. 52).

[84] K. v. Gleissenthall, R. G. Kıcı, D. Stefan and R. Jhala, "IODINE: Verifying Constant-Time execution of hardware," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1411–1428, ISBN: 978-1-939133-06-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/von-gleissenthall (cit. on p. 55).

[85] P. Borkar, C. Chen, M. Rostami *et al.*, "WhisperFuzz: White-Box fuzzing for detecting and locating timing vulnerabilities in processors," in *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA: USENIX Association, Aug. 2024, pp. 5377–5394, ISBN: 978-1-939133-44-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/borkar (cit. on p. 55).

[86] M. Fourné, D. D. A. Braga, J. Jancar *et al.*, ""these results must be false": A usability evaluation of constant-time analysis tools," in *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA: USENIX Association, Aug. 2024, pp. 6705–6722, ISBN: 978-1-939133-44-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/fourne (cit. on p. 55).

[87] M. S. Alvim, K. Chatzikokolakis, A. McIver, C. Morgan, C. Palamidessi and G. Smith, *The Science of Quantitative Information Flow* (Information Security and Cryptography). Cham, Switzerland: Springer, Springer Nature, 2020, ISBN: 978-3-319-96129-3. DOI: 10.1007/978-3-319-96131-6. [Online]. Available: http://www.scopus.com/inward/record.url?scp=85091583903&partnerID=8YFLogxK (visited on 15/08/2024) (cit. on p. 55).

[88] C. Barrett, P. Fontaine and C. Tinelli, *The Satisfiability Modulo Theories Library (SMT-LIB)*, www.SMT-LIB.org, 2016 (cit. on p. 55).

[89] C. Kwan and W. A. Hunt, "Automatic verification of right-greedy numerical linear algebra algorithms," in *2024 Formal Methods in Computer-Aided Design (FMCAD)*, 2024, pp. 242–250. DOI: 10.34727/2024/isbn.978-3-85448-065-5_30 (cit. on p. 55).

[90] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ser. ICFP '00, New York, NY, USA: Association for Computing Machinery, Sep. 2000, pp. 268–279, ISBN: 978-1-58113-202-1. DOI: 10.1145/351240.351266. [Online]. Available: https://dl.acm.org/doi/10.1145/351240.351266 (visited on 04/02/2025) (cit. on p. 55).