The Internals of GameTime: Implementation and Evaluation of a Timing Analyzer for Embedded Software

by

Jonathan Prakash Inocencio Kotker

A thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit Seshia, Chair
Professor Edward Lee
Professor Rastislav Bodík

Spring 2013

The thesis of Jonathan Prakash Inocencio Kotker, titled The Internals of GameTime: Implementation and Evaluation of a Timing Analyzer for Embedded Software, is approved:

Chair	______________________________	Date	____________
	______________________________	Date	____________
	______________________________	Date	____________

University of California, Berkeley

The Internals of GameTime: Implementation and Evaluation of a Timing Analyzer for Embedded Software

Abstract

The Internals of GameTime: Implementation and Evaluation of a Timing Analyzer for Embedded Software

by

Jonathan Prakash Inocencio Kotker

Master of Science in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Sanjit Seshia, Chair

Timing analysis is central to the design and implementation of cyber-physical systems. This thesis presents GameTime, a timing analysis toolkit that is based on a combination of game-theoretic online learning and systematic testing using satisfiability modulo theories (SMT) solvers. GameTime can be used to tackle a range of problems related to program timing, including estimating the worst-case execution time, predicting the distribution of execution times, and detecting timing-related anomalies.

This thesis describes the details of the implementation of GameTime. The notion of basis paths is used to handle the exponentially many paths in a program. The issues that arise during the translation of statements in C programs to the equivalent clauses in SMT queries are presented, and the techniques used by GameTime to address these issues are elaborated through examples. Finally, experimental results demonstrate the speed of GameTime analysis and the accuracy of the predictions made by GameTime, with a relative error margin of less than 5% on most of the benchmarks measured.

To Nanay, Tatay and Jason,
and to you, dear reader,
I dedicate this,
my (literal) Masterpiece.

Contents

List of Figures

List of Tables

Acknowledgments

They say that it takes a village to raise a thesis. This thesis is no exception: it benefits from the wonderful contributions of many people over the last three years. I cannot hope to acknowledge all of them explicitly, and I extend sincere apologies to those whose names I could not include for lack of space, time and memory.

I extend the utmost gratitude to my family: my father, Prakash, my mother, Erlinda, and my brother, Jason, whose infinite love, support, advice and patience has kept me going.

This thesis would not exist without the brilliant mentorship of my advisor, Professor Sanjit A. Seshia. His inspirational ideas, his unwavering motivation and his enthusiastic spirit created a great environment to perform cutting-edge research in. I am also grateful for the encouragement from Professor Edward Lee and Professor Rastislav Bodík, and for their helpful suggestions and insightful comments on various drafts of this thesis.

I am thankful for the wonderful support, incredible patience and constant encouragement from my friends, including, but *certainly* not limited to, Aditya Adiredja, Nishant Bhat, Carrie Cai, Kristine Chen, Long Cheng, Alexis He, Kaushik Iyer, Shrivats Iyer, Angela Juang, Anita Kalathil, Eric Kim, Tiffany Kwak, Jillian Moore, Nghi Lam, Brian Lin, Tom Magrino, Seshadri Mahalingam, Ankur Mandhania, Karan Malik, Michael Matloob, Olga Matveeva, Shaan Mulchandani, Varun Pai, Wei Peng, William Pe, Sunil Pedapudi, Evan Pu, Vyoma Shah, Brian Suh, Peyam Tabrizian, Hoa-Long Tam, Kevin Tran, Anirudh Todi, Brandon Wang, George Wang, John Wang, Wei Yeh, Evelyn Yung and Bonnie Zeng.

I would also like to acknowledge the good advice and good times provided by the talented and inspirational people in the Donald O. Pederson center, including my colleagues in Professor Seshia's group: Bryan Brady, Alexandre Donzé, Ruediger Ehlers, Luigi Di Guglielmo, Yen-Sheng Ho, Daniel Holcomb, Susmit Jha, Garvit Juniwal, Wenchao Li, Dorsa Sadigh, Rohit Sinha, Tan Wei Yang, Nishant Totla and Zach Wasson.

The GameTime toolkit benefits from the efforts of incredible students, including Andrew Chan, James Ferguson, Sagar Jain, Johny Lam, Jacob Levine, Min Xu and Lisa Yan. I am also thankful to the people of Chrona, the Toyota Technical Center and the Universität Salzburg, including Professor Wolfgrang Pree, Kenneth Butts, Silviu Craciunas, Jyotirmoy Deshmukh, Vineet Loya, Stefan Lukesch, Erhard Pointl, Stefan Resmerita, Koichi Ueda and Hakan Yazarel, for providing a platform to work on and a place to stay in Los Angeles.

I am also indebted to Babak Ayazifar for his excellent guidance and his infectious attention to detail, to Kayyum Mansoor for his worldly advices and his boundless kindness, and to my extended family and friends in Oman, India and the Philippines, especially Bina Malhotra and Neeta Sankarani, without whose guidance and faith I would never have found the courage to apply to Cal, and Susan Joseph and Amelia Merry, who have provided remote advice and reassurance over the years.

Finally, I am eternally grateful to the divine forces that have brought me to the thriving and dynamic community at Cal. It is an honor to have been inspired, enthralled and encouraged by my instructors, peers and students over these past seven years. Thank you, and please always eat your vegetables.

Chapter 1

Introduction

Cyber-physical systems, or *embedded computer systems*, are systems where computation is tightly integrated with the physical world [46, 30]. The physical properties of these increasingly ubiquitous systems, such as those relating to execution time or energy consumption, are thus determined significantly by the programs that run on them, and by the platforms that these programs run on. As a result, the quantitative analysis of these programs is crucial to the design, implementation and appraisal of these systems.

Timing analysis is a particularly important type of quantitative analysis that addresses several kinds of problems that arise in practice [45]. A classic problem is to estimate the worst-case execution time (WCET) of a terminating software task. The solutions to this problem are relevant in verifying if a deadline can be met for all of the possible inputs to the program: for example, it would be very useful to know if the brake-by-wire software in a car always actuates within a pre-specified time frame. The knowledge of the WCET also informs effective and efficient scheduling strategies. A related problem is the discovery of a test case whereby a task exhibits anomalous timing behavior: for example, the test case could cause the task to miss a deadline.

Another problem in timing analysis is estimating the distribution of the execution times that can be exhibited by a task. Finally, in "software-in-the-loop" simulations, the software implementation of a controller is simulated along with a model of the plant that it controls. These simulations are connected through execution time estimates. For the purposes of scalability, these simulations must be performed on a workstation, and not directly on the target platform. Hence, for the workstation-based simulations, it is important to predict the timing of a particular execution path on the target platform.

All of the problems described so far are variants of the same problem: given a program that terminates on all inputs, predict a particular execution time property. One approach to these problems is implemented in GameTime, a toolkit for the timing analysis of software. GameTime can predict not only extreme-case timing behavior, but also the timing along particular execution paths of a program, and thus several execution time statistics.

The theory behind GameTime has already been established in several existing papers, e.g., [46, 47]. This thesis describes different aspects of the implementation, along with an extensive experimental evaluation, that have not been described in those papers. The GameTime approach relies upon the ability to model execution paths through embedded C programs as queries made to a satisfiability modulo theories (SMT) solver. This approach, however, is complicated by the constructs of the C language that permit, for example, unions and pointer aliasing. This thesis presents a few algorithms and heuristics that attempt to encode many of the constructs and semantics of the C language as closely as possible, while allowing a straightforward translation of a path into an equivalent query to an SMT solver. The encoding strategies that are described in this thesis can also be useful for program verification algorithms or testing methodologies that rely on SMT solving.

In Chapter 2, the theory behind GameTime is briefly reiterated, and the relevant notion of *basis paths* is described through examples. Chapter 3 explores various steps of the GameTime toolflow. It also poses the challenges described in the preceding paragraph, as well as the details of the implementation that tackle these challenges. Experimental evidence of the accuracy of GameTime is presented in Chapter 4, while Chapter 5 concludes the thesis and presents avenues for future work.

To summarize, this thesis makes the following novel contributions:

- New modeling strategies that encode C constructs into SMT queries, such as unions, struct aliasing and pointer aliasing.
- Exploration and evaluation of various implementation options of the GameTime algorithm that permit, for example, different SMT solvers, different ILP solvers and basis randomization.
- Application of the GameTime algorithm to other quantitative properties, specifically the problem of estimating the energy consumption of a program.

1.1 Related Work

The estimation of execution time is a vast and active field of research. WCET analysis is comprehensively surveyed by Li and Malik [31] and Wilhelm et al [50]. Seshia and Rakhlin compare many of the approaches in these surveys to the GameTime approach in [46, 47].

The implementation of GameTime makes significant use of *symbolic execution* to automatically generate test cases. Many symbolic execution tools have been recently constructed to solve a wide variety of problems, such as test generation (e.g., [6]) or bug finding (e.g., [10]). As in GameTime, these tools employ constraint solvers to generate test inputs. However, for many of these tools, one main concern is the potentially exponential number of paths that need to be explored. GameTime addresses this concern by performing symbolic execution for only the basis paths of a program, the number of which scales polynomially as the size of the control-flow graph of the program.

Pointer and alias analysis also feature heavily in the implementation of GameTime. A survey of different analysis techniques is presented in [22]. One of the primary motivations of pointer analysis is to determine the set of memory locations that a pointer could point to at a program point, based on the paths that could lead to that program point. GameTime, however, only has to deal with one path from a small subset of program paths at a time. Consequently, it can significantly simplify its pointer analysis, while also leveraging the notion of basis paths to reason about properties of all the paths through the program. The pointer analysis performed by GameTime attempts to simplify the queries sent to the SMT solver, without having to provide supplementary axioms as the SLAM2 analysis engine from Microsoft does [3]. This allows GameTime to 'plug-and-play' an SMT solver: any SMT solver that works with the `QF_AUFBV` logic can be used with GameTime.

Chapter 2

Theoretical Background

In this chapter, we give a brief overview of the relevant aspects of GameTime, followed by a description of one of its core components, the generation of basis paths.

2.1 Overview

We consider programs P where loops have statically-known finite loop bounds and function calls have known finite recursion depths. Thus, P can be transformed to an equivalent program Q, where every execution path in the (possibly cyclic) control-flow graph of P is mapped one-to-one to a path in the acyclic control-flow graph of Q.

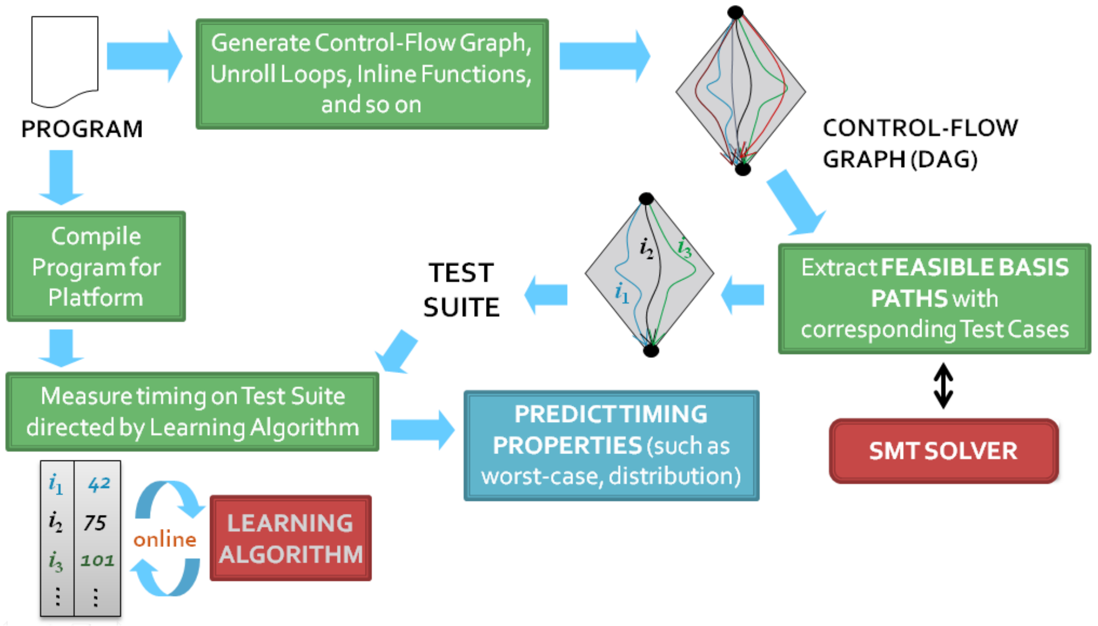

PROGRAM
Generate Control-Flow Graph, Unroll Loops, Inline Functions, and so on
CONTROL-FLOW GRAPH (DAG)
Compile Program for Platform
Extract FEASIBLE BASIS PATHS with corresponding Test Cases
TEST SUITE
i_1
i_2
i_3
SMT SOLVER
Measure timing on Test Suite directed by Learning Algorithm
PREDICT TIMING PROPERTIES (such as worst-case, distribution)
i_1 42
i_2 75
i_3 101
online
LEARNING ALGORITHM

Figure 2.1: **GameTime overview** [45].

Toolflow and Operation

Figure 2.1 depicts the operation of GameTime. The process begins with the generation of the control-flow graph (CFG) of the program. The CFG is assumed to have a single source node (entry point) and a single sink node (exit point); if not, dummy source and sink nodes are added.

The next step is a critical one, where a subset of program paths, called *basis paths*, are extracted. These basis paths form a basis for the set of all paths, in the standard linear algebra sense of a basis. This concept will be explored further in Section 2.2. Symbolic execution is used to generate a satisfiability modulo theories (SMT) formula for each candidate basis path. An SMT solver is invoked to ensure that the basis paths are feasible and to generate the test cases that will drive execution down those paths.

The original program is compiled for the target platform, and executed on these test cases. In the basic GameTime algorithm, described in [46, 47] and reiterated in Chapter 3, the sequence of tests is randomized, with basis paths chosen uniformly at random to be executed. The overall execution time of the program is recorded for each test case. From these end-to-end execution time measurements, the learning algorithm of GameTime generates a weighted graph model that is used for predictions about timing properties of interest.

Game-Theoretic Formulation and Theoretical Guarantees

The theoretical papers on GameTime ([46, 47]) formulate the problem of estimating the worst-case execution time (WCET), and the path in the graph with this execution time, in a game-theoretic manner. The estimation problem is modeled as a two-player game between the estimation tool $\mathcal{T}$ and the environment $\mathcal{E}$ of a program P.

We now briefly reiterate the rules of this game. The game proceeds over several rounds, $t = 1, 2, 3, \ldots$. In each round, the environment $\mathcal{E}$ chooses *path-independent* weights for the edges in the CFG. Concurrently, the tool $\mathcal{T}$ chooses an input to the program P, which drives program execution along a particular path x_t in the CFG. Once $\mathcal{T}$ chooses x_t, $\mathcal{E}$ uses this choice to pick a distribution from which it draws a *path-dependent* perturbation for each edge in the CFG. As a result, the weight of each edge is the sum of its assigned path-independent weight and the path-dependent perturbation. The sum of the weights of the edges along the path x_t are revealed to $\mathcal{T}$ as the length of the path x_t.

In round τ, $\mathcal{T}$ can stop the game and make a guess. $\mathcal{T}$ wins the game in round τ iff it correctly estimates the path that generates the worst-case execution time due to the states of the environment in previous rounds. The goal of $\mathcal{T}$ is thus to select a sequence of inputs $x_1, x_2, \ldots, x_\tau$ that will allow it to identify, with high probability, the longest execution time of P during the previous rounds $t = 1, 2, 3, \ldots, \tau$.

Intuitively, the path-dependent perturbation allows us to model the perturbation in the execution time of a path that arises due to, for example, the input or the measurement error. In practice, this perturbation is not arbitrary, or else prediction would be impossible: for instance, this perturbation can be bounded.

The predictions made by GameTime hold with high probability. The details of theoretical results about the predictions can be found in the theoretical papers on GameTime. While these details are beyond the scope of this thesis, we provide here a less formal and more intuitive description of the theoretical guarantees [28]:

1. Given any $\delta > 0$, GameTime can predict the execution time of any program path to within a tolerance of ε with probability $1 - \delta$ by timing a number of tests that is polynomial in the program size, in $\ln\left(\frac{1}{\delta}\right)$, and in a parameter $\mu_{\max}$.

   $\mu_{\max}$ is an upper bound on the mean perturbation to program path timing due to the variations in basic block time based on the path it lies on. Such a perturbation also includes variation in environmental conditions and measurement error. The tolerance ε is $O(b\mu_{\max})$, where b is the number of basis paths. The greater the mean perturbation, the larger the number of tests needed and the larger the value of ε.

2. For the estimation of the worst-case execution time (WCET), GameTime provides a similar high-probability guarantee on finding the path along which the WCET is exhibited. Once this path is identified, we can execute the path on the target platform and measure its execution time. Thus, if GameTime correctly finds the worst-case path, it accurately computes the WCET.

   More specifically, if the worst-case path timing is larger than the timing of any other path by a margin ρ, which is also $O(b\mu_{\max})$, then GameTime is guaranteed to find the worst-case path with probability $1 - \delta$ by timing a number of tests that is polynomial in the program size, in $\ln\left(\frac{1}{\delta}\right)$, and in $\mu_{\max}$.

2.2 Basis Paths

We explain the basis path generation process using the modular exponentiation code given in Figure 2.2(a). The unrolled version of the code of Figure 2.2(a) for a 2-bit exponent is given in Figure 2.2(b), and is the version of the code used for analysis, but not for measurement.

Modular exponentiation is a necessary primitive for implementing public-key encryption and decryption. In this operation, a base b is raised to an exponent e modulo a large prime number. In this particular example, we use the *square-and-multiply* method to perform the modular exponentiation, based on the observation that

$$b^e = \begin{cases} (b^2)^{e/2} = (b^{e/2})^2, & e \text{ is even,} \\ (b^2)^{(e-1)/2} \cdot b = (b^{(e-1)/2})^2 \cdot b, & e \text{ is odd.} \end{cases} \tag{2.1}$$

In the CFG extracted from a program, nodes correspond to program counter locations, and edges correspond to basic blocks or branches. Figure 2.3(a) denotes the control-flow graph for the code in Figure 2.2(b). Each source-sink path in the CFG can be represented as a 0-1 vector with m elements, where m is the number of edges in the CFG. The ith entry of the path vector is 1 iff the ith edge is on the path; otherwise, the entry is 0.

```
int modexp(int base, int exponent) {
  int result = 1;
  for(int i=EXP_BITS; i>0; i--) {
  // EXP_BITS = 2
    if ((exponent & 1) == 1) {
      result = (result * base) % p;
    }
    exponent >>= 1;
    base = (base * base) % p;
  }
  return result;
}
```

(a) Original code `P`

```
int modexp(int base, int exponent) {
  int result = 1;
  if ((exponent & 1) == 1) {
    result = (result * base) % p;
  }
  exponent >>= 1;
  base = (base * base) % p;
  // unrolling below
  if ((exponent & 1) == 1) {
    result = (result * base) % p;
  }
  exponent >>= 1;
  base = (base * base) % p;
  return result;
}
```

(b) Unrolled code `Q`

Figure 2.2: **Modular exponentation.** Both programs compute the value of `base`$^{\texttt{exponent}}$ modulo `p`.

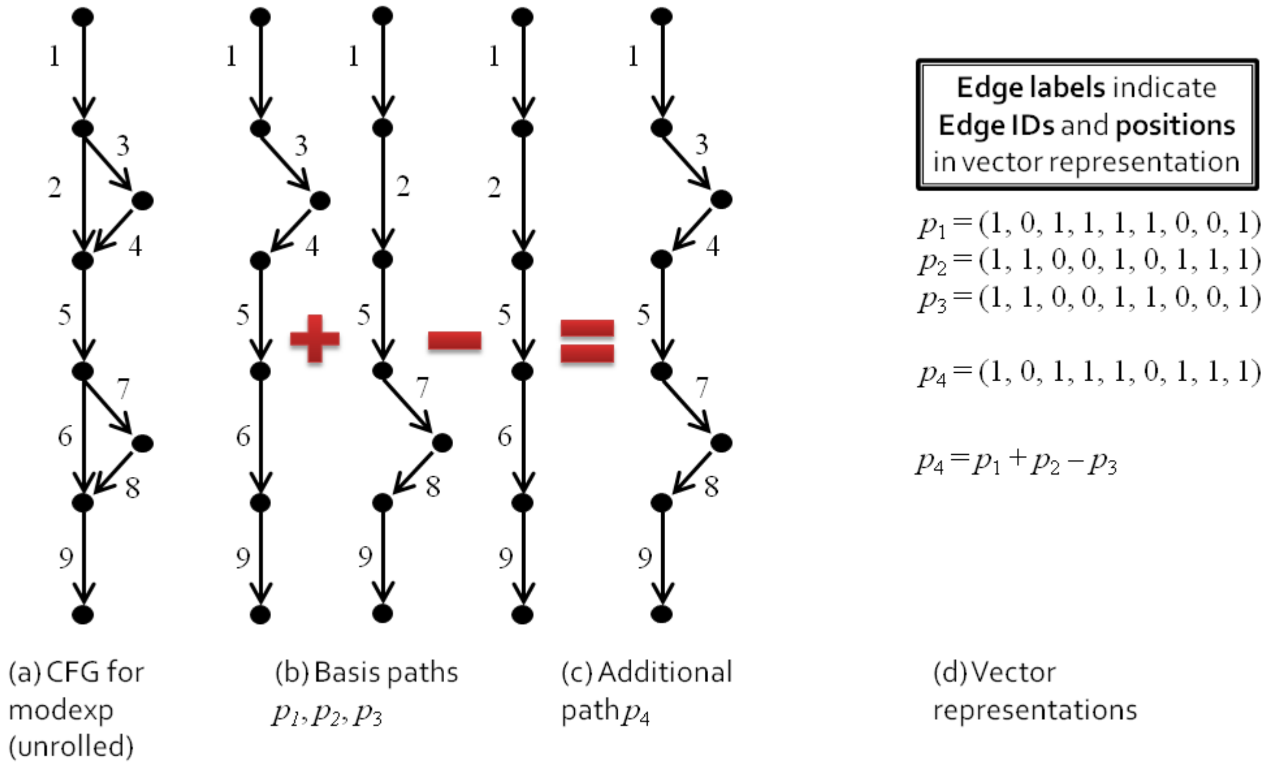

Edge labels indicate Edge IDs and positions in vector representation
$p_1 = (1, 0, 1, 1, 1, 1, 0, 0, 1)$
$p_2 = (1, 1, 0, 0, 1, 0, 1, 1, 1)$
$p_3 = (1, 1, 0, 0, 1, 1, 0, 0, 1)$
$p_4 = (1, 0, 1, 1, 1, 0, 1, 1, 1)$
$p_4 = p_1 + p_2 - p_3$
(a) CFG for modexp (unrolled)
(b) Basis paths p_1, p_2, p_3
(c) Additional path p_4
(d) Vector representations

Figure 2.3: **CFG and Basis Paths for Code in Fig. 2.2(b).**

For example, in the graph of Figure 2.3(a), each edge is labeled with its index in the vector representation of the path. Edges 2 and 3 respectively correspond to the else-branch, valid when the rightmost bit of `exponent` is 0, and the then-branch, valid when the rightmost bit of `exponent` is 1, of the conditional statement at line 3 in the code. Edge 5 corresponds to the basic block that comprises lines 6 and 7. Let $\mathcal{P}$ denote the subset of $\{0,1\}^m$ that corresponds to valid program paths. Note that this set can be exponentially large in m.

A key feature of GameTime is the ability to exploit correlations between paths, which allow the estimation of program timings along any path, by testing a relatively small subset of paths. This subset is a basis of the path-space $\mathcal{P}$, with two valuable properties: any path in the graph can be written as a linear combination of the paths in the basis, and the coefficients in this linear combination are bounded in absolute value.

The first property implies that the basis is a good representation for the exponentially-large set of possible paths; the second says that timings of some of the basis paths will be of the same order of magnitude as that of the longest path. These properties enable us to repeatedly sample timings of the basis paths to reconstruct the timings of all paths.

Figure 2.3(b) shows the basis paths for the graph of Figure 2.3(a). Here p_1, p_2, and p_3 are the paths that correspond to `exponent` taking the integer values with binary representations `01`, `10`, and `00` respectively. Figure 2.3(c) shows the fourth path p_4, expressible as the linear combination $p_1 + p_2 - p_3$ (see Figure 2.3(d)). This path corresponds to `exponent` taking the integer value with binary representation `11`.

The number of feasible basis paths is bounded by $m - n + 2$, where n is the number of nodes in the CFG. Note that our example graph has a "2-diamond" structure, with 4 feasible paths, any 3 of which make up a basis. In general, an "N-diamond" graph with 2^N feasible paths has at most $N + 1$ basis paths.

Chapter 3

Implementation

In this chapter, we discuss the implementation of the GameTime toolflow as presented in Chapter 2. We emphasize the strategies and complications that arise in the discovery of candidate basis paths, and in the modeling of the paths as SMT queries to test their feasibilities. We also briefly describe the other components within GameTime.

For this chapter, the terms *expression* and *statement* are defined as they are conventionally [27], [37]: An expression represents, and evaluates to, a piece of data, such as a number or a character. A statement expresses side effects, such as a function call or the evaluation of an expression, and can direct control flow. We also assume that the starting platform state is fixed and known.

GameTime operates in the five stages enumerated below, which will be further detailed in the sections that follow:

1. Preprocess the program.
2. Extract the CFG of the program.
3. Generate candidate basis paths.
4. Generate test cases.
5. Estimate the weights on the edges of the CFG and the lengths of different paths.

3.1 Preprocessing

Before any analysis is conducted, GameTime preprocesses the program using the CIL (C Intermediate Language) front-end for C [37, 36]. This preprocessed program will be used for the GameTime analysis, but the original program will be used for the measurement of the test cases produced by this analysis.

CIL performs several source-to-source transformations, which reduce the set of all possible constructs in C programs to a smaller, "cleaner" subset. These transformations provably [37] convert a program to an equivalent program that is easier to analyze. For all benchmarks presented in Chapter 4, this phase is very fast, completing in less than a minute.

The transformations performed by CIL [36] that are relevant to GameTime are:

1. *Explicit casting.* CIL inserts explicit casts for all type promotions and conversions.

2. *Function inlining.* As noted in the beginning of Section 2.1, function calls are inlined: the call is replaced with the body of the callee function, and the variables local to the callee are appropriately renamed. This is particularly useful for recursive calls.

3. *Loop detection and unrolling.* Loop detection can already be performed by a submodule within CIL, but not loop unrolling. A custom submodule to perform loop unrolling was thus written for GameTime using the OCaml interface to CIL. GameTime employs the loop detection submodule to generate a *loop configuration file*, wherein each line is the line number of a loop header, followed by the number of times the loop should be unrolled. Once the user edits this second column of numbers, GameTime uses the custom loop unrolling submodule to replace these loops with equivalent loop-free code.

   This extra step of loop detection and unrolling is only performed if GameTime detects loops in the program. The process of loop detection and unrolling is further simplified by the CIL transformation of all looping constructs to the equivalent `while` loops.

4. *Conversion of switch statements.* CIL transforms a switch statement to an equivalent `if`-statement. As a result, every branching point in the transformed code leads to exactly two possibilities. This permits GameTime to compress path representations through the introduction of *default edges*, which will be elaborated upon in Section 3.3.

5. *Conversion to pointer dereferences.* CIL transforms all array expressions and pointer expressions to equivalent expressions that dereference appropriately shifted pointer variables. In the transformed code, the dereference is done on a non-shifted pointer, so a temporary pointer variable is created to store a shifted pointer. For example, if `a` is the name of an array of `int` values, and `b` is an `int` variable, the assignment statement

   ```
   b = a[3];
   ```

   is transformed to the equivalent statements

   ```
   mem3 = a + 3;
   b = *mem3;
   ```

   The newly created temporary variable `mem3` has the type of a pointer to an `int` variable.

   Also, as part of this transformation, statements that dereference pointers more than once are transformed to statements that dereference temporary pointers exactly once.

   For example, the assignment statement

```
b = **a;
```

is transformed to

```
mem3 = *a;
b = *mem3;
```

As before, the newly created temporary variable `mem3` is a pointer to an `int` variable.

Finally, syntactic sugar is eliminated: for example, the expression `c->d` is converted to the equivalent expression `(*c).d`, where `c` is a pointer to a struct that has a field `d`.

A summary of examples of these transformations is provided in Table 3.1. All of these transformations ensure that pointers are either shifted *once* or dereferenced *once* in the analyzed code, which enables easier analysis. The modeling and analysis of these pointer expressions will be elaborated upon in Section 3.4.

Original C statement	Statement as modified by CIL
`b = a[3];`	`mem3 = a + 3; b = *mem3;`
`b = **a;`	`mem3 = *a; b = *mem3;`
`b = *(a + 4);`	`mem3 = a + 4; b = *mem3;`
`c->d = 3;`	`(*c).d = 3;`

Table 3.1: **Summary of some CIL transformations of pointer and array expressions.**

3.2 Extraction of CFG

Once the program has been preprocessed by CIL, the control-flow graph (CFG) can then be extracted using any compiler front-end. Currently, GameTime uses the C# API provided by the Microsoft Phoenix compiler front-end [34].

The CFG that is produced is slightly different from the standard representation: nodes correspond to the starts of basic blocks of the program, while edges indicate flow of control, and are labeled by either a conditional statement or a basic block.

The extraction results in a file that contains the CFG of the program, represented in the DOT language [17], a language maintained by Graphviz [18], open-source graph visualization software. This allows us to visualize the graph, if needed. In our experiments, this phase is also quick, taking less than a minute on the benchmarks used.

3.3 Generation of basis paths

Algorithm

The algorithm to generate a basis for the path space $\mathcal{P}$ was introduced in [2] and described in the context of GameTime in [46, 47]. It is reiterated in Algorithm 1, where it has also been annotated and expanded to include details relevant to the implementation.

The result of the algorithm is a set of basis paths for $\mathcal{P}$, as introduced in Section 2.2. By definition, the set is a spanner: every path in $\mathcal{P}$ can be represented as a linear combination of paths in this set. The set produced is also *2-barycentric*, a term introduced in [2]: a set of b paths $\{b_1, \ldots, b_b\} \subseteq \mathcal{P}$ is 2-barycentric if any path $x \in \mathcal{P}$ can be written as $x = \sum_{i=1}^{b} \alpha_i b_i$ with $|\alpha_i| \leq 2$. This ensures that, as discussed in Section 2.2, the timings of some of the basis paths will be of the same order of magnitude as that of the longest path.

Algorithm 1 Finding a 2-Barycentric Spanner Set for $\mathcal{P}$ [46, 47].

```
 1: $(b_1, \ldots, b_m) \leftarrow (e_1, \ldots, e_m)$
 2: for $i = 1$ to $m$ do                                     Compute a basis for $\mathcal{P}$.
 3:     if there are no more candidate paths to replace row $i$ of $B$ then
 4:         Go to the next iteration
 5:     end if
 6:     $p_i \leftarrow \text{argmax}_{x \in \mathcal{P}} |\det(B_{x,i})|$     Select a candidate path.
 7:     if $p_i$ is feasible then                              Test feasibility using SMT solver.
 8:         Generate corresponding test case
 9:         $b_i \leftarrow p_i$
10:     else                                                   Select another candidate path.
11:         Exclude $p_i$                                      Add constraint to ILP formulation where
                                                               the edges of $p_i$ are never taken together.
12:         Go to line 3
13:     end if
14: end for
15: while $\exists x \in \mathcal{P}, i \in \{1, \ldots, m\}$ such that
    $|\det(B_{x,i})| > 2|\det(B)|$ do                         Transform $B$ into a 2-barycentric spanner.
16:     Generate test case
17:     $b_i \leftarrow x$
18: end while
```

In Algorithm 1, m is the number of edges in the CFG of the program under analysis. It can be shown that the running time of the algorithm is quadratic in m [2].

$B = [b_1, \ldots, b_m]^\mathsf{T}$ is known as the *basis matrix*. B is initialized such that its ith row is e_i, the m-dimensional vector with 1 in the ith position and 0 in the other positions. GameTime also supports an option that permits a user to randomly swap the rows of the basis matrix after initialization and before the replacement of rows. For the purposes of this description, however, we do not consider this option, although the algorithm is still applicable in the presence of this randomization.

The ith iteration of the `for`-loop in lines 2 through 14 attempts to replace the ith row of the basis matrix with a path that is linearly independent of the other vectors in the matrix, including the previous $i-1$ basis paths identified so far. In line 6, $B_{x,i}$ is (a copy of) the basis matrix B, where the ith row has been replaced with the vector x. Formally, $B_{x,i} = [b_1, \ldots, b_{i-1}, x, b_{i+1}, \ldots, b_m]^{\mathsf{T}}$. The algorithm attempts to find a vector x that will maximize the determinant of $B_{x,i}$, which is then assigned as candidate path p_i. As explained in a later subsection, finding this vector is equivalent to solving an integer linear program. The intuition behind this step arises from viewing determinant computation as the computation of the volume of a polytope. Maximizing the determinant is equivalent to spreading the vertices of this polytope as far as possible to obtain a "diverse" set of basis paths [47].

There might not be such a candidate path p_i, which could arise in three cases:

(a) All paths in the path space $\mathcal{P}$ have already been considered.

(b) The quantity $\det |B_{x,i}|$ is zero, or below some specified (small) threshold.

(c) The number of candidate paths that have been attempted as replacements for the row has exceeded some specified (large) threshold.

In these cases, as per lines 4 and 5, the algorithm skips the current iteration of the `for`-loop, and the row is labeled as a "bad" row. In the implementation, the row is also moved to the bottom of the basis matrix, and the `for`-loop bounds are modified to ignore these "bad" rows. This movement does not, however, change the absolute value of the determinant of the matrix. As will be discussed in Section 3.5, the movement also does not change the predicted lengths of different paths in the CFG.

Otherwise, if there does exist a candidate path p_i, the statements along the path are collected and modeled as an SMT query. A SMT solver is then invoked to check the satisfiability of the SMT query and thus the feasibility of the candidate path. The modeling of the path as an SMT query will be described in Section 3.4.

If the path p_i is infeasible, then it is excluded from consideration, and the ith iteration is repeated, as depicted in lines 10 through 12. If the SMT solver supports the generation of an *unsatisfiable core*, or a subset of clauses in the query whose conjunction is still unsatisfiable, then the portion of the path that is associated with this subset of clauses is excluded instead. The exclusion of a path (or a portion thereof) will be elaborated upon in a later subsection, when the formulation of the integer linear program is detailed.

If the path is feasible, however, the ith row of B is replaced by p_i. The SMT solver also produces a satisfying assignment of variables in the SMT query, which is converted into a test case that drives program execution along p_i.

After the `for`-loop, we are left with a spanner set that may not be 2-barycentric. Lines 15 through 18 refine this set into a 2-barycentric spanner. As described, if there is any row in the basis matrix that can be replaced with a candidate path x, such that the determinant of the new basis matrix is more than twice the determinant of the current basis matrix, then the row is replaced with this path. As implemented, a `while`-loop iterates over the rows of the basis matrix, attempting this replacement of each row. The loop finishes when one pass over the basis matrix does not result in any replacement. In our experiments, this `while`-loop completes the refinement in at most 3 passes. GameTime provides an option to disable this refinement.

The runtime of this phase is dependent on the program under analysis, and the runtimes for various benchmarks are summarized in Chapter 4.

Basis Matrix Compression

The definition of the basis matrix in the previous subsection indicates that the matrix is an $m \times m$ matrix. However, for larger programs, the size of this matrix can be unnecessarily and prohibitively large, and we can optimize the matrix through one observation: all of the paths in the graph share many edges in common.

For example, in Figure 2.3(a), which portrays the CFG of the modular exponentiation code in Figure 2.2(b), all paths contain the edges 1, 5 and 9. This implies that the entries that correspond to these edges will be 1 for each row in the basis matrix, allowing the "compression" of the matrix by removing the resulting columns whose entries are all 1.

We can further optimize by modifying the path representation to use the edges along the path that are taken at each *branch point*, which is defined to be a node with more than one outgoing edge. As noted in Section 3.1, the preprocessing ensures that every branch point is incidental to exactly two outgoing edges. We designate one of these two edges as the *default edge*. This designation is arbitrary: in the current implementation, the edge that corresponds to the `then`-branch is designated the default edge.

Formally, let e_i be the ith edge in the CFG. Let B be the set of branch points in the CFG, labeled P_1 through $P_{|B|}$. We represent a path p as a 0-1 vector with $|B|$ elements. The ith entry of this vector is 1 iff the default edge of P_1 is on the path; otherwise, the entry is 0. Hence, if P_i is incidental to edges j and k, and edge j has been designated as its default edge, then an entry of 1 at the ith position indicates that edge j is on the path; otherwise, edge k is on the path.

This permits the compression of the path representations in Figure 2.3. If edge 3 is designated as the default edge for the first branch point and edge 6 is designated as the default edge for the second branch point, then the paths are represented as $p_1 = (1, 1)$, $p_2 = (1, 0)$, $p_3 = (0, 0)$, and $p_4 = (1, 1)$. The relationships between the paths are maintained: p_4 can still be expressed as $p_1 + p_2 - p_3$.

However, this new representation is a flawed replacement for the original. The basis matrix $B = [p_1, p_2, p_3]^{\mathsf{T}}$ is no longer a square matrix. Also, the presence of a zero vector in the matrix indicates that linear independence of the basis is no longer preserved. In the original representation, the zero vector did not translate to any legal path in the CFG; in this new representation, however, the zero vector does.

To correct for this, we first modify the CFG in a particular way: if the source node is a branch point, we add a dummy source node, along with an edge directed from the dummy source node to the original source node. We can now prepend each vector with an extra entry that denotes if the sole edge from the source node is taken; in other words, the edge from the source node is the designated default edge for the source node. Since there is now only one edge from the source node in the CFG, this entry is always 1. This ensures that the zero vector does not translate to any valid path in the CFG.

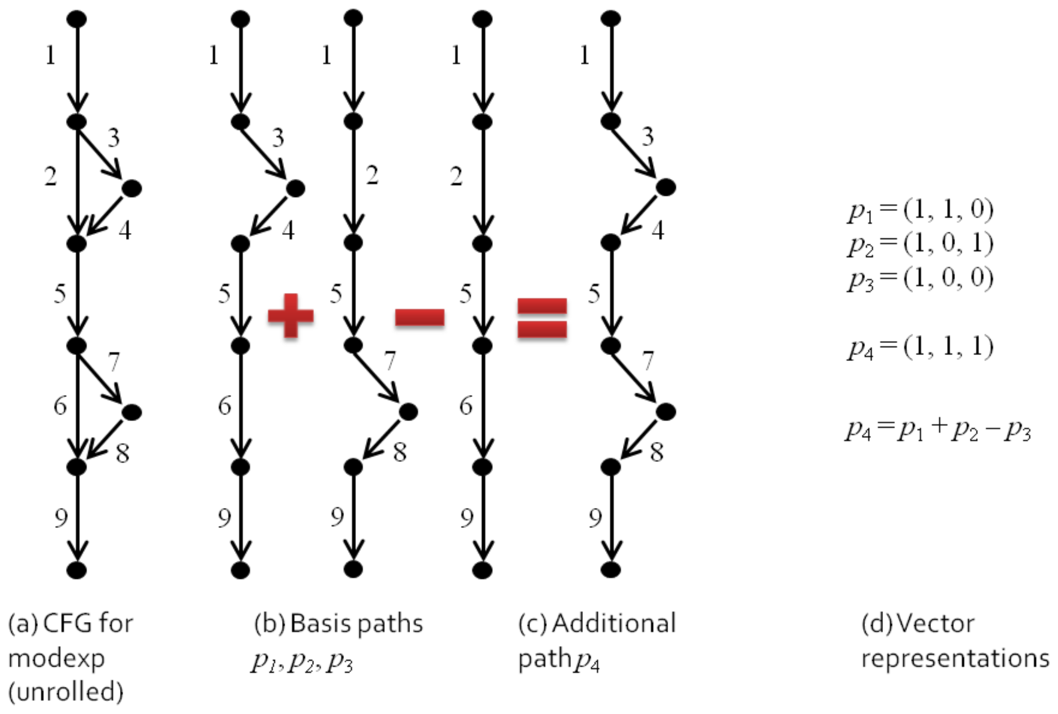

1
2
3
4
5
6
7
8
9
$p_1 = (1, 1, 0)$
$p_2 = (1, 0, 1)$
$p_3 = (1, 0, 0)$
$p_4 = (1, 1, 1)$
$p_4 = p_1 + p_2 - p_3$
(a) CFG for modexp (unrolled)
(b) Basis paths p_1, p_2, p_3
(c) Additional path p_4
(d) Vector representations

Figure 3.1: **CFG and Basis Paths for Code in Fig. 2.2(b), modified.** This demonstrates the path representation employed by the GameTime implementation, which is equivalent to the representation depicted in Figure 2.3.

This is the path representation used by the current implementation of GameTime: one entry for each branch point, and one for the source node. Continuing with the example of modular exponentiation, the basis paths are now represented as $p_1 = (1, 1, 0)$, $p_2 = (1, 0, 1)$, $p_3 = (1, 0, 0)$, and $p_4 = (1, 1, 1)$. This is summarized in Figure 3.1, which is a modified version of Figure 2.3 that demonstrates the new representation. The relationships between the paths are still maintained, and the basis matrix remains a square matrix, although its dimensions have now been reduced from $m \times m$ to $(|B| + 1) \times (|B| + 1)$.

Finally, we compute the value of $|B|$. Let n be the number of nodes in the CFG. Only $n - 1$ of these nodes have outgoing edges. From m, the number of edges, we subtract 1 for each of the $n - 1$ nodes. If a node has two outgoing edges, the second (non-default) one will be counted in the quantity $m-(n-1)$, or $m-n+1$, which is thus the number of branch points in the CFG, or $|B|$. Hence, the dimensions of the basis matrix are $(m-n+2)\times(m-n+2)$, and the maximum number of basis paths is $(m-n+2)$. In practice, since a basis path must be feasible, the number of basis paths b can be less than $(m-n+2)$. In fact, the non-"bad" rows of the basis matrix, of which there must be b, represent these basis paths.

Optimization Formulation

As remarked in the previous subsection, the problem of finding candidate paths that maximize the determinant of a basis matrix can be cast as an integer linear program. In this subsection, we detail the formulation of this integer linear program, as discussed in [31, 30], in the context of GameTime.

As defined before, let m be the number of edges in the CFG of a program, modified slightly as specified in the previous subsection, and let n be the number of nodes. We number the nodes from 1 through n and the edges from 1 through m; in particular, let the sole edge from the source node be edge 1. Let B be the set of branch points in the CFG, labeled P_1 through $P_{|B|}$.

Let e_i be an integer variable that denotes the number of times edge i is traversed along a path. Alternatively, to use the terminology of the theory of network flow, let e_i be the *flow* through edge i. Since the CFG is an acyclic graph, e_i can only take on the values of 0 and 1. To find a candidate path in the CFG, we enforce *flow constraints* on these variables e_i. These constraints arise from the structure of the CFG and from the paths that have already been considered.

1. *Unit flow at source.* There is unit flow along the edge from the source of the CFG. If this edge is k, then this constraint is expressed as the equality

$$e_k = 1.$$

For example, for the CFG presented in Figure 2.3(a), this constraint would be

$$e_1 = 1.$$

2. *Conservation of flow.* For each node i, the amount of flow that enters must equal the amount of flow that leaves. This constraint is expressed as the set of equalities

$$\forall i,\ 1 \le i \le n \qquad \sum_{k\in I_i} e_k = \sum_{\ell\in O_i} e_\ell,$$

where I_i is the (possibly empty) set of the numbers of the edges that enter node i and O_i is the (possibly empty) set of the numbers of the edges that leave node i. For example, for the CFG presented in Figure 2.3(a), these constraints would be

$$\begin{aligned} e_1 &= e_2 + e_3 \\ e_3 &= e_4 \\ e_2 + e_4 &= e_5 \\ e_5 &= e_6 + e_7 \\ e_7 &= e_8 \\ e_6 + e_8 &= e_9 \end{aligned}$$

3. *Path-exclusive constraints.* If a path (or a portion of a path) is excluded, then the edges in the path (or a portion) should not be taken together. Let E_{ex} be a set of the numbers of these edges that cannot be taken together. This implies that no valuation of the variables e_i, $i \in E_{\text{ex}}$, should set all of these variables to 1. This constraint is expressed as the inequality

$$\sum_{i \in E_{\text{ex}}} e_i \leq |E_{\text{ex}}| - 1.$$

For example, for the CFG presented in Figure 2.3(a), if the path p_2, defined in Figure 2.3(b), were to be excluded, perhaps because either it is infeasible or another candidate path needs to be furnished, then edges 1, 2, 5, 7, 8 and 9 should not be taken together. This is ensured by the constraint

$$e_1 + e_2 + e_5 + e_7 + e_8 + e_9 \leq 5.$$

Note that if all edges were taken together, the total flow on the left-hand side of the constraint would be 6, which violates the constraint.

We note that these flow constraints are *linear* in the *integer* variables e_i, and are thus the constraints of an integer linear program. To complete the formulation, we need an objective function. Since we need to maximize $|\det(B_{x,i})|$ for each row i of the basis matrix, we construct an objective function from this quantity.

Let $x = (1, e_{P_1}, e_{P_2}, \ldots, e_{P_{|B|}})$, where e_{P_i} is the variable that represents the flow through the default edge for branch point P_i. The entry of 1 in the first position indicates that the sole edge from the source is always taken, as noted in the previous subsection. With this definition, the quantity $|\det(B_{x,i})|$ results in a linear expression in (a subset of) the variables e_i. This expression is the objective function that we choose to maximize.

A solution of this integer linear program is a valuation of the variables e_i that translates to a (not necessarily feasible) path in the CFG of the program. As will be discussed in Section 3.4, the statements along this path will be collected and converted into the clauses of an SMT query, and an SMT solver will be used to determine the satisfiability of the query, and hence the feasibility of the path itself.

For example, for the CFG presented in Figure 2.3(a), say that we need to find a replacement for the first row of the basis matrix: in other words, we need to maximize the quantity $|\det(B_{x,1})|$. In this case, $x = (1, e_3, e_7)$, where e_3 and e_7 are the default edges for the two branch points in the CFG. The quantity $|\det(B_{x,1})|$ evaluates to the function $1 - e_7$, which is the quantity we choose to maximize. Given the constraints due to the structure of the CFG, the maximum value of this function occurs when e_7 is 0, and the value of e_3 is irrelevant. Hence, there are two possible solutions, represented by the paths p_1 and p_3 defined in Figure 2.3(b).

The current implementation of GameTime uses the PuLP Python package [52] to interface with many well-known integer linear programming solvers, including the GNU Linear Programming Kit (GLPK) [41], the Gurobi Optimizer [40] and the IBM CPLEX Optimizer [25]. The package is used to create and solve the integer linear program.

3.4 Generation of test cases

Once a candidate path has been selected, its feasibility needs to be determined. This is done in three steps, which are briefly described below and detailed in the subsections that follow:

1. Collect the statements along the candidate path.
2. Convert the intermediate expressions to the clauses of an SMT query.
3. Use an SMT solver to check the satisfiability of the SMT query. If the query is satisfiable, parse the resulting model and create a test case that drives execution along the path.

Step 1: Path statement collection

Microsoft Phoenix internal representation

This step is performed using the Microsoft Phoenix front-end [34], which permits many different kinds of C program analyses. The current implementation of GameTime uses the C# API of Phoenix to traverse the internal representation used by the front-end. An example of this internal representation is presented in Figure 3.16, which depicts the (unrolled) modular exponentiation code given in Figure 2.2(b). We note some features:

1. Each line of the internal representation is an *instruction*, which may have an *operator* and *operands*.
2. Each operand maintains type and size information about the corresponding variable or expression in the original C program. Each instruction maintains the number of the associated line in the C program.
3. The representation is in *static single assignment* (SSA) form [1], where each operand is assigned to exactly once. If an operand is assigned to more than once, a new "version" of the operand is created for each new assignment after the first. Each

new operand is suffixed with a number. For example, in Figure 3.16, the destination operand `_result<4>` on line 8 is a new version of the operand `_result`. This makes *def-use* information explicit: each operand is defined in exactly one instruction (the *def-instruction*) and used in at most one instruction (the *use-instruction*).

4. An operand that corresponds to a C variable is named the same as the variable, with an underscore prefix. Temporary operands, whose names are numbers prefixed with `t`, are assigned the results of instructions that correspond to sub-expressions within a complex C expression.

5. Operands also exist for dereferenced pointers. As an example, if `t283` is an operand that represents a pointer, then the operand `[t283]*` represents the dereferenced pointer.

6. There are many kinds of instructions, as described in the documentation for Microsoft Phoenix [33], a few of which are relevant to GameTime:

   (a) *Value instructions* are instructions that compute a value. They correspond to sub-expressions in a C program. The value is stored in a *destination operand*. For example, in Figure 3.16, line 22 is a value instruction:

   ```
   t287                = MULTIPLY _base<2>, _base<2>
   ```

   As indicated, the two *source operands*, both of which are the operand `_base<2>`, are multiplied, and the product is assigned to the destination operand, which is the temporary operand `t287`. This corresponds to the sub-expression `(base * base)` on line 7 of Figure 2.2(b). For our purposes, we define an *assignment instruction* as a value instruction whose destination operand is not a temporary operand. This includes instructions whose operator is `ASSIGN`. An example of an assignment instruction is the instruction on line 21:

   ```
   _exponent<6>        = SHIFTRIGHT t286, 1
   ```

   We treat these instructions specially since they directly correspond to assignments in the code. In this case, the instruction corresponds to line 6 of Figure 2.2(b):

   ```
   exponent >>= 1;
   ```

   (b) *Conditional instructions* are instructions that determine the truth value of a condition. They correspond to the conditional expressions of `if`-statements in a C program, and the value is also stored in a *destination operand*. For example, in Figure 3.16, line 10 is a conditional instruction:

   ```
   t283                = COMPARE(EQ) t282, 1
   ```

   As indicated, the equality of the values of the *source operands*, which here are `t282` and `1`, is tested, and the truth value is stored in the temporary operand `t283`. This corresponds to the condition `((exponent & 1) == 1)` on line 3 of Figure 2.2(b), where the value of the sub-expression `(exponent & 1)` is stored in the temporary operand `t282`, as defined on line 9 of Figure 3.16.

(c) *Label instructions* designate the start of a basic block, and provide a destination for branch instructions. For example, in Figure 3.16, the instruction on line 18 is a label instruction:

```
$L6: (references=2)
```

It heads the basic block that contains the C statements between the two `if`-statements in Figure 2.2(b), from lines 6 through 10. The `references` annotation indicates the number of the branch instructions this label is the destination of.

(d) *Branch instructions* account for the control flow in the program, and allows program execution to jump to different basic blocks, based on the truth value of an operand. For example, in Figure 3.16, the instruction on line 11 of Figure 3.16 is a branch instruction:

```
CONDITIONALBRANCH(True) t283, $L7, $L6
```

As indicated, if the value of the temporary source operand `t283` is true, then the next basic block executed will be the one headed by the label `$L7` (on line 12); otherwise, it will be the one headed by the label `$L6` (on line 18).

Branch instructions include `GOTO`-instructions, which cause a direct jump to another basic block without checking the truth value of any condition. For example, the instruction `GOTO $L8` on line 34 causes a jump to the basic block headed by the label `$L8` (on line 35).

(e) The SSA form introduces `PHI`-instructions, which represent ϕ-functions. These `PHI`-instructions, like the ϕ-functions they represent, provide a distinction between the possible values of a variable that result along different control flow paths. For example, in Figure 3.16, the instruction

```
_result<11>        = PHI _result<4>, _result<5>
```

on line 19 indicates that the operand `_result<11>` can be assigned the value of either the operand `_result<4>`, defined on line 8, or the operand `_result<5>`, defined on line 15, based on whether the branch instruction on line 11 caused program execution to jump to label `$L7` or to label `$L6`, respectively.

GameTime intermediate expressions

GameTime performs a symbolic execution of the instructions in the Phoenix internal representation that represent the statements along the candidate path. In particular, Phoenix exposes an API that allows the backward symbolic execution of the operands in these instructions. When presented with each of these instructions, GameTime uses this API to trace each operand to its def-instruction, and to construct its own intermediate expression that represents the instruction. This algorithm will be detailed later in this subsection.

The intermediate expressions produced by GameTime closely resemble expressions in C, but also permit a simple conversion to the clauses of an SMT query, while providing an accurate and logical memory model. The Backus-Naur Form grammar of these expressions is shown in Figure A.1 of Appendix A.

```
1  int modexp(int base, int exponent) {
2    int result = 1;
3    if ((exponent & 1) == 1) {
4     result = (result * base) % p;
5    }
6    exponent >>= 1;
7    base = (base * base) % p;
8    if ((exponent & 1) == 1) {
9      result = (result * base) % p;
10   }
11   exponent >>= 1;
12   base = (base * base) % p;
13   return result;
14 }
```

(a)

```
result == 1

(exponent & 1) == 1

result<1> == (result * base) % p

exponent<1> == (exponent >> 1)

base<1> == (base * base) % p

!((exponent<1> & 1) == 1)

exponent<2> == (exponent<1> >> 1)

base<2> == (base<1> * base<1>) % p
```

(b)

Figure 3.2: **Examples of GameTime intermediate expressions.** (a) Modular exponentiation code of Figure 2.2, where the statements along one path are highlighted. (b) The GameTime intermediate expressions that correspond to these statements.

An example of these intermediate expressions is presented in Figure 3.2. Figure 3.2(a) shows the (unrolled) modular exponentiation code from Figure 2.2(b), with the statements along the path p_1, as defined in Figure 2.3(b), highlighted with a light gray background. The body of the second `if`-statement (on line 8) has not been highlighted, since the edge that corresponds to the `then`-branch is not included in the path. Figure 3.2(b) depicts the GameTime intermediate expressions that correspond to the statements along the path. This example demonstrates two characteristics of these intermediate expressions:

1. As with the intermediate representation of Microsoft Phoenix, the intermediate expressions generated by GameTime are in SSA form. As a result, since every variable in these expressions is assigned exactly once, assignments are converted into equalities. For example, in line 6 of Figure 3.2(a),

   ```
   exponent >>= 1;
   ```

the variable `exponent` is assigned the result of a right-shift by 1 bit. This statement is converted to the GameTime intermediate expression

```
(exponent<1> == exponent >> 1)
```

with a new version of the variable `exponent`.

2. If a path does not contain the edge that corresponds to the `then`-branch of an `if`-statement, the condition must evaluate to false. To enforce this, GameTime inverts the intermediate expression that represents the condition. For example, in Figure 3.2(a), the `then`-branch of the second `if`-statement (on line 8) does not lie along the path. As a result, its condition `(exponent & 1) == 1` is converted to the GameTime intermediate expression

   ```
   !((exponent<1> & 1) == 1)
   ```

   where the `!` operator serves to invert the Boolean result of its argument expression. Note that the appropriate version of the variable `exponent` has been used, since it has already been assigned once.

Definitions

For the rest of this subsection, we define the function $\textsc{type}(\cdot)$, which maps a GameTime intermediate expression to the C data type of the corresponding C expression in the original program. If the corresponding C expression has a primitive type, such as `char` or `int`, this function evaluates to that type. For example, in Figure 3.2(a), if `exponent` is an integer variable, then both $\textsc{type}(\texttt{exponent})$ and $\textsc{type}((\texttt{exponent >> 1}))$ evaluate to `int`.

If, however, the corresponding C expression is a pointer, $\textsc{type}(\cdot)$ evaluates to $\texttt{int} \to \tau$, where τ is the type of the variable that the pointer points to. For example, if p is declared with the type `int*`, then $\textsc{type}(\texttt{p}) = \texttt{int} \to \texttt{int}$. This definition is recursive: if p is declared with the type `char**`, then $\textsc{type}(\texttt{p}) = \texttt{int} \to \texttt{int} \to \texttt{char}$. This definition indicates that a pointer can be considered a function that maps an integer (the address) to another variable, which can itself be a pointer [8].

We also define the function $\textsc{sizeof}(\cdot)$, which maps a GameTime intermediate expression to the number of bits that the value of the corresponding C expression would occupy in memory. For example, on a machine where integers occupy 4 bytes of memory, both $\textsc{sizeof}(\texttt{exponent})$ and $\textsc{sizeof}((\texttt{exponent >> 1}))$ evaluate to 32.[1] We similarly define the function $\textsc{sizeof}_t(\cdot)$, which maps a type, as returned by the function $\textsc{type}$, and returns the number of bits that a variable of that type would occupy in memory. For example, on the same machine, $\textsc{sizeof}_t(\texttt{int}) = 32$. By definition, we note that, for some variable v,

$$\textsc{sizeof}_t(\textsc{type}(v)) = \textsc{sizeof}(v)$$

[1]This function is different from the `sizeof` function in C, which returns the *byte*-size.

The types and sizes of the expressions are obtained from the type and size information maintained by the corresponding Phoenix operands. As a result, the domains of these functions are extended to include Phoenix operands.

Finally, we define the semantics of the offset operator (`.`). The syntax of an intermediate expression that uses this operator is

$$\textit{expr}\,\texttt{.}\,\textit{offset}$$

where *expr* and *offset* are themselves intermediate expressions. If the size of this expression is s, then this expression evaluates to the value whose bit representation is located between, and including, bits *offset* and $\textit{offset} + s - 1$, where bit 0 is the least significant bit. This definition proves useful, as we shall describe, in the modeling of arrays and structs. If the size of *expr* is also s, then the expressions *expr*`.0` and *expr* evaluate to the same value.

Modeling and translation of C constructs

An overview of the symbolic execution algorithm that GameTime uses to construct its intermediate expressions is presented in Algorithm 2. Though the algorithm as described is specific to the internal representation employed by Microsoft Phoenix, it can be easily adapted to any other internal representation used by other compiler front-ends, such as CIL itself and the LLVM compiler infrastructure [49].

Algorithm 2 Constructing GameTime intermediate expressions for a path.

```
1: function ConstructExpressions($p$)
2:     return [ConstructExpression($inst$, $p$) for each instruction $inst$ along $p$]
3: end function
```

The input to Algorithm 2 is the path that results from solving the integer linear program formulated in the previous step, but in an alternate representation. The solution of the program is a list of edges along the path: GameTime maps this list to a list of the IDs of the corresponding Microsoft Phoenix basic blocks, which is the representation used. This enables Algorithm 2 to collect the instructions along the input path p, merely by sequentially collecting the instructions that these basic blocks comprise. Line 2 of Algorithm 2 borrows the list comprehension notation from Python: as depicted, the output is a list of the non-null outputs of the function ConstructExpression when applied to each instruction.

Algorithm 3 defines the function ConstructExpression, with details omitted for the sake of brevity. The input is a Microsoft Phoenix instruction and the path p that contains the instruction *inst*, and the output is either a GameTime intermediate expression that represents the input instruction or a null value. The function heavily uses a helper recursive function TraceOperand, which is defined in Algorithm 4. This helper function takes an operand o of an instruction and the path p that contains the instruction as inputs, and returns the GameTime intermediate expression that corresponds to the operand.

Algorithm 3 Constructing a GameTime intermediate expression.

```
1: function ConstructExpression($inst$, $p$)
2:     if $inst$ is a branch instruction then
3:         $conditional\_expr \leftarrow$ TraceOperand(source operand, $p$)
4:         if basic block headed by the second (false) label is in $p$ then
5:             $conditional\_expr \leftarrow$ !($inst$)
6:         end if
7:         return $conditional\_expr$
8:     else if $inst$ is an assignment instruction then
9:         $src\_exprs \leftarrow$ [TraceOperand($s, p$) for each source operand $s$]
10:        Construct expression $e$ using $src\_exprs$ and operator of $inst$
11:        $dest\_expr \leftarrow$ TraceOperand(destination operand, $p$)
12:        if destination operand is a pointer or a struct then
13:            AddToAliasTable($dest\_expr$, $e$)
14:            return null
15:        end if
16:        return equality expression between $dest\_expr$ and $e$
17:    else
18:        return null
19:    end if
20: end function
```

The function TraceOperand "expands" an operand into the approriate sub-expression in the C program. It traces a temporary operand back to its def-instruction, and recursively traces other temporary operands that may appear in this def-instruction. The base case occurs when the operand is not temporary: the operand represents a variable in the C program. In this case, the function returns an expression with the original variable name.

The function ConstructExpression returns a null value if the input instruction is neither a branch instruction nor an assignment instruction. If, however, the instruction is a branch instruction, the source operand of the instruction is traced backwards, as noted on line 3 of Algorithm 3, which results in a conditional expression. If the conditional expression were false, then program execution would jump to the basic block headed by the second label. Hence, if this basic block were in p, then the condition should be falsified. As specified earlier, this is achieved by using the ! operator to invert the value of the conditional expression.

If, however, the instruction is an assignment instruction, the intermediate expressions for the source operands are obtained, as on line 9 of Algorithm 3, and the larger intermediate expression that is associated with the right-hand side is constructed, as on line 10. This construction is straightforward for many kinds of C statements present in embedded systems code. In particular, arithmetic expressions, expressions with bitwise operators, and Boolean expressions afford a direct translation to their GameTime expression counterparts. A table that maps different Phoenix operators to their GameTime counterparts is provided in

Algorithm 4 Tracing an operand of an instruction.

```
 1: function TraceOperand($o$, $p$)
 2:     $def\_inst \leftarrow$ def-instruction of $o$
 3:     if $o$ is not a temporary operand then
 4:         $result \leftarrow$ original name of variable
 5:     else if $o$ represents the dereference of pointer operand $ptr$ then
 6:         $pointer\_expr \leftarrow$ TraceOperand($ptr$)
 7:         $deref\_expr \leftarrow pointer\_expr$[0].0
 8:         $result \leftarrow$ Lookup(Normalize($deref\_expr$), $o$)
 9:     else if $o$ represents a field in a struct $s$ with offset $offset$ then
10:         $field\_expr \leftarrow s$.$offset$
11:         $result \leftarrow$ Lookup(Normalize($field\_expr$), $o$)
12:     else
13:         $src\_exprs \leftarrow$ [TraceOperand($s, p$) for each source operand $s$ of $def\_inst$]
14:         $result \leftarrow$ expression constructed using $src\_exprs$ and operator of $def\_inst$
15:     end if
16:     type($result$) $\leftarrow$ type($o$)
17:     sizeof($result$) $\leftarrow$ sizeof($o$)
18:     return result
19: end function
```

Table A of Appendix A. Finally, as on line 16, an equality expression is then constructed between this intermediate expression and the expression obtained for the destination operand.

Nonetheless, there are a few special cases of C expressions that need to be handled differently in these two functions. We describe these below:

(a) *Expressions with division and remainder.* If a C expression is a division expression, it can either be a signed division or an unsigned division. This distinction is made clearer in the intermediate expressions produced by GameTime: if the division performed is unsigned, the operator `/u` is used; otherwise, the operator `/s` is used.

Also, if a C expression is either a division expression or a remainder expression, an additional intermediate expression is constructed to ensure that the second operand is not zero. For example, the intermediate expressions for the C expression `a % b` are

```
a % b
b != 0
```

(b) *Expressions with casting between primitive data types.* As mentioned in the preprocessing described in Section 3.1, CIL supplies explicit casts for all type conversions and promotions. If a statement contains a cast from a variable of one primitive data type to another, the associated Phoenix instruction uses the `CONVERT` operator. GameTime translates instructions with this operator into either a bit extraction expression or a bit extension expression: if a variable `a` is cast to a variable `b`, and SIZEOF(`a`) > SIZEOF(`b`), the result is a bit extraction expression; if SIZEOF(`a`) < SIZEOF(`b`), the result is a bit extension expression. This translation allows GameTime to represent the variables as bitvectors, when the expressions are converted to the clauses of an SMT query. It also helps GameTime model the C semantics of casting between primitive types.

For example, let `a` be an `int` variable that is cast to a `char` variable, as in the C statement

```
char b = (char)a;
```

If `char` variables occupy 1 byte on a little-endian machine, while `int` variables occupy 4 bytes, this statement is converted into a bit extraction expression:

```
(b<1> == bitExtract(7, 0, a))
```

This expression indicates that bits 0 through 7, where 0 is the position of the least significant bit, of the `int` variable `a` are assigned to (a new version of) the smaller `char` variable `b`. GameTime supports an option that allows a user to specify the endianness of the environment where the test cases will be measured: on a big-endian machine, this expression would extract bits 24 through 31 instead.

For another example, let `a` be a `char` variable that is cast to an `int` variable, as in the C statement

```
int b = (int)a;
```

This statement is converted into a bit extension expression:

```
(b<1> == signExtend(a, 24))
```

This expression signifies that the variable `a` should be sign-extended by 24 bits before its assignment to (a new version of) the variable `b`. This bit extension can be a zero-extension if the smaller variable is unsigned: in this case, the expression would be

```
(b<1> == zeroExtend(a, 24))
```

(c) *Expressions with arrays and pointers.*

Assumptions: We do not consider statements and expressions where a pointer has been cast to a primitive type. For example, if `p` is an `int` variable and `q` is a pointer to an `int` variable, we discount statements such as `p = (int)q;` or comparison expressions such as `q > 3`, the latter of which CIL transforms to the equivalent expression `(int)q > 3` when making casts explicit. This ensures that pointers do not appear in the intermediate expressions produced by GameTime, although dereferenced pointers, whose value is of a primitive type, may appear. We also do not consider dynamic memory allocation.

Modeling and translation: As described in Section 3.1, the preprocessing performed by CIL reduces array expressions and pointer expressions to equivalent expressions where a pointer variable is either shifted once or dereferenced once. Thus, a statement such as `b = *(a + 3);` or `b = a[3];` where `a` is a pointer to an integer variable and `b` is an integer variable, is converted to the equivalent statements

```
mem3 = a + 3;
b = *mem3;
```

The newly created temporary variable `mem3` has the type of a pointer to an `int` variable.

We can thus assume that subscription expressions, such as `a[b]`, do not show up in the preprocessed code.[2]

As demonstrated above and by the semantics of C, the dereference of a pointer variable `p` is equivalent to the access of the first element of an array named `p`. In other words, the expression `*p`, which returns the value of the variable whose address is stored in `p`, evaluates to the same value as the subscription expression `p[0]`. GameTime can thus treat array expressions and pointer expressions equivalently.

With the introduction of the pointer transformations conducted by CIL, and with the possibility of pointer arithmetic, we are now faced with the challenge of *pointer aliasing*, whereby two or more pointers can point to the same location in memory, and the same location can thus be accessed and modified using any of the pointers.

```
1       int* q = p;
2       *q = 3;
3       if (*p == 3) {
4           return 1;
5       } else {
6           return 0;
7       }
8   }
```

Figure 3.3: **Example of pointer aliasing.** In this example, the variables `p` and `q` point to the same memory location, after the assignment statement on line 3.

[2] CIL keeps subscription expressions if the array that is indexed has a fixed size. These expressions can be replaced either by passing the program through a new custom CIL submodule, or during the symbolic execution conducted in Microsoft Phoenix. The latter is done in the current implementation of GameTime.

Consider the code sample presented in Figure 3.3. The first line of the function `foo` assigns the argument `p`, which is a pointer variable, to another pointer variable `q`. This implies that the variables `p` and `q` now contain the address of the same integer variable. Hence, when `q` is dereferenced and assigned a value, the memory location that `q` points to is assigned the integer value 3. This also implies that the expression `*p` evaluates to 3, and that the function `foo` will always return 1, no matter what the input.

The analysis of pointer aliasing, or *alias analysis*, has been the subject of a large number of studies in the field of program analysis, a survey of which is provided in [22]. Since only one path through a program is being traced, GameTime performs a path-sensitive, context-insensitive, intraprocedural alias analysis.

For the description of this analysis, we first assume assignment statements that adhere to the BNF grammar presented in Figure 3.4. In the grammar, the destination of the assignment statements is a pointer: ⟨*expression*⟩ represents a pointer, and the statements assign one (maybe shifted) pointer to another, which results in pointer aliasing. Notice that the shifts are constant values.

$$\begin{array}{lcl}
\langle \textit{constant} \rangle & ::= & \texttt{[0-9]}+ \\
 & | & \text{`}\texttt{-}\text{'}\ \langle \textit{constant} \rangle \\
 & & \\
\langle \textit{assignment\_stmt} \rangle & ::= & \langle \textit{expression} \rangle\ \text{`}\texttt{=}\text{'}\ \langle \textit{expression} \rangle\ \text{`}\texttt{+}\text{'}\ \langle \textit{constant} \rangle\ \text{`}\texttt{;}\text{'} \\
 & | & \langle \textit{expression} \rangle\ \text{`}\texttt{=}\text{'}\ \langle \textit{expression} \rangle\ \text{`}\texttt{-}\text{'}\ \langle \textit{constant} \rangle\ \text{`}\texttt{;}\text{'} \\
 & | & \langle \textit{expression} \rangle\ \text{`}\texttt{=}\text{'}\ \langle \textit{expression} \rangle\ \text{`}\texttt{;}\text{'}
\end{array}$$

Figure 3.4: **BNF grammar of assignment statements with pointers as destinations.** These assignment statements result in pointer aliasing.

We demonstrate the analysis using the example presented in Figure 3.5(a). In the example, `a`, `b` and `c` are pointer variables that point to `int` variables. The path that is being traced is highlighted. The assignments on lines 2 and 3 are made to pointer variables. On line 2, the pointer `b` is pointed to a location defined in relation to the location that `a` points to. As a result, any expression of the form `b[_]`, where _ denotes any GameTime intermediate expression used as an index, is actually an alias for the expression `a[_ + 1]`. This information is recorded in an *alias table*, as shown in Figure 3.5(b).

An alias table maintains information about expressions that can be replaced by other, equivalent expressions. Hence, any expression of the form `b[_].0` can be replaced by an equivalent expression `a[_ + 1].0`, with any appropriate expression in place of _. The pointer `a` is called the *base* (or *canonical*) pointer for the pointer `b`. This approach allows GameTime to assume that a memory location can be referenced using only one expression, which affords simpler analysis. The role of the offset is irrelevant for this example, but will be clarified in a later example.

```
1 int foo(int* a) {
2     int* b = a + 1;
3     int* c = b + 2;
4     if (*b > *c) {
5         return 1;
6     } else {
7         return 0;
8     }
9 }
```

(a) Example code with pointer aliasing.

Expression	*Replacement Expression*
`b[_].0`	`a[_ + 1].0`
`c[_].0`	`a[_ + 3].0`

(b) Alias table.

`(a[1] > a[3])`

(c) GameTime intermediate expressions that correspond to the statements along the highlighted path.

Figure 3.5: **Demonstration of the GameTime alias analysis.**

On line 3, the pointer variable `c` is pointed to a location defined in relation to the location that `b` points to. Again, this implies that any expression of the form `c[_]` is an alias for the expression `b[_ + 2]`, where _ denotes any GameTime intermediate expression. However, since the expression `b[_]` is itself an alias for the expression `a[_ + 1]`, the expression `c[_]` is transitively an alias for the expression `a[(_ + 2) + 1]`, or `a[_ + 3]`. This information is also recorded in the alias table, as shown in Figure 3.5(b).

Finally, the intermediate expression that corresponds to the condition on line 4 of Figure 3.5(a) is obtained. Since `*b` is equivalent to `b[0]`, and `*c` is equivalent to `c[0]`, the intermediate expression for the condition is `b[0] > c[0]`. The alias table is then used to "correct" the pointer dereferences to produce the expression `(a[1] > a[3])`.

We now provide a more formal description of this analysis. Recall that in Algorithm 3, the intermediate expressions that correspond to the destination operand and to the right-hand side of an instruction are constructed. If the destination operand is a pointer, as detected on line 12 of Algorithm 3, the function AddToAliasTable is invoked to add an entry to the alias table. ConstructExpression itself returns a null value.

The function AddToAliasTable is defined in Algorithm 5. The two arguments are the intermediate expression that corresponds to the destination operand and the expression assigned to the destination. Lines 2 through 6 convert the latter into an addition expression, if it is not one already: this ensures that the augend is an expression that represents the base pointer, and that the addend is a constant expression. Line 7 breaks this addition expression into the base pointer expression and the constant expression.

Line 13 checks if the base pointer expression is already in the alias table; if already present, the replacement expression is used as the base pointer expression instead. This is useful in the case, as demonstrated in Figure 3.5, where a pointer is defined in relation

Algorithm 5 Adding an assignment to the alias table.

```
function AddToAliasTable($dest\_expr$, $src\_expr$)
    if $src\_expr$ is $base\_expr$ - $constant$ then
        $src\_expr$ is $base\_expr$ + $(-1 \times constant)$
    else if $src\_expr$ is $base\_expr$ then
        $src\_expr$ is $base\_expr$ + 0
    end if

    $base\_expr$ + $constant \leftarrow src\_expr$
    if $base\_expr$ is a pointer then
        $base\_expr \leftarrow base\_expr$[_].0
    else
        $base\_expr \leftarrow base\_expr$._
    end if

    if $base\_expr$ is in the alias table then
        $base\_expr \leftarrow$ Replacement expression from the alias table
    end if

    if $dest\_expr$ is a pointer then
        $expr.offset \leftarrow base\_expr$
        $(\texttt{int} \to \tau_{\text{src}}) \leftarrow \text{TYPE}(expr)$
        $(\texttt{int} \to \tau_{\text{dest}}) \leftarrow \text{TYPE}(dest\_expr)$
        $offset \leftarrow offset + (constant \times \text{SIZEOF}(\tau_{\text{dest}}))$ % $\text{SIZEOF}(\tau_{\text{src}})$
        Replace _ in $expr$ with _ + $\lfloor (constant \times \text{SIZEOF}(\tau_{\text{dest}}))$ / $\text{SIZEOF}(\tau_{\text{src}}) \rfloor$
        Simplify $expr$ and $offset$
    end if

    if $dest\_expr$ is a pointer then
        Add $dest\_expr$[_].0 to the alias table, with replacement $expr.offset$
    else
        Add $dest\_expr$._ to the alias table, with replacement $expr.offset$
    end if

end function
```

to another pointer, which itself is defined in relation to another. The rest of the algorithm creates the new replacement expression for the destination expression, and adds an entry between these two expressions to the alias table. Any previously added entry for the destination expression is overwritten.

When a pointer is dereferenced, as on line 5 of Algorithm 4, the intermediate expression that corresponds to the pointer itself is obtained first, as on line 6. For example, if the operand is `[t283]*`, then the expression for the operand `t283` is obtained. An index of `0` and an offset of `0` are attached to this expression to indicate that the pointer is now dereferenced. The function Normalize, to borrow from the terminology of [51], replaces this expression using an entry in the alias table. An overview of this function is provided in Algorithm 6.

Algorithm 6 **Normalizing a dereferenced pointer.**

```
 1: function NORMALIZE($expr$)
 2:     Pattern-match $expr$ against the keys in the alias table
 3:     if matching expression is found then
 4:         return replacement expression, with _ replaced appropriately
 5:     else
 6:         return $expr$
 7:     end if
 8: end function
```

The final step in the trace of an operand that represents a dereferenced pointer is to look up the variable present at the offset in the normalized expression. For this purpose, we define a function Lookup, to once again borrow from the terminology of [51]. This function is invoked on the output of the Normalize function, as on line 8 of Algorithm 4. An overview of this function is provided in Algorithm 7.

Algorithm 7 **Looking up the variable at an offset.**

```
 1: function LOOKUP($expr.offset$, $o$)
 2:     if $offset$ is 0 and SIZEOF($expr$) = SIZEOF($o$) then
 3:         return $expr$
 4:     else if $expr$ is a struct then
 5:         $fields \leftarrow$ Fields of $expr$ between offsets $offset$ and $offset + \text{SIZEOF}(o)$
 6:         if $fields$ has only one field $f$ then
 7:             return bit-adjusted FIELDASARRAY($expr.f$)
 8:         else
 9:             $subscript\_exprs \leftarrow$ [bit-adjusted FIELDASARRAY($expr.f$) for each $f$ in $fields$]
10:             return concatenation of expressions in $subscript\_exprs$
11:         end if
12:     else
13:         return bitExtract($offset + \text{SIZEOF}(o) - 1, offset, expr$)
14:     end if
15: end function
```

As defined, Lookup($expr.offset$, $operand$), for some Phoenix operand $operand$ and expression $expr.offset$, returns an expression whose value has a bit representation that spans positions *offset* through $\textit{offset} + \textsc{sizeof}(\textit{operand}) - 1$ of the bit representation of *expr*. The simplifying assumption is made that $\textit{offset}+\textsc{sizeof}(\textit{operand}) \leq \textsc{sizeof}(\textit{expr})$.

In the examples so far, the operand and its intermediate expression have the same size, and the offset has always been zero. Hence, $\textsc{Lookup}(\textit{expr}.0, \textit{operand})$ evaluates to *expr*, as reiterated on lines 2 and 3 of Algorithm 7. This indicates that the bits from position 0 through position $\textsc{sizeof}(\textit{operand}) - 1$ of the bit representation of *expr* is trivially the bit representation of *expr*.

The role of the offset is clarified through this example:

```
b = (char*)a + 2;
c = *b;
```

where `a` is a pointer to an `int` variable, `b` is a pointer to a `char` variable, and `c` is a `char` variable. Here, `a` is cast to be a pointer to a `char` variable, before being shifted and assigned to `b`. On a little-endian machine where `char` variables occupy 1 byte and `int` variables occupy 4 bytes, this results in `b` pointing to the third byte of the variable that `a` points to. Specified alternatively, the expression `*b` (and also `b[0]`) evaluates to the third byte of `a[0]`.

This case is handled by lines 16 through 23 of Algorithm 5. On lines 20 and 21, the algorithm employs C pointer arithmetic to calculate the new index and bit offset introduced by the addend. In many cases, $\tau_{\text{src}} = \tau_{\text{dest}}$, and 0 is thus added to the offset on line 20. In this example, the destination pointer (`b`) points to variables of type $\tau_{\text{dest}} =$ `char`, while the source pointer (`a`) points to variables of type $\tau_{\text{src}} =$ `int`. The total bit offset (16 bits) is the addend (`2`) multiplied by the bit-size of `char` variables. The pointer `b` thus points to a location 16 bits away from the location that the pointer `a` points to. Since this location is still within the `int` variable (of size 32 bits) that `a` points to, the expression `b[_]` and its replacement expression `a[_].16` are added to the alias table.

When the pointer `b` is dereferenced, the dereference expression `b[0].0` is normalized to the expression `a[0].16`. As shown on line 6 of Algorithm 7, the $\textsc{Lookup}$ function then converts this expression to the intermediate expression

```
bitExtract(23, 16, a[0])
```

As a result, the assignment to the variable `c` is converted to the equivalence expression

```
(c<1> == bitExtract(23, 16, a[0]))
```

If, however, the pointer `b` were dereferenced and then assigned to, as in the statements

```
b = (char*)a + 2;
*b = c;
```

the resulting equivalence expression

```
bitExtract(23, 16, a<1>[0]) == c
```

does not enforce the constraint that the other bits of `a<1>[0]` should remain the same. Without this constraint, the SMT solver could arbitrarily assign the other bits. To prevent this, the *complementary expression* of the bit-extraction expression is also constructed. A complementary expression concatenates the bits that are not present in the bit-extraction expression. For the example above, this additional equivalence expression is constructed:

```
concat(bitExtract(31, 24, a<1>[0]), bitExtract(15, 0, a<1>[0])) ==
    concat(bitExtract(31, 24, a[0]), bitExtract(15, 0, a[0]))
```

The process of translating a Phoenix operand for a dereferenced pointer is summarized in Figure 3.6, using the dereferenced pointer `b` in line 4 of Figure 3.5(a) as an example.

$$\texttt{[t283]*} \longrightarrow \texttt{b[0].0} \xrightarrow{\textsc{Normalize}} \texttt{a[1].0} \xrightarrow{\textsc{Lookup}} \texttt{a[1]}$$

Figure 3.6: **Translation of the Phoenix operand for a dereferenced pointer.** The pointer is `b`, dereferenced in line 4 of Figure 3.5(a). The Phoenix temporary operand `t283` represents `b`.

(d) *Expressions with fixed-length arrays.* If an array is declared to have a fixed length, additional intermediate expressions are generated to constrain the indices in array subscription expressions, to ensure that the array initializations in test cases are "safe", or are performed within the bounds of the array. This is particularly useful if the index is an expression. For example, in the C statements

```
int a[10], b;
if (a[b] > 0) { ... }
```

`a` is declared to be a fixed-length array that can hold 10 `int` variables. As a result, GameTime constructs two expressions that constrain the value of `b`, along with the intermediate expression for the condition itself:

```
a[b] > 0
  b >= 0
  b < 10
```

(e) *Expressions with structs.* In C, structured type objects, or *structs*, are aggregates of a fixed set of named objects, which are possibly of different types [27]. An example is presented in Figure 3.7(a): a new struct type named `my_struct` is defined with two `int` *fields* `a` and `b`. The argument to the function `foo` is a variable `s` with type `my_struct`. The C expressions `s.a` and `s.b` are employed to access the values of the two `int` fields of `s`. Fields can themselves also have a struct type.

```c
typedef struct {
    int a;
    int b;
} my_struct;

int foo(my_struct s) {
    if (s.a > s.b) {
     return 1;
    } else {
        return 0;
    }
}
```

(a) Example code with structs.

```
(_gtF_a[_gtA_s] > _gtF_b[_gtA_s])
```

(b) GameTime intermediate expression that corresponds to the conditional expression along the highlighted path.

Figure 3.7: **Demonstration of the GameTime modeling of structs.**

We demonstrate the GameTime modeling of structs using the conditional expression along the path highlighted in Figure 3.7(a). In Phoenix and GameTime, a field of a struct is represented by its bit-offset from the start of the struct. For example, GameTime converts the C expression `s.a` to the intermediate expression `s.0`, and `s.b` to `s.32`, on a machine where integer variables occupy 32 bits. The expressions themselves have sizes of 32 bits each. In general, when tracing an operand that represents the access of a field f in a struct s, as depicted in line 9 of Algorithm 4, GameTime generates an intermediate offset expression s.*offset*, where *offset* is the bit-offset of f. The size of this offset expression is the size of the operand that is being traced. This representation is one of the few suggested in [51], and allows GameTime to handle many kinds of struct aliasing, as detailed later.

The definitions of the functions Normalize and Lookup, shown earlier in the context of arrays and pointers, are expanded to account for field access expressions. These field access expressions, as shown in the previous paragraph, are offset expressions. In particular, the function Lookup uses the input offset and the size of the input operand to determine the field, or fields, of a struct that are being accessed.

For now, we consider the case where exactly one field of a struct is accessed. GameTime models a struct field access expression as an array subscription expression, using the recursive function FieldAsArray defined in Algorithm 8. In the model, fields are represented by arrays, and structs are represented by integer indices into these arrays. As an example, the expression `s.a` is converted into the expression `_gtF_a[_gtA_s]`, where `_gtF_` and `_gtA_` are prefixes that are added to prevent possible name collisions: the `_gtA_` prefix is added to struct type names[3] and the `_gtF_` prefix is added to the names of the fields within a struct.

Algorithm 8 Modeling a struct field access as an array access.

```
1:  function FieldAsArray(field_expr)
2:      if field_expr is expr.f then
3:          $\tau_f \leftarrow$ TYPE(f)
4:          if $\tau_f$ is a struct type then
5:              $\tau_f \leftarrow$ int
6:          end if
7:          Create array variable _gtF_f with type int $\rightarrow \tau_f$
8:          return _gtF_f[FieldAsArray(expr)]
9:      else
10:         return int variable with the same name as field_expr, prefixed with _gtA_
11:     end if
12: end function
```

As discussed in the modeling of arrays, the type of an array is defined to be `int` $\rightarrow \tau$, where τ is the type of the array elements: this definition is maintained for the arrays that represent fields. The type τ of each element of these arrays is the same as the type of the corresponding field, as instructed by line 7 of Algorithm 8. Each struct is represented by an `int` variable, which is the base case of Algorithm 8. Any inner structs, or structs that are themselves fields of another struct, are also represented by `int` variables, as shown on line 5 of Algorithm 8. Table 3.2 shows examples of field access expressions and the associated GameTime intermediate expressions. The third column declares the types of the variables that appear in the intermediate expressions.

The representation of fields in a struct as numerical offsets from the start of the struct enables GameTime to deal with *struct aliasing*. Consider the code sample presented in Figure 3.8. Two structured types, `my_struct` and `my_other_struct` are defined, and a global variable `p`, of the type `my_struct`, is created. In the function `foo`, a pointer to the variable `p` is created (`&p`). This pointer, which points to a variable of type `my_struct`, is cast as a pointer that points to a variable of type `my_other_struct`. This casted pointer is then dereferenced and assigned to a variable `q` of type `my_other_struct`. The net outcome of this assignment statement, on line 14 of Figure 3.8, is that the struct variables `p` and `q` are aliases for the same (64-bit) region of memory, although the expressions employed to access the values contained in that region is different for either

[3]The `A` in the `_gtA_` prefix signifies that a structured type is an *aggregate* data type.

C expression	*Intermediate expression*	*Expression types*
`a.b`	`_gtF_b[_gtA_a]`	$\text{TYPE}(\texttt{\_gtA\_a}) = \texttt{int}$ $\text{TYPE}(\texttt{\_gtF\_b}) = \texttt{int} \to \texttt{int}$
`a.b[0]`	`_gtF_b[_gtA_a][0]`	$\text{TYPE}(\texttt{\_gtA\_a}) = \texttt{int}$ $\text{TYPE}(\texttt{\_gtF\_b}) = \texttt{int} \to (\texttt{int} \to \texttt{int})$
`a[0].b`	`_gtF_b[_gtA_a[0]]`	$\text{TYPE}(\texttt{\_gtA\_a}) = \texttt{int} \to \texttt{int}$ $\text{TYPE}(\texttt{\_gtF\_b}) = \texttt{int} \to \texttt{int}$
`a.b.c`	`_gtF_c[_gtF_b[_gtA_a]]`	$\text{TYPE}(\texttt{\_gtA\_a}) = \texttt{int}$ $\text{TYPE}(\texttt{\_gtF\_b}) = \texttt{int} \to \texttt{int}$ $\text{TYPE}(\texttt{\_gtF\_c}) = \texttt{int} \to \texttt{int}$
`a.b.c[0]`	`_gtF_c[_gtF_b[_gtA_a]][0]`	$\text{TYPE}(\texttt{\_gtA\_a}) = \texttt{int}$ $\text{TYPE}(\texttt{\_gtF\_b}) = \texttt{int} \to \texttt{int}$ $\text{TYPE}(\texttt{\_gtF\_c}) = \texttt{int} \to (\texttt{int} \to \texttt{int})$
`a.b[0].c`	`_gtF_c[_gtF_b[_gtA_a][0]]`	$\text{TYPE}(\texttt{\_gtA\_a}) = \texttt{int}$ $\text{TYPE}(\texttt{\_gtF\_b}) = \texttt{int} \to (\texttt{int} \to \texttt{int})$ $\text{TYPE}(\texttt{\_gtF\_c}) = \texttt{int} \to \texttt{int}$
`a[0].b.c`	`_gtF_c[_gtF_b[_gtA_a[0]]]`	$\text{TYPE}(\texttt{\_gtA\_a}) = \texttt{int} \to \texttt{int}$ $\text{TYPE}(\texttt{\_gtF\_b}) = \texttt{int} \to \texttt{int}$ $\text{TYPE}(\texttt{\_gtF\_c}) = \texttt{int} \to \texttt{int}$

Table 3.2: **Examples of GameTime modeling of struct field accesses.** The first column shows a struct field access expression, while the second column shows its corresponding GameTime intermediate expression. The third column shows the type of each variable in the intermediate expression.

variable. Here, the expressions `p.a` and `q.c` are aliases for the same location in memory, as are the expressions `p.b` and `q.d`. Nonetheless, the fields `b` and `d`, for example, are at the same location of structs `p` and `q` respectively, and so the C expression `p.b` is represented by the intermediate expression `p.32` (before lookup) and the C expression `q.d` by the intermediate expression `q.32`. This is also true for the fields `a` of `p` and `c` of `q`. This observation permits GameTime to use an alias table to convert accesses of fields in `q` to accesses of fields in `p`.

Struct aliasing is employed in many C programs that run on embedded systems. One of their major uses, as seen in a few benchmarks described in Chapter 4, is to allow different functions in a program to access the contents of the same buffer, but to interpret the contents in different ways and to ignore irrelevant parts of the buffer.

We now demonstrate how GameTime uses an alias table to analyze programs that include struct aliasing. For this demonstration, we assume C statements of the form

$$\tau_{\text{dest}}\ \texttt{q = *((}\tau_{\text{dest}}\ \texttt{*) \&p);}$$

where τ_{dest} is the (structured) type of `q` and τ_{src} is the (structured) type of `p`.

```
 1 typedef struct {
 2     int a;
 3     int b;
 4 } my_struct;
 5
 6 typedef struct {
 7     int c;
 8     int d;
 9 } my_other_struct;
10
11 my_struct p;
12
13 int foo(void) {
14     my_other_struct q = *((my_other_struct *) &p);
15     if (q.c > q.d) {
16         return 1;
17     } else {
18         return 0;
19     }
20 }
```

Figure 3.8: **Example of struct aliasing.** In this example, the variables `p` and `q` refer to the same region of memory, after the assignment statement on line 14.

We also assume that if two structured types have the same numbers and types of fields, which are declared in the same order, then the fields of variables of these structured types are laid out in the same way in memory.

We use the example presented in Figure 3.9(a), which is the same as the code sample shown in Figure 3.8, but with one path highlighted. GameTime uses Algorithm 3, which defines the function ConstructExpression, to construct the intermediate expressions for the statements along the highlighted path.

The assignment on line 14 is made to a struct variable. Since the destination is a struct, as detected on line 12 of Algorithm 3, the function AddToAliasTable is invoked to add an entry to the alias table. As instructed on line 11 of Algorithm 5, where the function is defined, it uses the *base* (or *canonical*) struct variable `p` to generate the replacement expression `p._`. On line 27, the function uses the destination struct variable `q` to produce the alias expression `q._`. An entry is then added to the alias table, as shown in Figure 3.9(b), which signifies that `q._` is an alias for `p._`: any expression that matches `q._` can be swapped out for the expression `p._`, for some appropriate

```
 1 typedef struct {
 2     int a;
 3     int b;
 4 } my_struct;
 5
 6 typedef struct {
 7     int c;
 8     int d;
 9 } my_other_struct;
10
11 my_struct p;
12
13 int foo(void) {
14     my_other_struct q = *((my_other_struct *) &p);
15     if (q.c > q.d) {
16      return 1;
17     } else {
18         return 0;
19     }
20 }
```

(a) Example code with struct aliasing.

Expression	*Replacement Expression*
`q._`	`p._`

(b) Alias table.

```
(_gtF_a[_gtA_p] > _gtF_b[_gtA_p])
```

(c) GameTime intermediate expressions that correspond to the statements along the highlighted path.

Figure 3.9: **Demonstration of the GameTime struct alias analysis.**

replacement of the placeholder _. This approach allows GameTime to assume that a field in a struct can be referenced using only one expression, affording simpler analysis.

Now, when the operand that represents a field access is traced, as detected on line 9 of Algorithm 4, where the function TraceOperand is defined, the appropriate offset expression is first constructed, as on line 10. For example, the field access expression `q.c` is converted to the offset expression `q.0`. The Normalize function, defined in Algorithm 6, attempts to find a replacement for this expression in the alias table. In this case, it does, and produces the expression `p.0`, where the expression `0` replaces the placeholder _.

The Lookup function, defined in Algorithm 7, uses the offset in the expression (`0`) and the size of the expression (32 bits) to determine the field of the struct that is being accessed. Here, it determines that the field `a` of the struct `p` is being accessed, and the function uses the FieldToArray function, defined in Algorithm 8, to produce the array subscription expression `_gtF_a[_gtA_p]`. The final intermediate expression for the conditional expression on line 15 of Figure 3.9(a) is shown in Figure 3.9(c).

The process of translating a Phoenix operand for a struct field access is summarized in Figure 3.10, using the field access `q.c` in line 15 of Figure 3.9(a) as an example.

However, the struct type that is aliased, and its aliasing struct type, need not necessarily

$$\texttt{t283} \longrightarrow \texttt{q.0} \xrightarrow{\textsc{Normalize}} \texttt{p.0} \xrightarrow{\textsc{Lookup}} \texttt{\_gtF\_a[\_gtA\_p]}$$

Figure 3.10: **Translation of the Phoenix operand for a struct field access.** The field access is `q.c`, used in line 15 of Figure 3.9(a). The Phoenix temporary operand `t283` represents `q.c`.

have the same number, type, or even order of fields. In the code sample presented in Figure 3.11, the four eight-bit `char` fields of the struct variable `p` occupy the same 32 bits of memory as the one 32-bit `int` field of the struct variable `q`. This struct aliasing implies that the bit representation of `q.f` is a concatenation of the bit representations of the four `char` fields of `p`. The order of concatenation depends on the endianness of the machine on which the test cases will be timed: this endianness can be specified as a configuration option for GameTime.

```
 1  typedef struct {
 2      char a;
 3      char b;
 4      char c;
 5      char d;
 6  } my_struct;
 7
 8  typedef struct {
 9      int f;
10  } my_other_struct;
11
12  my_struct p;
13
14  int foo(void) {
15      my_other_struct q = *((my_other_struct *) &p);
16      if (q.f > 0) {
17          return 1;
18      } else {
19          return 0;
20      }
21  }
```

Figure 3.11: **Example of struct aliasing where the structs have fields of different types.** In this example, the variables `p` and `q` refer to the same region of memory, after the assignment statement on line 15. However, `p` has four `char` fields and `q` has one `int` field.

This case is recognized in the function Lookup, as defined in Algorithm 7. The

GameTime offset expression `q.f`, used in line 16 of Figure 3.12, corresponds to the field access expression `q.0`. This expression is normalized to `p.0`. As instructed on line 5 of Algorithm 7, the function Lookup uses the offset (`0`) and the size of the operand (32 bits) to look up the fields between offsets 0 and 32 of the struct type of `p`, or of `my_struct`. The four access expressions for these fields — `p.a`, `p.b`, `p.c` and `p.d` — are converted into array subscription expressions using the function FieldAsArray, as per line 9. Assuming a little-endian machine, the array subscription expressions are concatenated to produce the following GameTime intermediate expression for the field access expression `q.f`:

```
concat(_gtF_d[_gtA_p],
       concat(_gtF_c[_gtA_p],
              concat(_gtF_b[_gtA_p], _gtF_a[_gtA_p])))
```

The nested concatenation accounts for the fact that the `concat` operator is binary.

Another case arises when the definitions are swapped, as in Figure 3.12. In this example, the one 32-bit `int` field of the struct variable `p` occupies the same 32 bits of memory as the four `char` fields of the struct variable `q`. This struct aliasing implies that the bit representation of `q.a` is merely eight bits of the bit representation of `p.f`. If the machine on which the test cases are timed is little-endian, then the bits of `q.a` are the first eight bits, from positions 0 through 7. If the machine is big-endian, however, then the bits of `q.a` are the last eight bits, from positions 24 through 31.

This case is also recognized in the function Lookup. The GameTime offset expression `q.a`, used in line 16 of Figure 3.12, corresponds to the field access expression `q.0`. This expression is normalized to `p.0`. As instructed on line 5 of Algorithm 7, the function Lookup uses the offset (`0`) and the size of the operand (8 bits) to look up the fields between offsets 0 and 8 of the struct type of `p`, or of `my_struct`. One field `a` overlaps this specified region, but the field is not fully contained in this region. The access expression for this field, `p.a`, is converted into an array subscription expression using the function FieldAsArray, as per line 9. The expression is then bit-adjusted, as also specified on line 9: here, the appropriate bits are extracted. Assuming a little-endian machine, the GameTime intermediate expression for the field access expression `q.a` is

```
bitExtract(7, 0, _gtF_a[_gtA_p])
```

A variety of such expressions are constructed for a variety of different cases of struct aliasing that arise in embedded systems code, as long as the field access expressions are converted to the appropriate intermediate offset expression.

(f) *Expressions with unions.* In C, a *union* is an object that contains, at different times, any one of several members of different types [27]. An example is presented in Figure 3.13: a new union type named `my_union` is defined with an `int` field `a` and a `char` field `b`. A

```
1  typedef struct {
2      int f;
3  } my_struct;
4
5  typedef struct {
6      char a;
7      char b;
8      char c;
9      char d;
10 } my_other_struct;
11
12 my_struct p;
13
14 int foo(void) {
15     my_other_struct q = *((my_other_struct *) &p);
16     if (q.a == q.d) {
17         return 1;
18     } else {
19         return 0;
20     }
21 }
```

Figure 3.12: **Another example of struct aliasing where the structs have fields of different types.** In this example, the variables `p` and `q` refer to the same region of memory, after the assignment statement on line 15. However, `p` has one `int` field and `q` has four `char` fields.

global variable `u` is then defined to have this union type. This indicates that the same region of memory (4 bytes) can be accessed either using `u.a` or `u.b` at different times during the program execution. In other words, the C expressions `u.a` and `u.b` are aliases for the same four bytes of memory, although the latter refers to only one byte, while the former expression refers to all four bytes.

This case is treated as a version of struct aliasing. GameTime treats unions as structs that are aliases for the same region of memory, and there are as many structs as there are fields in the union. This implies that one field of the union is the *base* (or *canonical*) field, just as one struct can be an alias for a *base* (or *canonical*) struct.

To demonstrate this, we refer again to the example in Figure 3.13. The field access expression `u.a` is converted to the normalized offset expression `u.0`, whose size is 4 bytes. As a result, the Lookup function returns the array subscription expression `_gtF_a[_gtA_u]`. However, the field access expression `u.b` is also converted to the

```
typedef union {
    int a;
    char b;
} my_union;

my_union u;
```

Figure 3.13: **Example code with a union.**

normalized offset expression `u.0`, whose size is 1 byte. In this case, on a little-endian machine, the LOOKUP function returns the array subscription expression

```
bitExtract(7, 0, _gtF_a[_gtA_u])
```

For this example, the field `a` is the base field for the union type `my_union`.

(g) *Expressions with address-taken variables.* Here, we consider C statements of the form

```
p = &q;
```

where the address of the variable `q` is stored in the pointer variable `p`. In general, `q` can be replaced by any *lvalue*, or any C expression that names a region of storage [27, 37]. However, an equivalent set of statements can be made where the lvalue is assigned to a temporary variable. We can thence assume that `q` is a variable.

Hardekopf and Lin [20] describe an approach that models these statements in the context of the LLVM internal representation. We reiterate this approach in the context of GAMETIME and the Microsoft Phoenix internal representation. In the terminology of [20], the variable `q`, whose address is being stored in `p`, is designated as an *address-taken variable.*

When translating the assignment statement to an intermediate expression, GAMETIME adds `q` to an internally maintained list of address-taken variables. Then, the expression `&q` is replaced by a temporary pointer variable:

```
p = _gtPTR0;
```

Here, the pointer variable `_gtPTR0` has the same type as the pointer variable `p`. The name is arbitrarily assigned and prefixed with `_gt` to avoid name collision with existing pointer variables.

Effectively, _gtPTR0 is now an alias for the pointer &q. This further implies that a dereference of _gtPTR0 will produce the value q. GameTime thus produces the intermediate equivalence expressions

```
 _gtPTR0[0] == q
p<1> == _gtPTR0[0]
```

for the original assignment statement.

To complete the modeling, every occurrence of the variable q in subsequent statements will be replaced with the dereference _gtPTR0[0]. In general, any occurrence of an address-taken variable, after its address has been taken, is replaced by a dereference of the associated temporary pointer variable. An equality expression is generated between the address-taken variable and the dereferenced temporary pointer variable.

(h) *Assume-statements.* GameTime provides *assume-statements* as a mechanism for user-supplied assumptions about program points. For example, in Figure 3.14, line 2 is an assume-statement, which notifies GameTime that the only values that should be assigned to the variable a should be less than 3.

```
1 int foo(int a) {
2     __gt_assume(a < 3);
3     if (a > 0) {
4         return 1;
5     } else {
6         return 0;
7     }
8 }
```

Figure 3.14: **Example of assume-statements.**

Assume-statements can be useful in a few cases: For example, this enables a user to specify any constraints on variables or the arguments to a function, such as the argument variable a in Figure 3.14, without having to explicitly supply a new branch point. Also, this enables the study of data-dependent timing: Though a range of values can drive a program along a certain path, the timing of the path can vary within this range. The assume-statements permit a user to constrain the values that can be assigned in a test case, and therefore to study the timing variations along the same path.

When constructing the intermediate expressions, GameTime detects a call to the C function __gt_assume and traces the operand that represents the argument to the function. The resulting intermediate expression is added to the list of intermediate expressions returned by the function ConstructExpressions.

Step 2: Conversion to SMT query

GameTime uses the `QF_AUFBV` logic [4], defined as part of the SMT-LIB v2.0 standard [5], to construct and solve an SMT query. The `QF_AUFBV` logic works with closed quantifier-free (‘`QF`’) formulas over the theory of fixed-size bitvectors (‘`BV`’) and bitvector arrays (‘`A`’), extended with support for uninterpreted functions (‘`UF`’).

The Microsoft Phoenix API is used to generate a SMT query for each intermediate expression produced. As a result, any solver that supports the `QF_AUFBV` logic can work with GameTime: the current implementation supports Z3 [13], the theorem prover from Microsoft Research, and Boolector [7].

The result of ConstructExpressions is a list of GameTime intermediate expressions that correspond to the statements along a path. The construction detailed in the previous step, and the BNF grammar presented in Figure A.1, indicate that these expressions are built up from fixed-size variable expressions and array subscription expressions. The conversion of the intermediate expressions to the assertions of a SMT query is thus straightforward: Each variable v is represented as a bitvector, whose size is $\textsc{sizeof}(v)$, while the arrays defined in the `QF_AUFBV` logic represent the arrays in the intermediate expressions.

The SMT query that corresponds to the statements along the path highlighted in Figure 3.2(a), through the exemplar modular exponentiation code, is shown in Figure 3.15, presented in the SMT-LIB v2.0 input format [5]. The query starts with the declarations of the variables used in the assertions of the query, followed by the assertions themselves. We note that, along with the variables in the SMT query that represent variables in the intermediate expressions, Boolean constraint variables with the prefix `__gtCONSTRAINT` are also declared. These constraint variables serve a dual purpose: First, these variables are asserted to be equivalent to each converted statement, and the final assertion uses an `and` expression to ensure that all of these constraint variables, and thus the converted statements, are true in a satisfying assignment. Second, and more important, if a satisfying assignment cannot be found, some solvers, such as Z3, return a (not necessarily strict) subset of these constraint variables as the unsatisfiable core. This subset helps GameTime determine which intermediate expressions are the causes for infeasibility. The tool then finds the relevant C statements, the basic blocks that contain them, and thus the constraint edges in the CFG that should not be taken together along any path. This set of edges is added as a path-exclusive constraint in the ILP formulation.

The translation of most GameTime intermediate expressions into the corresponding SMT-LIB expressions is straightforward. Table A in Appendix A maps the operators available in intermediate expressions to their SMT-LIB equivalents, permitting a simple conversion. However, array subscription expressions are treated differently. A sample SMT-LIB declaration of an array `a`, whose elements are 32-bit `int` values, is

```
(declare-fun a () (Array (_ BitVec 32) (_ BitVec 32)))
```

The declaration states that `a` has the set of 32-bit bitvectors as its domain and range: the domain is the set of indices, while the range is the set of possible values in the array.

```
(declare-fun __gtCONSTRAINT0 () Bool)
(declare-fun __gtCONSTRAINT1 () Bool)
(declare-fun __gtCONSTRAINT2 () Bool)
(declare-fun __gtCONSTRAINT3 () Bool)
(declare-fun __gtCONSTRAINT4 () Bool)
(declare-fun __gtCONSTRAINT5 () Bool)
(declare-fun __gtCONSTRAINT6 () Bool)
(declare-fun __gtCONSTRAINT7 () Bool)
(declare-fun __gtCONSTRAINT8 () Bool)
(declare-fun __gtCONSTRAINT9 () Bool)
(declare-fun __gtCONSTRAINT10 () Bool)
(declare-fun result () (_ BitVec 32))
(declare-fun result<1> () (_ BitVec 32))
(declare-fun exponent () (_ BitVec 32))
(declare-fun base () (_ BitVec 32))
(declare-fun p () (_ BitVec 32))
(declare-fun result<2> () (_ BitVec 32))
(declare-fun exponent<1> () (_ BitVec 32))
(declare-fun base<1> () (_ BitVec 32))
(declare-fun exponent<2> () (_ BitVec 32))
(declare-fun base<2> () (_ BitVec 32))
(assert (= __gtCONSTRAINT0 (= result<1> (_ bv1 32))))
(assert (= __gtCONSTRAINT1 (= (bvand exponent (_ bv1 32)) (_ bv1 32))))
(assert (= __gtCONSTRAINT2 (= result<2> (bvsmod base p))))
(assert (= __gtCONSTRAINT3 (= exponent<1> (bvlshr exponent (_ bv1 32)))))
(assert (= __gtCONSTRAINT4 (= base<1> (bvsmod (bvmul base base) p))))
(assert (= __gtCONSTRAINT5 (not (= (bvand exponent<1> (_ bv1 32)) (_ bv1 32)))))
(assert (= __gtCONSTRAINT6 (= exponent<2> (bvlshr exponent<1> (_ bv1 32)))))
(assert (= __gtCONSTRAINT7 (= base<2> (bvsmod (bvmul base<1> base<1>) p))))
(assert (= __gtCONSTRAINT8 (not (= p (_ bv0 32)))))
(assert (= __gtCONSTRAINT9 (not (= p (_ bv0 32)))))
(assert (= __gtCONSTRAINT10 (not (= p (_ bv0 32)))))
(assert (and __gtCONSTRAINT0 __gtCONSTRAINT1 __gtCONSTRAINT2 __gtCONSTRAINT3
             __gtCONSTRAINT4 __gtCONSTRAINT5 __gtCONSTRAINT6 __gtCONSTRAINT7
             __gtCONSTRAINT8 __gtCONSTRAINT9 __gtCONSTRAINT10))
(check-sat)
(exit)
```

Figure 3.15: **Example of a SMT query.** This SMT query, presented in the SMT-LIB v2.0 input format, asserts the conditions that would drive program execution along the path highlighted in Figure 3.2(a).

If an array subscription expression is being assigned a value, as in the example expression `a<1>[0] == 2`, the translated SMT-LIB expression uses the `store` function, defined in the `QF_AUFBV` logic:

```
(= a<1> (store a __gtINDEX0 (_ bv2 32)))
(= __gtINDEX0 (_ bv0 32))
```

The `store` function takes an array, an index, and a value to insert at that index, and returns a new array with a modified value at the given index [11]. This new array is assigned to a new array variable, maintaining the SSA form of the GameTime intermediate expressions.

If an array subscription expression is not assigned a value, as in the example expression `a[0] > 3`, the translated SMT-LIB expression uses the `select` function, also defined in the `QF_AUFBV` logic:

```
(bvsgt (select a __gtINDEX0) (_ bv3 32)))
(= __gtINDEX0 (_ bv0 32))
```

The `select` function takes an array and an index, and returns the value stored in the array at that index [11].

We note that, in the example SMT-LIB expressions shown so far, the index used is itself a variable: these variables are temporary, with the prefix `__gtINDEX`. Once the GameTime intermediate expressions have been constructed, all indices in array subscription expressions are replaced with these temporary variables. Additional expressions are added, which establish equivalences between these temporary index variables and the index expressions they replace. This post-processing is conducted to replace the possibly complex expressions that can be used as array indices, such as in the expression `a[(2 * j) + (3 * k)]`. In such cases, only the value of the index expression is needed, not the expression itself. This replacement thus aids the next step, where the satisfying model returned by a SMT solver is parsed: in a model, assignments are made to numerical indices of an array. The temporary indices prevent the parser from needing to evaluate the possibly complex index expressions.

Another caveat arises during the conversion of array subscription expressions. The `QF_AUFBV` logic only permits arrays whose domain and range are bitvectors. This poses an issue with expressions such as `a[2][1][7]`, where `a[2]` and `a[2][1]` themselves evaluate to arrays: the logic does not allow a direct conversion of these inner subscriptions directly to `select` expressions [4]. The current GameTime approach is to convert such expressions into single-dimensional array subscription expressions where the indices are concatenated. The intermediate expression `a[2][1][7] > 8` is thus converted to the SMT-LIB expressions

```
(bvsgt (select a (concat __gtINDEX0 __gtINDEX1 __gtINDEX2)) (_ bv8 32))
(= __gtINDEX0 (_ bv2 32))
(= __gtINDEX1 (_ bv1 32))
(= __gtINDEX2 (_ bv7 32))
```

The temporary indices replace `2`, `1`, and `7`, respectively. `a` is now declared to be an array variable whose domain is the set of 96-bit bitvectors, each made from the concatenation of three 32-bit bitvectors. Storage into arrays is similar: the intermediate expression `(a<1>[2][1][7] == 8)` is converted to the SMT-LIB expressions

```
(= a<1> (store a (concat __gtINDEX0 __gtINDEX1 __gtINDEX2) (_ bv8 32)))
(= __gtINDEX0 (_ bv2 32))
(= __gtINDEX1 (_ bv1 32))
(= __gtINDEX2 (_ bv7 32))
```

This concatenation of indices is turned on in GameTime by default. However, an option is provided to the user to turn this off. This is useful for SMT solvers such as Z3, which supports an extension [35] to the `QF_AUFBV` logic where the `select` function can also return arrays. Accordingly, the declaration of `a` is

```
(declare-fun a () (Array (_ BitVec 32) (Array (_ BitVec 32) (_ BitVec 32))))
```

The intermediate expression `a[2][1] > 7` is then translated to the SMT-LIB expressions

```
(bvsgt (select (select a __gtINDEX0) __gtINDEX1) (_ bv7 32))
(= __gtINDEX0 (_ bv2 32))
(= __gtINDEX1 (_ bv1 32))
```

The `select` expressions are nested: the innermost `select` expression, for example, produces an array as its output, which is the input to another `select` expression.

Storage into arrays is similar: the intermediate expression `(a<1>[2][1] == 8)` is converted to the SMT-LIB expressions

```
(= a<1> (store a __gtINDEX0
               (store (select a __gtINDEX0) __gtINDEX1 (_ bv8 32))))
(= __gtINDEX0 (_ bv2 32))
(= __gtINDEX1 (_ bv1 32))
```

In the above SMT-LIB expression, the array at index `__gtINDEX0` (or `2`) of array `a` is selected. This array is re-stored into the same index, but only after the index `__gtINDEX1` (or `1`) has been updated with the value to be stored.

Step 3: Model parsing

Once the SMT query is constructed, it is provided as input to an SMT solver. A path is feasible if, and only if, the corresponding SMT query is satisfiable. If the query is satisfiable, the SMT solver outputs a *model*, which supplies values for the variables in the query. The query presented in Figure 3.15 is satisfiable: Figure 3.17(a) shows an example of a model produced by Z3, and Figure 3.18(a) shows an example produced by Boolector.

These models are parsed to construct the test cases that will be measured. These test cases contain the assignments present in the model, but in the format of C statements. Assignments to temporary variables, such as the Boolean constraint variables and the different "versions" of a variable, are ignored. Figure 3.17(b) shows the test case that corresponds to the model produced by Z3, and Figure 3.18(b) shows the test case for the model produced by Boolector. We observe that different solvers can produce different values to drive program execution along the same path; for example, the value for the variable `base` is different. In both test cases though, the variable `exponent` is assigned `1`, as expected.

3.5 Edge weight and path length estimation

After the basis paths have been generated and the corresponding test cases obtained, we use the test cases to measure the lengths of these paths on the environment under analysis. For our experiments, we used deterministic processor simulators from a fixed initial hardware state as the environments. We estimate the weights on the edges of the graph using the GameTime algorithm presented in [46, 47], which is modified to account for the different path representation used by the implementation of GameTime.

More formally, we define the estimated weight vector $\tilde{w} = (\tilde{w}_1, \tilde{w}_{P_1}, \tilde{w}_{P_2}, \ldots, \tilde{w}_{P_{|B|}})$, where $\tilde{w}_1$ is the weight of the sole edge (edge 1) from the source node, and where $\tilde{w}_{P_i}$ is the weight of the default edge of branch point P_i. Let v_i be the length (or timing) of the ith basis path. We define the vector $v \in \mathbb{R}^{(|B|+1)}$ as the vector $(v_1, v_2, \ldots, v_b, 0, \ldots, 0)$. The extra entries of 0 account for the rows of the basis matrix that are “bad” rows, or rows that do not correspond to basis paths.

According to the GameTime algorithm, we can compute the estimated weights with the equation $B\tilde{w} = v$, or $\tilde{w} = B^{+}v$, where B^{+} is the Moore-Penrose pseudo-inverse of B, defined as $B^{+} = B^{\mathsf{T}}(BB^{\mathsf{T}})^{-1}$. Since Algorithm 1 ensures that the determinant of B is never zero, B is invertible, and hence B^{+} is also B^{-1}, the inverse of the matrix B.

Intuitively, the equation $B\tilde{w} = v$ allows GameTime to estimate weights on the default edges of the source node and the branch points of the CFG. Since a path is represented using these edges, GameTime is now able to predict the length ℓ_p of any path p in the graph with the equation $\ell_p = \tilde{w}p^{\mathsf{T}}$.

The invertibility of B ensures that the equation $\tilde{w} = B^{+}v$ will always have a solution. In the context of timing analysis, the weights estimated on the default edges do *not* necessarily represent the timings of the corresponding basic blocks. Intuitively, since all program executions traverse a path in the CFG that starts from the source and ends at the sink, the timings of each individual basic block along a path does not matter, but the sum of the timings yields the timing (or length) of the path.

The additional entries of 0 added for the “bad” rows (if any) of the basis matrix indicate that these rows do not represent valid basis paths, and their lengths cannot thus be obtained through measurement. If the spanner set were to include the paths represented by these rows, any feasible path expressed as a linear combination of elements in this larger set can have zeros as the coefficients associated with these paths. As a result, the lengths of these paths can be arbitrarily chosen, and have been chosen to be zero for the current implementation. The need for these additional entries can be elided by removing these rows from B, rendering the basis matrix as a $b \times (|B|+1)$ matrix. However, the matrix would not always be invertible, although the Moore-Penrose pseudoinverse B^{+} would always exist.

Longest (or shortest) path prediction

Once the weights have been estimated, the longest (or shortest) feasible path can be predicted, and its length estimated. This problem can again be cast as an integer linear program.

This integer linear program is similar to the program presented in Section 3.3. The two problems have the same constraints, which ensure that a valuation of the variables e_i translates to valid, though not necessarily feasible, paths. The sole difference is in the objective function used. Let x be the longest path in the CFG, defined as $x = (1, e_{P_1}, e_{P_2}, \ldots, e_{P_{|B|}})$, where e_{P_i} is the variable that represents the flow through the default edge for branch point P_i. As discussed earlier in this section, its length is given by $\tilde{w}x^{\mathsf{T}}$, which results in an expression that is linear in (a subset of) the variables e_i. This expression is the objective function for the integer linear program. Since x is the longest path, we choose to maximize this function. The solution of this integer linear program will be used to construct the path x.

As in Algorithm 1, we determine the feasibility of this path using an SMT solver. If the path is infeasible, it is excluded, and the new integer linear program is solved. If the path is feasible, however, GameTime returns the path as the predicted longest feasible path in the CFG, and the quantity $\tilde{w}x^{\mathsf{T}}$ is its estimated length.

The current implementation of GameTime supports the generation of the k longest (or shortest) feasible paths in the CFG. For example, the predicted second longest feasible path in the CFG can be obtained by excluding the predicted longest feasible path, and then solving the resulting integer linear program, progressively excluding infeasible paths until a feasible path is obtained. Shorter and shorter paths can be obtained by progressively excluding the longest paths, and solving the resulting integer linear programs. The predicted shortest feasible path in the CFG can be obtained by *minimizing* the objective function, instead of maximizing.

```
   {*StaticTag}, {*NotAliasedTag}, {*UndefinedTag}<1> = START _modexp_simple(T)
_modexp_simple: (references=1)
   _base<2>, _exponent<3> = ENTERFUNCTION
                       ENTERBLOCK ScopeSymbol276
                       ENTERBLOCK ScopeSymbol278
                       ENTERBLOCK ScopeSymbol281
                       ENTERBLOCK
   _result<4>        = ASSIGN 1
   t282              = BITAND _exponent<3>, 1
   t283              = COMPARE(EQ) t282, 1
                       CONDITIONALBRANCH(True) t283, $L7, $L6
$L7: (references=1)
                       ENTERBLOCK
   t285              = REMAINDER* _base<2>, _p
   _result<5>        = ASSIGN t285
                       EXITBLOCK
                       GOTO $L6
$L6: (references=2)
   _result<11>       = PHI _result<4>, _result<5>
   t286              = ASSIGN _exponent<3>
   _exponent<6>      = SHIFTRIGHT t286, 1
   t287              = MULTIPLY _base<2>, _base<2>
   t288              = REMAINDER* t287, _p
   _base<7>          = ASSIGN t288
   t289              = BITAND _exponent<6>, 1
   t290              = COMPARE(EQ) t289, 1
                       CONDITIONALBRANCH(True) t290, $L9, $L8
$L9: (references=1)
                       ENTERBLOCK
   t291              = MULTIPLY _result<11>, _base<7>
   t292              = REMAINDER* t291, _p
   _result<8>        = ASSIGN t292
                       EXITBLOCK
                       GOTO $L8
$L8: (references=2)
   _result<12>       = PHI _result<11>, _result<8>
   t293              = ASSIGN _exponent<6>
   _exponent<9>      = SHIFTRIGHT t293, 1
   t294              = MULTIPLY _base<7>, _base<7>
   t295              = REMAINDER* t294, _p
   _base<10>         = ASSIGN t295
                       RETURN _result<12>, $L12(T)
                       EXITBLOCK
                       EXITBLOCK ScopeSymbol281
                       EXITBLOCK ScopeSymbol278
                       EXITBLOCK ScopeSymbol276
$L11: (references=0)
                       GOTO {*StaticTag}, $L5
$L5: (references=1)
                       UNWIND
$L12: (references=1)
                       GOTO {*StaticTag}, $L3
$L3: (references=1)
                       EXITFUNCTION
$L2: (references=0)
                       END
```

Figure 3.16: **Example of the Microsoft Phoenix internal representation.** This is the internal representation of the (unrolled) modular exponentiation code given in Figure 2.2(b).

```
(define-fun result<2> () (_ BitVec 32)
  #x00000000)
(define-fun p () (_ BitVec 32)
  #x00000001)
(define-fun base<1> () (_ BitVec 32)
  #x00000000)
(define-fun __gtCONSTRAINT1 () Bool
  true)
(define-fun __gtCONSTRAINT2 () Bool
  true)
(define-fun base<2> () (_ BitVec 32)
  #x00000000)
(define-fun __gtCONSTRAINT0 () Bool
  true)
(define-fun __gtCONSTRAINT8 () Bool
  true)
(define-fun __gtCONSTRAINT3 () Bool
  true)
(define-fun __gtCONSTRAINT9 () Bool
  true)
(define-fun exponent () (_ BitVec 32)
  #x00000001)
(define-fun result<1> () (_ BitVec 32)
  #x00000001)
(define-fun __gtCONSTRAINT4 () Bool
  true)
(define-fun __gtCONSTRAINT7 () Bool
  true)
(define-fun __gtCONSTRAINT6 () Bool
  true)
(define-fun exponent<2> () (_ BitVec 32)
  #x00000000)
(define-fun __gtCONSTRAINT10 () Bool
  true)
(define-fun __gtCONSTRAINT5 () Bool
  true)
(define-fun exponent<1> () (_ BitVec 32)
  #x00000000)
(define-fun base () (_ BitVec 32)
  #x02007db7)
```

(a) Model produced by Z3

```
base = 0x2007db7;
exponent = 0x1;
p = 0x1;
```

(b) Corresponding test case

Figure 3.17: **Example of a Z3 model and the corresponding test case.** The model is produced by the SMT solver Z3, v4.3.0, in response to the SMT query presented in Figure 3.15. The values in the model and in the test case are supplied in hexadecimal notation.

```
__gtCONSTRAINT0 1
__gtCONSTRAINT1 1
__gtCONSTRAINT2 1
__gtCONSTRAINT3 1
__gtCONSTRAINT4 1
__gtCONSTRAINT5 1
__gtCONSTRAINT6 1
__gtCONSTRAINT7 1
__gtCONSTRAINT8 1
__gtCONSTRAINT9 1
__gtCONSTRAINT10 1
result xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
result<1> 00000000000000000000000000000001
exponent 00000000000000000000000000000001
base 10101101100101101110110100000101
p 11111111111111111111001000010100
result<2> 11111111111111111111001101001001
exponent<1> 00000000000000000000000000000000
base<1> 11111111111111111111010001101001
exponent<2> 00000000000000000000000000000000
base<2> 11111111111111111111001000011101
```

(a) Model produced by Boolector

```
base = 0xad96ed05;
exponent = 0x1;
p = 0xfffff214;
```

(b) Corresponding test case

Figure 3.18: **Example of a Boolector model and the corresponding test case.** The model is produced by the SMT solver Boolector, v1.5.118, in response to the SMT query presented in Figure 3.15. The values in the model are supplied in binary notation, while those in the test case are supplied in hexadecimal notation.

Chapter 4

Experimental Results

The goal of the first set of experiments presented in this chapter is to demonstrate that, by measuring only a small subset of the paths in the control-flow graph of a function under analysis, GameTime can predict the execution time of any arbitrary program path, in a collection of environments and fixed starting states. Another experiment aims to demonstrate that GameTime can also predict the energy consumption of any arbitrary program path.

4.1 Benchmarks

The benchmark suite for these sets of experiments comprises either loop-free programs, or programs where the loops have statically-known finite bounds and function calls have known finite recursion depths. In the latter case, the loops are unrolled and the function calls are inlined. Both the GameTime toolkit and the benchmark suite are located at `http://uclid.eecs.berkeley.edu/gametime`.

Benchmark	*Lines of Code*	*CFG Characteristics*			*Basis Paths, b*
		Nodes, n	Edges, m	Total Number of Paths	
`modexp`	44	28	31	16	5
`altitude`	223	36	40	11	6
`bs`	146	70	80	120	9
`toyota-1`	812	232	266	744	17
`toyota-2`	157	146	166	648	20
`irobot`	160	234	268	3521	36
`cnt`	1949	612	711	1.2×10^{30}	101
`statemate`	990	1088	1268	7.1×10^{16}	157

Table 4.1: **Characteristics of benchmarks.**

Some of the characteristics of these benchmarks are shown in Table 4.1. For each benchmark, the second column shows the number of lines of C code in the function under analysis.

The next three columns provide numerical characteristics of the control-flow graph (CFG) of the function. The total number of paths is calculated using a dynamic programming algorithm that, however, does not exclude infeasible paths.

The last column shows the number of basis paths, b, in the CFG, as discovered by GameTime. We note that the total number of paths in the CFG grows exponentially in the size of the graph, reaching a size of 1.2×10^{30} for the `cnt` benchmark. However, the number of basis paths grows polynomially in the size of the graph: this arises from the fact that the number of basis paths has an upper bound of $m - n + 2$. The benchmarks in Table 4.1 are arranged in ascending order of the number of basis paths identified by GameTime.

The benchmarks include a variety of programs, ranging from small toy programs to several real programs run on embedded systems. An example of the former is the `modexp` benchmark is a variant of the modular exponentiation code, presented in Figure 2.2(b), but modified to use 4-bit exponents instead of 2, with a base of 2 and a prime modulus of 1048583.

The `altitude` benchmark is a task in the open-source PapaBench software for an unmanned aerial vehicle [38]. The two `toyota` benchmarks are functions obtained from the software used in the engine control systems of Toyota vehicles.

The `irobot` benchmark is used to control an iRobot Create autonomous robot platform [12]. The benchmark is the body of an infinite sense-compute-actuate loop that encodes a state machine, which maintains the state of an iRobot Create as it attempts to move forward until it senses an obstacle, in which case it will try to move around the obstacle.

The other three benchmarks are supplied as part of the Mälardalen benchmark suite [19]. The `bs` benchmark performs a binary search for an integer in an array of 15 integer elements. The `cnt` benchmark counts the number of non-negative numbers in a 10×10 matrix. Finally, the `statemate` benchmark is code that has been generated from a Statemate [21] Statecharts model for an automotive window control system.

4.2 Execution Time: Experiments and Results

The execution time analysis experiments were conducted in three different environments: the PTARM simulator, the SimIt-ARM simulator, and the RTE-2000H-TP In-Circuit Emulator. These environments, the experiments and the results are described below.

PTARM simulator

Simulator Description and Experimental Setup

The PTARM simulator v1.0 is a simulator of a Precision Timed (PRET) machine [24] developed by Liu et al. [32] that is based on a subset of the ARMv4 instruction set architecture. The simulator was designed as part of an effort to introduce predictable and repeatable timing to real-time embedded processor architectures [23]. This particular simulator was chosen because of its cycle accuracy, its ease of availability and its timing repeatability, the third of which helps to ensure that there are no spurious variations in the execution time measurements of test cases. The experiments were run in the simulator through its command-line interface, in a Cygwin [43] terminal on a 64-bit Windows machine. The GNU ARM toolchain [15], recommended by the simulator documentation [23], was used to compile the test cases.

Experiment Description and Results

Table 4.2 summarizes the measurements conducted for different benchmarks on the PTARM simulator v1.0. It is divided into six sub-tables, one for each combination of two SMT solvers and three integer linear programming (ILP) solvers. The two SMT solvers used were Z3 [13], the theorem prover from Microsoft Research, and Boolector [7]. The three ILP solvers used were the GNU Linear Programming Kit (GLPK) [41], the Gurobi Optimizer [40], and the default solver that accompanies the PuLP Python package [52], which is a version of CBC (Coin Branch and Cut) [9].

For each benchmark and combination of SMT solvers and ILP solvers presented in Table 4.2, the runtimes (or execution times) of the basis paths were measured on the PTARM simulator, from the same initial state of the simulator. Then, the runtimes of arbitrarily selected paths were predicted by GameTime, using the runtimes collected for the basis paths. The runtimes for these paths were then measured on the simulator, and the predictions were compared against the measurements.

In the table, the second column shows the number of arbitrarily selected paths whose runtimes were predicted and measured. The next two columns show the maximum and minimum measured runtimes of any basis path, while the next two columns show the maximum and minimum runtimes of *all* the paths that have been measured. The maximum absolute and relative errors of the predicted runtime, with respect to the measured runtime, are displayed in the next two columns: note that these two errors need not necessarily be for the same path. The last-but-one column displays the amount of time taken by GameTime to perform its analysis of a benchmark and to generate its basis paths.

Finally, the last column has an entry of ‘Yes’ if the path that is measured to have the maximum execution time, among all of the paths measured, is also the path that is predicted to have the maximum execution time. Otherwise, a numerical entry denotes the relative position of the path among all of the paths measured: for example, an entry of ‘7’ indicates that the path that is measured to have the maximum execution time is actually predicted to have an execution time that is the seventh longest.

We observe that, for the PTARM simulator, predictions and measurements match with a very small relative error: the maximum relative error (3.74%) arises for the experiment that uses the `altitude` benchmark, with Z3 as the SMT solver and Gurobi as the ILP solver. The `cnt` benchmark exhibits the least variability in its very low error rate: this could be attributed to its neat branch structure, which consists of a repeated `if`-statement that tests if a number is non-negative before incrementing a counter. We note that the timings of the basis paths are of the same order of magnitude as those of the arbitrarily chosen paths, which was one of the desired features of a basis, as presented in Chapter 2.

The variability in predictions across the combinations of SMT solvers and ILP solvers seems to arise for two reasons: First, different ILP solvers provide different solutions to the same integer linear program, especially in the first few iterations of producing candidate basis paths, where the integer linear programs have multiple possible solutions. This is borne out by the various bases produced by different ILP solvers. Second, the runtime of a path can depend on the values used to drive program execution along the path, especially if the path involves operations such as multiplication. Since these experiments were conducted with no constraints on the variables besides those enforced by the statements along a path, *data-dependent* variability was not explicitly considered and could manifest itself in the tolerance (ε, described in Chapter 2) and error. Further study of such data dependency is needed.

We also observe that in most cases, GameTime is able to predict the path with the longest runtime. The anomalously high rankings in the last column for the `bs` benchmark can be explained by the fact that, when the paths are arranged in descending order of runtimes, the first 24 paths are within a mere two cycles of each other. In the other cases, data-dependent runtime variation could account for the inability of GameTime to predict the path with the longest runtime, since a change in the combination of SMT solvers and ILP solvers, and hence the resulting test cases, changes the entries in the last column.

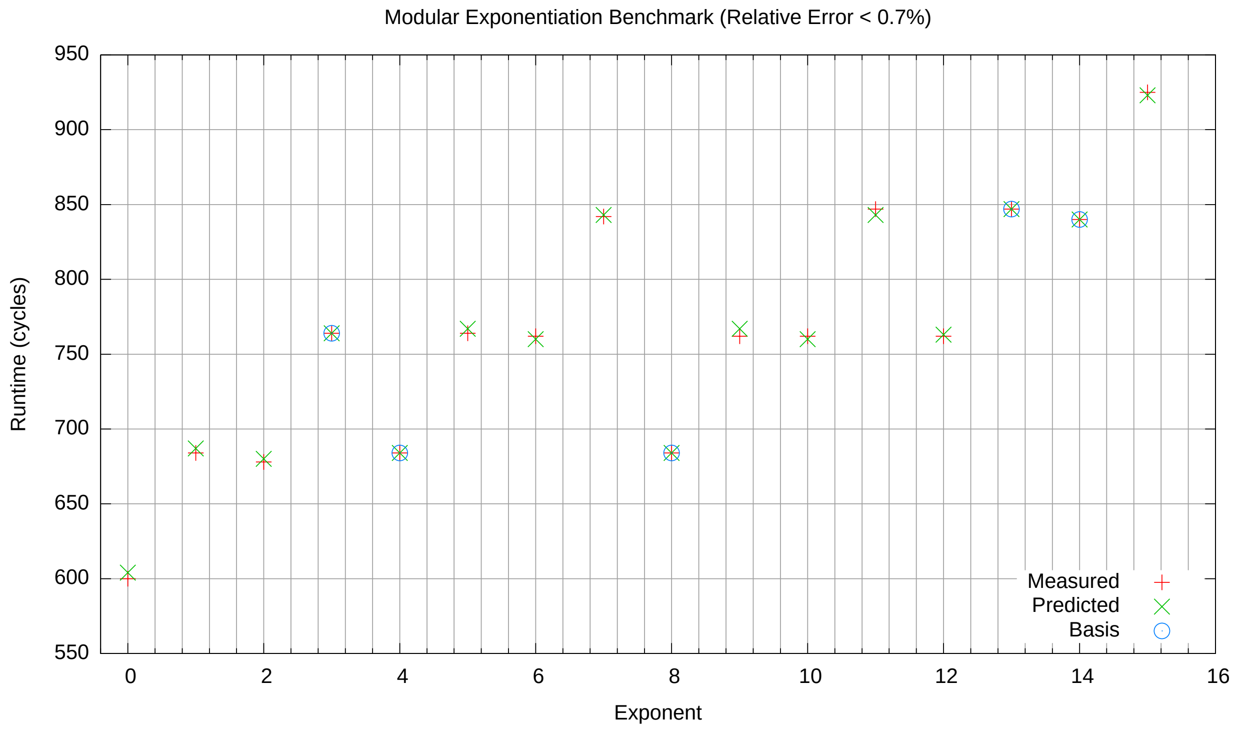


Figure 4.1: **Runtimes of the `modexp` benchmark on the PTARM simulator.** Z3 and GLPK were, respectively, the SMT solver and the ILP solver used for this analysis.

The graphs in Figures 4.1 and 4.2 present alternative views of these results. Figure 4.1

displays the runtimes, both predicted and measured, of the `modexp` benchmark for different values of the exponent. These measurements were conducted on the PTARM simulator, and the analysis was performed with Z3 as the SMT solver and GLPK as the ILP solver. The five basis paths have also been labeled.

As corroborated by Table 4.2, the error margin is very small, with the GameTime predictions within 5 cycles of the measurements. Since the graph depicts all of the possible values for the exponent, we can also conclude that GameTime accurately predicts the path with the worst-case execution time (WCET) and the WCET itself.

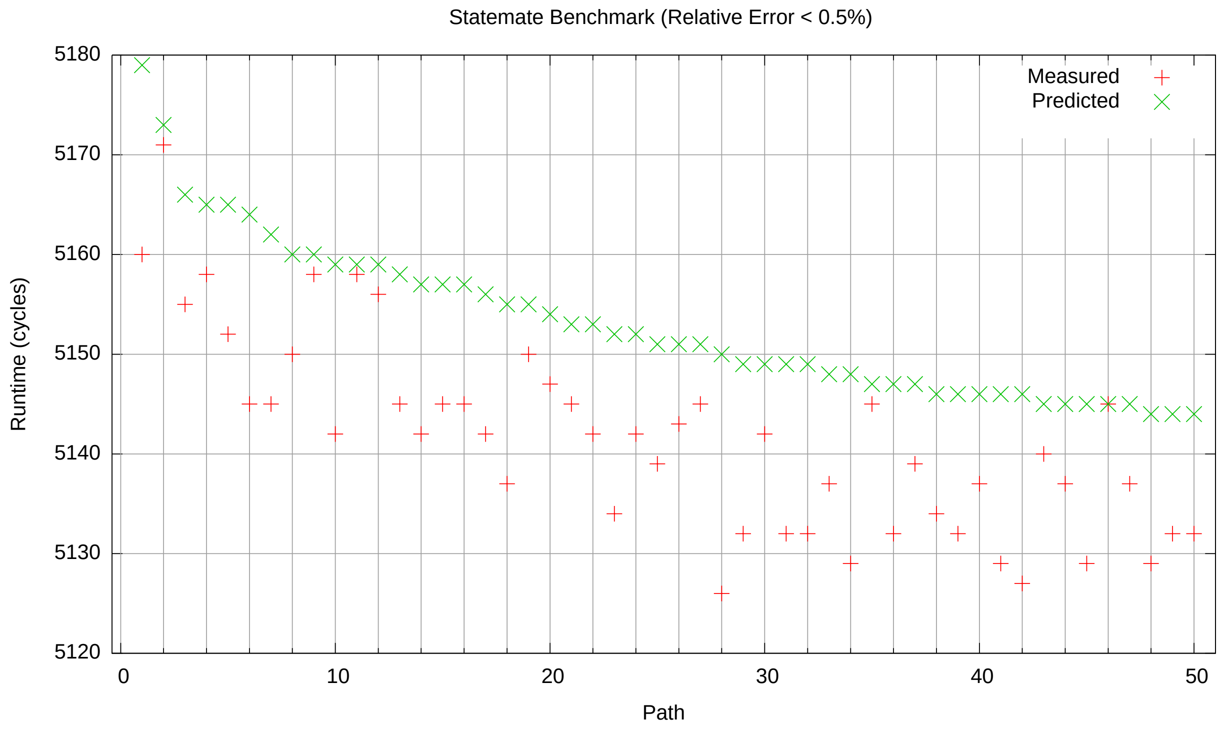


Figure 4.2: **Runtimes of the `statemate` benchmark on the PTARM simulator.** Z3 and GLPK were, respectively, the SMT solver and the ILP solver used for this analysis.

Figure 4.2 displays the runtimes, both predicted and measured, of the `statemate` benchmark, for the fifty predicted longest feasible paths in the CFG of the benchmark. These measurements were also made on the PTARM simulator, and the analysis was performed with Z3 as the SMT solver and GLPK as the ILP solver. None of the basis paths have runtimes in the range presented in Figure 4.2 and have thus not been labeled. As corroborated by Table 4.2, the error margin is still very small, with the GameTime predictions within 0.5% of the measurements. GameTime is thus able to predict runtimes using fewer than 200 basis paths, without needing to exhaustively enumerate the 7.1×10^{16} possible paths. Also, while GameTime does not accurately estimate the path with the worst-case execution time, the measured runtime of the estimated longest path is within 0.2% of the measured runtime of the actual longest path. Once again, data dependency can be investigated as a possible reason for the misprediction.

SimIt-ARM simulator

Simulator Description and Experimental Setup

SimIt-ARM is a cycle-accurate simulator for the StrongARM-1100 processor. It is based on the Operation State Machine formalism presented in [42]. As with the PTARM simulator, the simulator was chosen for its ease of availability, its cycle accuracy and its ability to simulate an entire system, including the memory management unit [44]. The simulator was installed on a 64-bit machine running the Ubuntu operating system. The experiments were conducted in the simulator through its command-line interface. The programs were compiled using the `arm-linux-gcc` compiler, as recommended by the simulator documentation [44].

Experiment Description and Results

A subset of the experiments performed on the PTARM simulator were also conducted on the SimIt-ARM simulator, with the results summarized in Table 4.4. We observe that, for the SimIt-ARM simulator, the `modexp` benchmark exhibits a small relative error across all available combinations of SMT solvers and ILP solvers. The maximum relative errors for the `altitude` and `bs` benchmarks, however, vary significantly across the combinations. As described before, this could be due to the data dependency of timing: the `altitude` and `bs` benchmarks have a relatively small amount of paths, yet there is a large range of values that can drive a benchmark along a particular path. A study of this data dependency can drive future work on the GameTime toolkit.

RTE-2000H-TP In-Circuit Emulator

Emulator Description and Experimental Setup

Experiments analogous to those conducted on the PTARM simulator and SimIt-ARM simulator were also performed on the RTE-2000H-TP In-Circuit Emulator (ICE). This ICE, created by the Midas Lab [29], was the emulator employed by Toyota, for its own research purposes, when these experiments were performed. The ICE was used to communicate with industrial control software.

The two benchmarks used for the experiments were obtained from control software. The test cases for each benchmark were supplied to the ICE through a custom front-end on a Windows workstation. The front-end was capable of printing out timing information and inserting breakpoints, both useful for debugging the engine control software. Breakpoints were placed before and after each benchmark: When the first breakpoint was reached, the test case was inserted, the system time was recorded, and execution was resumed; when the second breakpoint was reached, the time was once again recorded, and the difference was designated as the execution time of the benchmark.

Experiment Description and Results

Table 4.2 aggregates the results of the runtime experiments conducted on the RTE-2000H-TP ICE. GameTime used Z3 as the SMT solver and GLPK as the ILP solver to generate the basis paths and the corresponding test cases.

Benchmark	*Number of Predictions*	*Basis Path Times*		*Measured Times*		*Prediction Error*		*Analysis Time (s)*	*Predicted Longest*
		Max	Min	Max	Min	Abs	Rel		
`toyota-1`	27	78	77	81	80	2	2.5%	< 120	Yes
`toyota-2`	30	97	96	98	97	2	2.04%	< 120	Yes

Table 4.2: **Measurements performed on RTE-2000H-TP In-Circuit Emulator.** The runtimes of the paths, as shown in columns 3 through 6, are presented in nanoseconds, as is the absolute prediction error. The last-but-one column shows the time for the GameTime analysis in seconds.

We observe that the timings of the basis paths differ from each other by at most one nanosecond. Nonetheless, GameTime is able to predict the runtimes of other arbitrary paths with a small relative error of less than 2.5%. Also, for both benchmarks, GameTime accurately predicted the path with the WCET and the WCET itself.

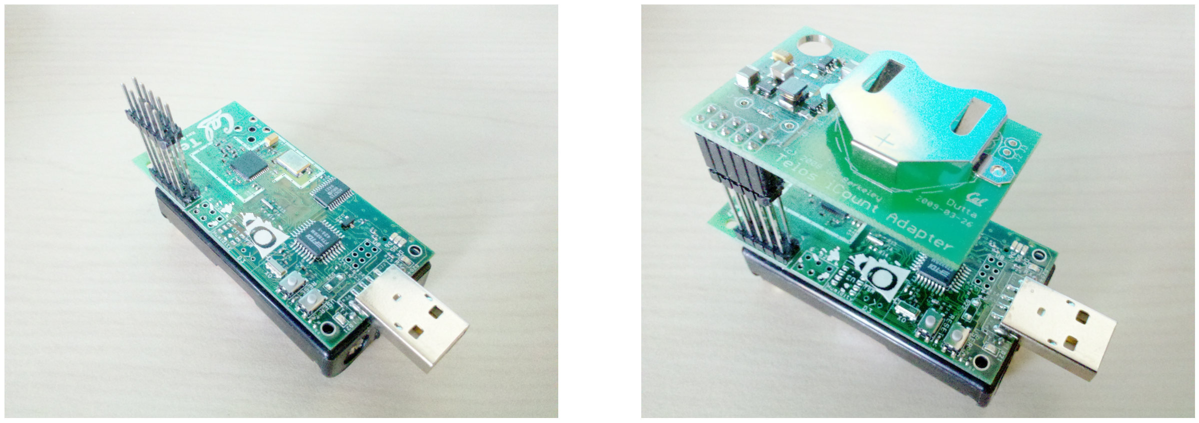
4.3 Energy Consumption: Experiments and Results

In work conducted in collaboration with Dorsa Sadigh, the GameTime toolkit was also employed to estimate the energy consumption of a program, in an attempt to extend the theory and utility of GameTime to other quantitative properties besides execution time.

Physical Apparatus

The energy consumption measurements were made on a TelosB mote, as shown in Figure 4.3(a). The mote was developed at University of California, Berkeley, and has a TI MSP430 microcontroller with 10kB of RAM. Data is collected from the mote via its USB interface, and programs are also sent to the mote through this interface. The operating system, TinyOS [48], is open-source, and programs written for the TinyOS are written in a dialect of C called nesC [16]. The MSP430 microcontroller, developed by Texas Instruments, is a RISC processor with a 16-bit architecture, and has no data or instruction cache [26].

The mote already allows us to determine the runtime of a program: we merely reset a timer and start it just before running a program, polling the value of the timer when the program is completed. However, to measure the energy usage of a program, we attach an iCount adapter, as shown in Figure 4.3(b). The design of the iCount adapter is explained in [14]: Its energy measurements have an error rate of ±20%.



(a) TelosB mote. (b) TelosB mote with the iCount Adapter.

Figure 4.3: **TelosB mote used for energy consumption measurements.**

Experiment Description and Results

Only one benchmark was used for this experiment: a variant of the modular exponentiation code, presented in Figure 2.2(b), which has been modified to use 5-bit exponents instead of 2, with a prime modulus of 1048583. In this experiment, the basis paths of the benchmark were first discovered by GameTime. The energy consumptions of these basis paths on the TelosB mote were then measured: to obtain one measurement, an interface to the TelosB mote, with the iCount adapter attached, was polled to obtain the energy consumed by 10000 iterations of each basis path, and thus the average energy consumed by each basis path. GameTime then used these measurements to predict the average energy consumptions of all 32 possible paths through the benchmark code. These predictions were then compared against the energy consumption values measured on the mote, averaged across 10000 iterations.

Figure 4.4 shows the energy consumption values for different values of the exponent, when 2 is used as the base for the benchmark. The six basis paths have also been labeled. The error margin between the measured and predicted values is very small, with a maximum relative error of less than 1%. Since all the possible values for the exponent have been measured, we also observe that, as with execution time, GameTime can also predict the path with the worst-case energy consumption and the energy consumption itself.

Figure 4.5 shows the energy consumption values for different values of the exponent, when 15 is used as the base for the benchmark instead of 2. Again, the error margin between the measured and predicted values is very small, with a maximum relative error of less than 5%. GameTime also predicts the path with the worst-case energy consumption and the energy consumption itself. This experiment provides credence to the possible usage of GameTime as an energy estimation tool, although further experiments need to be conducted.

Benchmark	*Number of Predictions*	*Basis Path Times*		*Measured Times*		*Prediction Error*		*Analysis Time (s)*	*Predicted Longest*
		Max	Min	Max	Min	Abs	Rel		
Z3 (SMT) and GLPK (ILP)									
`modexp`	16	847	684	925	600	5	0.67%	5.43	Yes
`altitude`	9	806	107	806	107	3	0.38%	6.15	Yes
`bs`	31	738	270	740	270	6	0.82%	27.34	7
`irobot`	82	543	146	543	146	6	1.27%	71.57	Yes
`cnt`	201	15954	15954	15954	15954	1	0.01%	879.73	Yes
`statemate`	257	4644	3175	5160	3032	24	0.47%	2311.02	2
Z3 (SMT) and Gurobi (ILP)									
`modexp`	16	847	600	925	600	6	0.88%	5.36	Yes
`altitude`	9	806	109	806	107	16	3.74%	5.55	Yes
`bs`	31	739	270	740	270	7	0.95%	25.66	7
`irobot`	56	543	146	543	466	5	1.06%	56.27	Yes
`cnt`	201	15954	15954	15954	15954	1	0.01%	920.11	Yes
Z3 (SMT) and PuLP Default (ILP)									
`modexp`	16	847	600	925	600	9	1.06%	5.36	Yes
`altitude`	9	804	109	804	107	2	0.94%	5.55	Yes
`bs`	31	739	270	740	270	4	0.68%	32.30	11
`irobot`	57	517	146	543	465	6	1.27%	59.69	Yes
`cnt`	201	15954	15954	15954	15954	1	0.01%	949.56	Yes
Boolector (SMT) and GLPK (ILP)									
`modexp`	16	847	684	925	600	5	0.67%	5.82	Yes
`altitude`	9	809	107	809	107	6	0.76%	7.73	Yes
`bs`	31	741	270	741	270	5	0.68%	101.79	2
`cnt`	201	15873	15382	15951	15304	14	0.09%	1453.71	2
Boolector (SMT) and Gurobi (ILP)									
`modexp`	16	847	600	925	600	6	0.88%	5.91	Yes
`altitude`	9	809	109	809	107	6	0.76%	7.60	Yes
`bs`	31	738	270	741	270	3	0.41%	90.39	3
`cnt`	201	15766	15483	15951	15304	35	0.23%	989.17	Yes
Boolector (SMT) and PuLP Default (ILP)									
`modexp`	16	842	678	925	600	9	1.06%	5.85	Yes
`altitude`	9	809	107	809	109	6	0.76%	16.34	Yes
`bs`	31	741	270	740	270	5	0.68%	121.79	3
`cnt`	201	15607	15590	15951	15307	13	0.08%	< 900	Yes

Table 4.3: **Measurements performed on PTARM Simulator v1.0.** The runtimes of the paths, as shown in columns 3 through 6, are presented in number of cycles, as is the absolute prediction error. The last-but-one column shows the time for the GameTime analysis in seconds.

Benchmark	*Number of Predictions*	*Basis Path Times*		*Measured Times*		*Prediction Error*		*Analysis Time (s)*	*Predicted Longest*
		Max	Min	Max	Min	Abs	Rel		
Z3 (SMT) and GLPK (ILP)									
`modexp`	16	975	861	1032	805	0	0%	8.74	Yes
`altitude`	9	1489	264	1489	264	277	22.97%	6.86	Yes
`bs`	31	1545	772	1576	771	611	39.55%	29.08	5
Z3 (SMT) and Gurobi (ILP)									
`modexp`	16	976	805	1032	793	12	1.51%	6.54	Yes
`altitude`	9	1489	264	1489	264	139	29.26%	6.30	Yes
`bs`	31	1436	772	1576	771	356	23.88%	25.68	2
Z3 (SMT) and PuLP Default (ILP)									
`modexp`	16	975	861	1032	793	12	1.41%	6.17	Yes
`altitude`	9	1489	264	1491	264	188	19.37%	7.47	Yes
`bs`	31	1545	772	1576	771	358	26.55%	9.74	6
Boolector (SMT) and GLPK (ILP)									
`modexp`	16	975	861	1032	805	0	0%	6.45	Yes
`altitude`	9	1396	264	1396	264	6	0.52%	11.14	Yes
`bs`	31	1484	772	1576	759	327	28.26%	99.88	7
Boolector (SMT) and Gurobi (ILP)									
`modexp`	16	976	805	1032	805	1	0.11%	6.85	Yes
`altitude`	9	1396	264	1396	264	6	1.26%	7.13	Yes
`bs`	31	1368	772	1564	759	334	21.82%	108.11	16
Boolector (SMT) and PuLP Default (ILP)									
`modexp`	16	975	861	1032	805	1	0.12%	10.14	Yes
`altitude`	9	1396	264	1396	264	6	1.26%	10.31	Yes
`bs`	31	1484	772	1576	771	335	28.96%	38.31	Yes

Table 4.4: **Measurements performed on SimIt-ARM Simulator v3.0.** The runtimes of the paths, as shown in columns 3 through 6, are presented in number of cycles, as is the absolute prediction error. The last-but-one column shows the time for the GameTime analysis in seconds.

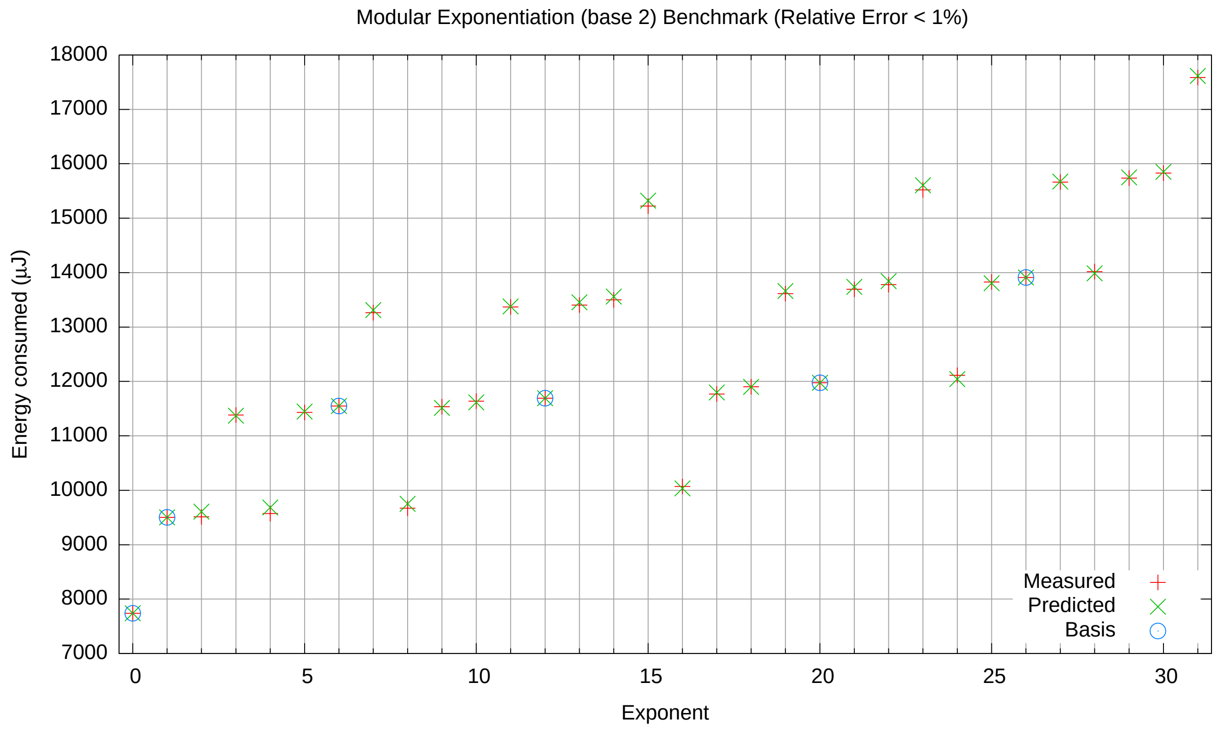


Figure 4.4: **Energy consumed by the `modexp` benchmark (base 2) on the TelosB mote.** The energy consumption values are averaged across 10000 iterations. Z3 and GLPK were, respectively, the SMT solver and the ILP solver used for this analysis.

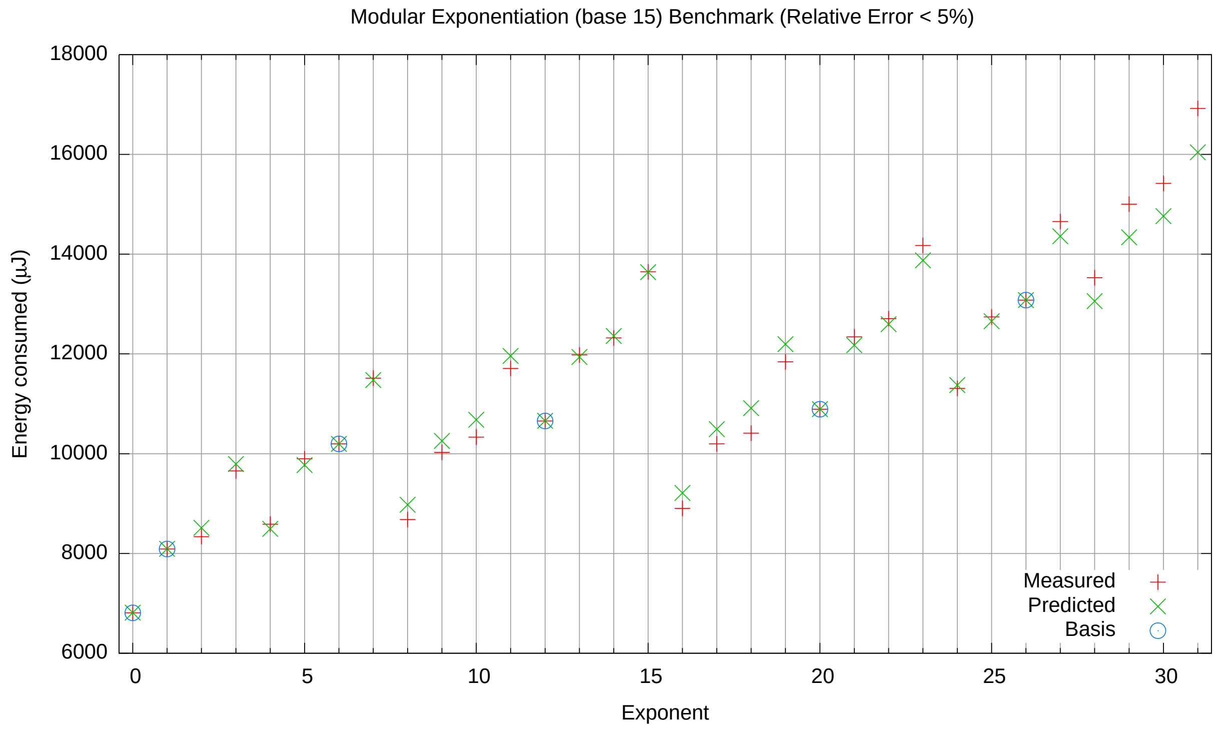


Figure 4.5: **Energy consumed by the `modexp` benchmark (base 15) on the TelosB mote.** The energy consumption values are averaged across 10000 iterations. Z3 and GLPK were, respectively, the SMT solver and the ILP solver used for this analysis.

Chapter 5

Conclusions and Future Work

In this thesis, the different parts of the GameTime timing analysis toolkit are described. In particular, the challenges that arise when encoding different constructs in embedded C programs as clauses in SMT queries are presented, and the algorithms and heuristics that are used to address these challenges are detailed.

The results of experiments that involve the GameTime toolkit are also summarized. The experimental benchmark suite comprises either loop-free programs, or programs where the loops have statically-known finite bounds and function calls have known finite recursion depths. Experimental evidence shows that GameTime is able to predict the runtime of any program path within a small error margin, without the need to exhaustively enumerate and measure all program paths, using only the measurements of a small number of basis paths. These measurements are sufficient to predict the measurements of paths that are not sampled, and also, within a small error margin, the path with the worst-case execution time and the execution time itself. One experiment also invites the possibility of using GameTime as an energy estimation tool.

Extensions and improvements can be made to the GameTime toolkit to tackle different problems. For example, as suggested in Chapter 4, GameTime could be improved to account for the data-dependency of timing. The current implementation also has a few numerical optimization issues: for example, the threshold below which a row is considered "bad" is arbitrarily chosen. A more programmatic choice for this threshold would be useful. GameTime could also be extended to encode other C constructs, such as function pointers.

The GameTime learning algorithm, which predicts edge weights, could be improved by relaxing the restriction that measurements must be made end-to-end. The current implementation provides rudimentary support for "snipping", whereby a user can attach C labels to two different statements, and GameTime only generates basis paths for the "snipped" portion of the code between the two labels. This mechanism could be employed to measure certain portions of the program under analysis; these measurements can then be used to improve edge weight predictions and thus path length predictions.

There is also ongoing work to use GameTime for termination analysis, for detection of potential timing attacks, and for timing analysis of interrupt-driven programs [28].

Appendix A

Assorted Implementation Details

Microsoft Phoenix instruction operator	GameTime *intermediate expression operator*
`ADD`	`+`
`SUBTRACT`	`-`
`NEGATE`	`-`
`MULTIPLY`	`*`
`DIVIDE`	`/s`, `/u`
`REMAINDER`	`%`
`ASSIGN`	`==`
`SHIFTLEFT`	`<<`
`SHIFTRIGHT`	`>>`, `>>>`
`BITAND`	`&`
`BITOR`	`|`
`BITXOR`	`^`
`BITCOMPLEMENT`	`~`
`COMPARE(GT)`	`>`
`COMPARE(UGT)`	`>u`
`COMPARE(GE)`	`>=`
`COMPARE(UGE)`	`>=u`
`COMPARE(LT)`	`<`
`COMPARE(ULT)`	`<u`
`COMPARE(LE)`	`<=`
`COMPARE(ULE)`	`<=u`
`COMPARE(EQ)`	`==`
`COMPARE(NE)`	`!=`

Table A.1: **GameTime intermediate expression operators.** This table shows the operators in GameTime intermediate expressions that correspond to different operators of Microsoft Phoenix instructions.

GameTime *intermediate expression operator*	*SMT-LIB v2.0 expression operator*
`+`	`bvadd`
`-`	`bvsub`
`-` (negation)	`bvneg`
`*`	`bvmul`
`/s`	`bvsdiv`
`/u`	`bvudiv`
`%`	`bvsmod`
`==`	`=`
`<<`	`bvshl`
`>>`	`bvashr`
`>>>`	`bvlshr`
`&`	`bvand`
`|`	`bvor`
`^`	`bvxor`
`~`	`bvnot`
`concat`	`concat`
`zeroExtend`	`zero_extend`
`signExtend`	`sign_extend`
`bitExtract`	`extract`
`!`	`not`
`>`	`bvsgt`
`>u`	`bvugt`
`>=`	`bvsge`
`>=u`	`bvuge`
`<`	`bvslt`
`<u`	`bvult`
`<=`	`bvsle`
`<=u`	`bvule`
`ite`	`ite`

Table A.2: **SMT-LIB v2.0 expression operators.** This table shows the operators in SMT-LIB v2.0 expressions that correspond to different operators of GameTime intermediate expressions.

```
⟨constant⟩        ::= [0-9]+
                   |  '-' ⟨constant⟩

⟨identifier⟩      ::= [a-zA-Z_](a-zA-Z0-9_)*

⟨variable⟩        ::= ⟨identifier⟩
                   |  ⟨variable⟩ '<' ⟨constant⟩ '>'

⟨expression⟩      ::= ⟨constant⟩
                   |  ⟨variable⟩
                   |  '(' ⟨expression⟩ ')'
                   |  ⟨arith_expr⟩
                   |  ⟨array_expr⟩
                   |  ⟨bitwise_expr⟩
                   |  ⟨Boolean_expr⟩
                   |  ⟨compare_expr⟩
                   |  ⟨offset_expr⟩

⟨arith_expr⟩      ::= ⟨expression⟩ '+' ⟨expression⟩
                   |  ⟨expression⟩ '-' ⟨expression⟩
                   |  ⟨expression⟩ '*' ⟨expression⟩
                   |  ⟨expression⟩ '/s' ⟨expression⟩
                   |  ⟨expression⟩ '/u' ⟨expression⟩
                   |  ⟨expression⟩ '%' ⟨expression⟩

⟨array_expr⟩      ::= ⟨variable⟩ '[' ⟨expression⟩ ']'
                   |  ⟨array_expr⟩ '[' ⟨expression⟩ ']'

⟨bitwise_expr⟩    ::= '~' ⟨expression⟩
                   |  ⟨expression⟩ '&' ⟨expression⟩
                   |  ⟨expression⟩ '|' ⟨expression⟩
                   |  ⟨expression⟩ '^' ⟨expression⟩
                   |  ⟨expression⟩ '<<' ⟨expression⟩
                   |  ⟨expression⟩ '>>' ⟨expression⟩
                   |  ⟨expression⟩ '>>>' ⟨expression⟩
                   |  'concat' '(' ⟨expression⟩ ',' ⟨expression⟩ ')'
                   |  'zeroExtend' '(' ⟨expression⟩ ',' ⟨expression⟩ ')'
                   |  'signExtend' '(' ⟨expression⟩ ',' ⟨expression⟩ ')'
                   |  'bitExtract' '(' ⟨expression⟩ ',' ⟨expression⟩ ',' ⟨expression⟩ ')'

⟨Boolean_expr⟩    ::= 'true'
                   |  'false'
                   |  ⟨expression⟩ 'and' ⟨expression⟩
                   |  ⟨expression⟩ 'or' ⟨expression⟩
                   |  '!' ⟨expression⟩

⟨compare_expr⟩    ::= ⟨expression⟩ '==' ⟨expression⟩
                   |  ⟨expression⟩ '>' ⟨expression⟩
                   |  ⟨expression⟩ '<' ⟨expression⟩
                   |  ⟨expression⟩ '>=' ⟨expression⟩
                   |  ⟨expression⟩ '<=' ⟨expression⟩
                   |  ⟨expression⟩ '!=' ⟨expression⟩
                   |  'ite' '(' ⟨expression⟩ ',' ⟨expression⟩ ',' ⟨expression⟩ ')'

⟨offset_expr⟩     ::= ⟨expression⟩ '.' ⟨expression⟩
```

Figure A.1: **Backus-Nauf Form Grammar of the GameTime intermediate expressions.** These intermediate expressions are meant to resemble C expressions as closely as possible.

Appendix B

Package Overview

The GameTime toolkit is located at http://uclid.eecs.berkeley.edu/gametime. A Python interface to the tool is currently available through the `gametime` package. An example Python program that imports this package and uses its API is shown in Listing B.1. The class `Analyzer` provides many of the commonly used functions of GameTime, including the generation of basis paths and the generation of the longest feasible paths.

Listing B.1: **Example file that uses the Python interface to GameTime.**

```
import os

from gametime import Analyzer, GameTime, PathType
from gametime.fileHelper import createDir
from gametime.projectConfiguration import readProjectConfigFile


def generateBasisPaths(projectConfigXmlFile, analyzerLocation):
    """This sample function demonstrates how to initialize a GameTime project
    with the XML file provided, and how to generate the Path objects that
    represent the basis paths of the code specified in the XML file.

    The function saves the information from the Path objects, and the Analyzer
    object used for analysis itself, to different files.

    @param projectConfigXmlFile XML file used to initialize and configure
        a GameTime project.
    @param analyzerLocation Location of the saved Analyzer object.
    """
    # Initialize the GameTime project with the contents of the XML file.
    projectConfig = readProjectConfigFile(projectConfigXmlFile)

    # Create a new Analyzer object for this analysis.
    analyzer = GameTime.analyze(projectConfig)

    # Generate a list of the Path objects that represent the basis paths of
    # the code specified in the XML file.
    basisPaths = analyzer.generateBasisPaths()

    # To keep the filesystem clean, create a directory for the basis paths
    # in the temporary directory created by GameTime for its analysis.
    # Write the information contained in the Path objects to this directory.
    basisDir = os.path.join(projectConfig.locationTempDir, "basis")
    createDir(basisDir)
    analyzer.writePathsToFiles(basisPaths, writePerPath=False,
                               rootDir=basisDir)

    # Save the analyzer for later use.
    analyzer.saveToFile(analyzerLocation)

def generatePaths(analyzerLocation, basisValuesLocation, numPaths, pathType):
    """This sample function demonstrates how to load an Analyzer object,
    saved from a previous analysis, from a file, and how to load the values
    to be associated with the basis Path objects from a file.

    Once these values are loaded, the function generates as many feasible paths
    as possible, upper-bounded by the "numPaths" argument. This argument is
    ignored if the function is called to generate all of the feasible paths
    of the code being analyzed.

    The type of paths that will be generated is determined by
    the "pathType" argument, which is a class variable of
    the "PathType" class. For a description of the types, refer to
    the documentation of the "PathType" class.

    The function saves the information from the Path objects, and the Analyzer
    object used for analysis itself, to different files.

    @param analyzerLocation Location of the saved Analyzer object.
    @param basisValuesLocation Location of the file that contains
        the values to be associated with the basis Path objects.
    @param pathType Type of paths to generate, represented by a class
        variable of the "PathType" class. The different types of paths are
        described in the documentation of the "PathType" class.
    @param numPaths Upper bound on the number of paths to generate.
    """
    # Load an Analyzer object from a file saved from a previous analysis.
    analyzer = Analyzer.loadFromFile(analyzerLocation)

    # Load the values to be associated with the basis Path objects
    # from the file specified.
    analyzer.loadBasisValuesFromFile(basisValuesLocation)

    # Generate the list of the Path objects that represent
    # the feasible paths requested.
    paths = analyzer.generatePaths(numPaths, pathType)

    # To keep the filesystem clean, create a directory for these feasible
    # paths in the temporary directory created by GameTime for its analysis.
    # Write the information contained in the Path objects to this directory.
    projectConfig = analyzer.projectConfig
    pathsDirName = PathType.getDescription(pathType)
    pathsDir = os.path.join(projectConfig.locationTempDir, pathsDirName)
    createDir(pathsDir)
    analyzer.writePathsToFiles(paths, writePerPath=False, rootDir=pathsDir)

    # Save the analyzer for later use.
    analyzer.saveToFile(analyzerLocation)
```

GameTime can be configured using either the `Configuration` class of the `gametime` package or the configuration XML file presented in Listing B.2.

Listing B.2: **Example GameTime configuration XML file.**

```
<?xml version="1.0" ?>

<!-- This is the XML file that configures GameTime for all projects. -->
<gametime-config>
    <!-- Information about the memory layout of the target machine. -->
    <memory>
        <!-- Word size on the machine that GameTime is being run on. -->
        <bitsize>32</bitsize>
        <!-- Endianness of the target machine. This configuration
             recognizes two options: "big" for big-endian and
             "little" for little-endian. -->
        <endianness>little</endianness>
    </memory>

    <!-- Annotations that can be added to the code under analysis. -->
    <annotations>
        <!-- Annotation that is used when additional conditions need to be
             provided to GameTime, without adding more branches to the code
             under analysis. This is used, for example, to control the values
             that GameTime assigns certain variables in the test cases that
             it produces. -->
        <assume>__gt_assume</assume>
    </annotations>

    <!-- Identifiers that will be appended to different "special"
         variables that will be sent to the underlying SMT solver. -->
    <identifiers>
        <!-- Identifier that gets prepended to the name of either an
             aggregate type or an aggregate object. -->
        <aggregate>__gtAGG__</aggregate>
        <!-- Identifier that gets prepended to the name of a Boolean
             variable that is true if, and only if, a constraint is true. -->
        <constraint>__gtCONSTRAINT</constraint>
        <!-- Identifier that gets prepended to the names of functions
```

```
             that have not been inlined. -->
        <efc>__gtEFC__</efc>
        <!-- Identifier that gets prepended to a field reference. -->
        <field>__gtFIELD__</field>
        <!-- Identifier that gets prepended to the names of variables that
             replace indices within array accesses. -->
        <tempindex>__gtINDEX</tempindex>
        <!-- Identifier that gets prepended to the names of temporary
             pointers. -->
        <tempptr>__gtPTR</tempptr>
        <!-- Identifier used for temporary variables. -->
        <tempvar>__gt</tempvar>
    </identifiers>

    <!-- Names for temporary files and folders that are generated during
         the GameTime toolflow. -->
    <temps>
        <!-- Name for temporary files that store project configuration
             information. -->
        <project-config>project-config</project-config>
        <!-- Name for temporary files that store the locations of
             other files to be included in the source file. -->
        <included>included</included>
        <!-- Name for temporary loop configuration files, which store
             the location of loop headers, generated during the process of
             loop unrolling. -->
        <loop-config>loop-config</loop-config>

        <!-- Suffix for general purposes, such as to append to the name
             of the temporary file to be analyzed by GameTime, and to the
             name of the temporary directory that stores the temporary
             files generated during the GameTime toolflow. -->
        <suffix>-gt</suffix>

        <!-- Name for the GameTime mode when the Phoenix section of
             the toolflow is used to create the DAG containing the basic
             blocks of a function unit. -->
        <phx-create-dag>create-dag</phx-create-dag>
        <!-- Name for temporary files that store a directed acyclic graph
             (in DOT format). -->
        <dag>dag</dag>
        <!-- Name for temporary files that store a mapping between
             the IDs of the basic blocks in the control-flow graph and
             their "adjusted" IDs. -->
        <dag-id-map>dag-id-map</dag-id-map>

        <!-- Name for temporary files that store the Phoenix intermediate
             representation of the function being analyzed. -->
        <phx-ir>ir</phx-ir>
        <!-- Name for the GameTime mode when the Phoenix section of
```

```
            the toolflow is used to find the conditions along paths. -->
        <phx-find-conditions>find-conditions</phx-find-conditions>

        <!-- Name for temporary files that each store the integer linear
            programming problem that, when solved, produced a path. -->
        <path-ilp-problem>path-ilp-problem</path-ilp-problem>
        <!-- Name for temporary files that each store the IDs of the nodes
            in a directed acyclic graph along a path. -->
        <path-nodes>path-nodes</path-nodes>
        <!-- Name for temporary files that each store the line numbers
            of the statements along a path. -->
        <path-line-numbers>path-line-numbers</path-line-numbers>
        <!-- Name for temporary files that each store the conditions
            along a path. -->
        <path-conditions>path-conditions</path-conditions>
        <!-- Name for temporary files that each store information about
            the edges that are associated with the conditions and
            assignments along a path. -->
        <path-condition-edges>path-condition-edges</path-condition-edges>
        <!-- Name for temporary files that each store information about
            the line numbers and truth values of the conditional points
            along a path. -->
        <path-condition-truths>path-condition-truths</path-condition-truths>
        <!-- Name for temporary files that each store information about
            all the array accesses made in conditions and assignments
            along a path. -->
        <path-array-accesses>path-array-accesses</path-array-accesses>
        <!-- Name for temporary files that each store the expressions
            associated with the temporary index variables of aggregate
            accesses along a path. -->
        <path-agg-index-exprs>path-agg-index-exprs</path-agg-index-exprs>
        <!-- Name for temporary files that each store the value associated
            with a path. -->
        <path-value>path-value</path-value>
        <!-- Name for temporary files that store information about
            all of the paths traced during analysis. -->
        <path-all>path-all</path-all>

        <!-- Prefix for temporary files that store the SMT queries
            corresponding to different paths. -->
        <path-query>path-query</path-query>
        <!-- Name for temporary files that contain the models for
            satisfiable SMT queries. -->
        <smt-model>smt-model</smt-model>
        <!-- Name for temporary files that store all of the SMT queries
            that have been made during analysis. -->
        <path-query-all>path-query-all</path-query-all>

        <!-- Prefix for temporary files that store information about
            a case, an assignment to variables that would drive an
```

```
                execution of the code along a path. -->
        <case>case</case>

        <!-- Name for files that store the basis matrix after analysis
                has been conducted to determine feasible basis paths. -->
        <basis-matrix>basis-matrix</basis-matrix>
        <!-- Name for files that store the quantities (for example, execution
                times) associated with running or simulating the program under
                analysis on inputs that drive it along its basis paths. -->
        <basis-values>basis-values</basis-values>
        <!-- Name for temporary files that store a directed acyclic graph
                (in DOT format), where the edges are annotated with weights. -->
        <dag-weights>dag-weights</dag-weights>
        <!-- Name for temporary files that store the distribution of quantities
                (for example, execution times) associated with each path in the
                control-flow graph of the program being analyzed. -->
        <distribution>distribution</distribution>
    </temps>

    <!-- Locations of useful tools. -->
    <tools>
        <!-- Location of the main Phoenix DLL. -->
        <phoenix>bin/GameTime.dll</phoenix>
        <!-- Location of the main Cilly driver. -->
        <cil>cil/bin/cilly</cil>
    </tools>

    <!-- Locations of SMT solvers. -->
    <smt-solvers>
        <!-- Location of the Boolector executable. If the absolute
                location is not provided (as is the case by default),
                and Boolector is used for a query, the location is assumed
                to be present in the PATH environment variable. -->
        <boolector>boolector.exe</boolector>
        <!-- Location of the Python interface of Z3, the SMT solver
                from Microsoft. If the absolute location is not provided
                (as is the case by default), and Z3 is used for a query,
                the location is assumed to be present in either
                the PYTHONPATH environment variable or
                the PATH environment variable. -->
        <z3>z3.pyc</z3>
    </smt-solvers>
</gametime-config>
```

A GameTime project can be configured using either the `ProjectConfiguration` class of the `gametime` package or a project configuration XML file, an example of which is displayed in Listing B.3. A GameTime project specifies, among other things, the function to be analyzed and the path to the file that contains the function. The example XML file configures GameTime to analyze the modular exponentiation code from Figure 2.2(b).

Listing B.3: **Example GameTime project configuration XML file.**

```
<?xml version="1.0" ?>

<!-- This is the XML file that configures GameTime for a specific project. -->
<gametime-project>
    <!-- Information about the function to analyze. -->
    <file>
        <!-- Location of the file containing the function to analyze. -->
        <location>modexp_simple.c</location>
        <!-- Function to analyze. -->
        <analysis-function>modexp_simple</analysis-function>
        <!-- Label in the function to start analysis from.
             If left empty, GameTime will start analysis of the function
             from its beginning. -->
        <start-label></start-label>
        <!-- Label in the function to end analysis at.
             If left empty, GameTime will end analysis of the function
             at its end. -->
        <end-label></end-label>
    </file>

    <!-- Information to preprocess the file using source-to-source
         transformations before analysis. -->
    <preprocess>
        <!-- Locations of other files to be included in the source file.
             More than one file can be specified, and the names must be
             separated by whitespaces. -->
        <include></include>
        <!-- Functions to inline, if any. More than one function
             can be specified, and the names must be separated
             by whitespaces. -->
        <inline></inline>
        <!-- Uncomment this tag to ask GameTime to detect loops. -->
        <!-- <detect-loops/> -->
        <!-- Uncomment this tag to ask GameTime to unroll loops. -->
        <!-- <unroll-loops/> -->
    </preprocess>

    <!-- Configuration options for various features of the analysis. -->
    <analysis>
        <!-- Uncomment this tag to randomize the basis that GameTime
             starts the analysis with. Without randomization,
             the initial basis is the standard basis. -->
        <!-- <randomize-initial-basis/> -->
        <!-- Uncomment this tag to model multi-dimensional arrays
             as nested arrays, or arrays whose elements can also
             be arrays, in an SMT query. If commented out,
             a multi-dimensional array will be modeled as
             a one-dimensional array, and the indices
             of an access will be concatenated. -->
        <!-- <model-as-nested-arrays/> -->
        <!-- Uncomment this tag to prevent the refinement of the basis
             into a 2-barycentric spanner. -->
        <!-- <prevent-basis-refinement/> -->
        <!-- Integer linear programming solver used to solve integer linear
             programs to generate candidate paths. The following options are
             recognized: "cbc" for CBC; "cbc-pulp" for the version of CBC
             provided with the PuLP package; "cplex" for CPLEX;
             "glpk" for GLPK; "gurobi" for Gurobi; "xpress" for Xpress.
             If no solver is specified, the default solver of the PuLP package
             will be used. -->
        <ilp-solver></ilp-solver>
        <!-- SMT solver that GameTime uses to check the satisfiability
             of an SMT query. The following options are recognized:
             "boolector" (or "boolector-lingeling") for Boolector with
             Lingeling as the SAT solver;
             "boolector-minisat" for Boolector with MiniSAT as the SAT solver;
             "boolector-picosat" for Boolector with PicoSAT as the SAT solver;
             "z3" for Z3. -->
        <smt-solver>z3</smt-solver>
    </analysis>

    <!-- Debugging options. -->
    <debug>
        <!-- Uncomment this tag to keep the temporary files that CIL
             produces during its analysis. -->
        <!-- <keep-cil-temps/> -->
        <!-- Uncomment this tag to dump the Phoenix intermediate
             representation of the function under analysis to a file. -->
        <!-- <dump-ir/> -->
        <!-- Uncomment this tag to keep debugging information and
             files produced by the integer linear programming solver. -->
        <!-- <keep-ilp-solver-output/> -->
        <!-- Uncomment this tag to dump information about the path
             being traced. -->
        <!-- <dump-path/> -->
        <!-- Uncomment this tag to dump information about all of the paths
             that have been traced during analysis to a file. -->
        <!-- <dump-all-paths/> -->
        <!-- Uncomment this tag to dump information produced when
             an IR-level instruction is traced backward. -->
        <!-- <dump-instruction-trace/> -->
        <!-- Uncomment this tag to dump information produced when
             an SMT query is created. -->
        <!-- <dump-smt-trace/> -->
        <!-- Uncomment this tag to dump information about all of
             the SMT queries that have been made during analysis
             to a file. -->
        <!-- <dump-all-queries/> -->
        <!-- Uncomment this tag to keep debugging information and
```

```
                files produced by the parser. -->
        <!-- <keep-parser-output/> -->
    </debug>
</gametime-project>
```

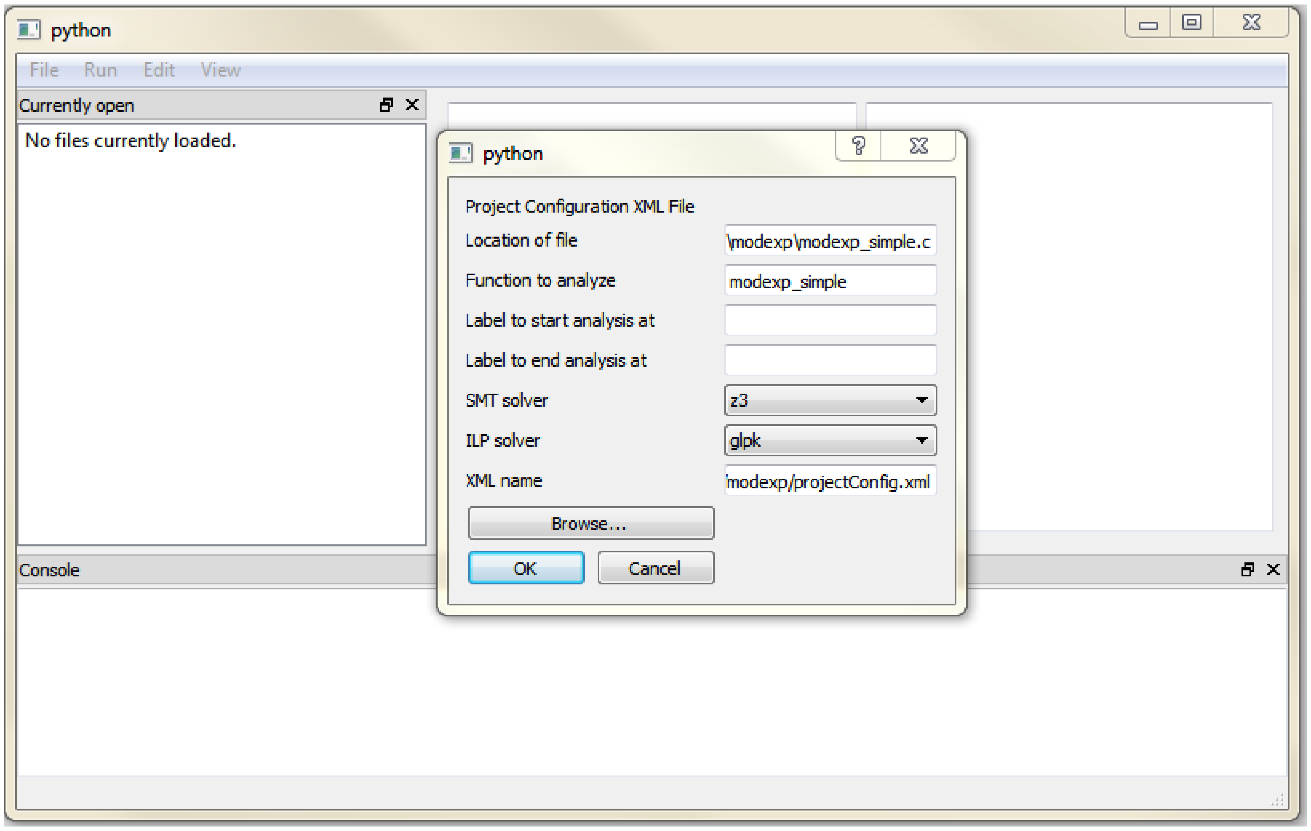
Once the basis paths have been determined, and the corresponding test cases produced, the measurements for each basis path can be provided through a file, where each line is the number of the basis path followed by the measured value of the basis path, separated by whitespace. Lines that start with the # character are ignored as comments. An example of such a file is shown in Listing B.4, which contains cycle counts for the basis paths of (a variant of) the modular exponentiation code from Figure 2.2(b).

Listing B.4: **Example file with basis path measurements.**

```
# These cycle counts were obtained on the PTARM Simulator v1.0, obtained from
# http://chess.eecs.berkeley.edu/pret/src/ptarm-1.0/ptarm_simulator.html.
# The basis paths were generated with Z3 v4.3.0 as the backend SMT solver and
# the GLPK package v1.5.3 as the backend integer linear programming solver.

# Basis path 1:
# exponent = 0x3;
1   764

# Basis path 2:
# exponent = 0x4;
2   684

# Basis path 3:
# exponent = 0x8;
3   684

# Basis path 4:
# exponent = 0xd;
4   847

# Basis path 5:
# exponent = 0xe;
5   840
```

Currently, GameTime also has a rudimentary graphical user interface (GUI), created with the PySide Python package [39] released by Nokia. Figure B.1 demonstrates the GUI before a GameTime project has been started. The dialog box enables a user to specify different options related to a project, such as the SMT solver used and the path to the file that contains the function to be analyzed.

Figure B.2 demonstrates the GUI after it has generated the basis paths for the code under analysis. In the example shown, the code under analysis is the modular exponentiation code from Figure 2.2(b), modified for 4-bit exponents. The C statements along one of the basis paths have been highlighted.



No files currently loaded.

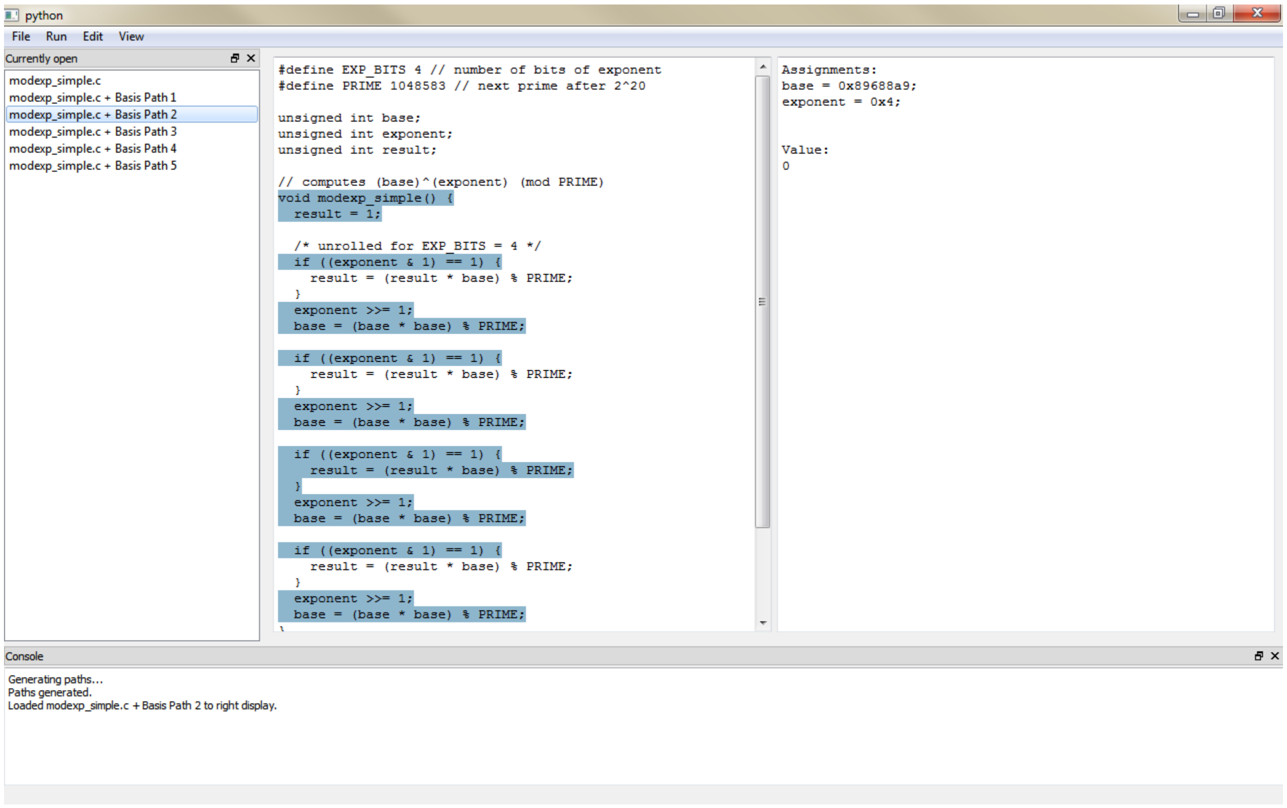
Project Configuration XML File

Location of file	\modexp\modexp_simple.c
Function to analyze	modexp_simple
Label to start analysis at	
Label to end analysis at	
SMT solver	z3
ILP solver	glpk
XML name	modexp/projectConfig.xml

Figure B.1: **GUI demonstration: New GameTime project is started.**

In Figure B.3, the GUI provides the user with a dialog box to enter the measurements for the test cases that correspond to the basis paths. These measurements can be supplied either manually or through the import of a file in the format shown in Listing B.4.

Finally, once the measurements for the test cases that correspond to the basis paths have been entered, the GUI allows the user to determine the longest feasible (or the worst-case) path. In Figure B.4, the user has chosen to determine the five worst-case paths. The worst-case path is highlighted in the middle panel and its corresponding test case and predicted length are shown in the rightmost panel. As shown in Figure B.4, the exponent 15, with all of the bits in its bit representation set to 1, drives the modular exponentiation code along its worst-case path.

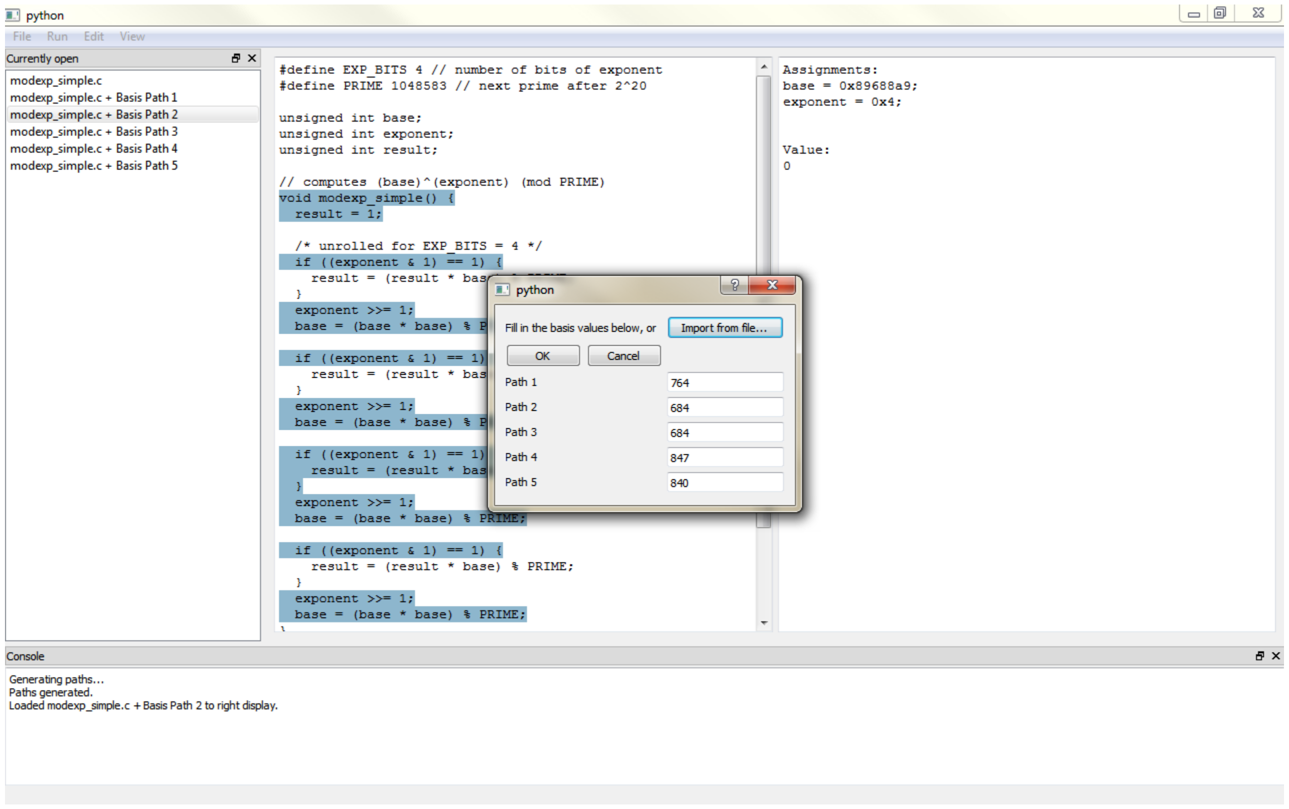

python
File
Run
Edit
View
Currently open
modexp_simple.c
modexp_simple.c + Basis Path 1
modexp_simple.c + Basis Path 2
modexp_simple.c + Basis Path 3
modexp_simple.c + Basis Path 4
modexp_simple.c + Basis Path 5
Console

```
#define EXP_BITS 4 // number of bits of exponent
#define PRIME 1048583 // next prime after 2^20

unsigned int base;
unsigned int exponent;
unsigned int result;

// computes (base)^(exponent) (mod PRIME)
void modexp_simple() {
  result = 1;

  /* unrolled for EXP_BITS = 4 */
  if ((exponent & 1) == 1) {
    result = (result * base) % PRIME;
  }
  exponent >>= 1;
  base = (base * base) % PRIME;

  if ((exponent & 1) == 1) {
    result = (result * base) % PRIME;
  }
  exponent >>= 1;
  base = (base * base) % PRIME;

  if ((exponent & 1) == 1) {
    result = (result * base) % PRIME;
  }
  exponent >>= 1;
  base = (base * base) % PRIME;

  if ((exponent & 1) == 1) {
    result = (result * base) % PRIME;
  }
  exponent >>= 1;
  base = (base * base) % PRIME;
}
```

```
Assignments:
base = 0x89688a9;
exponent = 0x4;


Value:
0
```

```
Generating paths...
Paths generated.
Loaded modexp_simple.c + Basis Path 2 to right display.
```

Figure B.2: **GUI demonstration: Basis paths have been generated.**



modexp_simple.c
modexp_simple.c + Basis Path 1
modexp_simple.c + Basis Path 2
modexp_simple.c + Basis Path 3
modexp_simple.c + Basis Path 4
modexp_simple.c + Basis Path 5

```
#define EXP_BITS 4 // number of bits of exponent
#define PRIME 1048583 // next prime after 2^20

unsigned int base;
unsigned int exponent;
unsigned int result;

// computes (base)^(exponent) (mod PRIME)
void modexp_simple() {
  result = 1;

  /* unrolled for EXP_BITS = 4 */
  if ((exponent & 1) == 1) {
    result = (result * bas
  }
  exponent >>= 1;
  base = (base * base) % P

  if ((exponent & 1) == 1)
    result = (result * bas
  }
  exponent >>= 1;
  base = (base * base) % P

  if ((exponent & 1) == 1)
    result = (result * bas
  }
  exponent >>= 1;
  base = (base * base) % PRIME;

  if ((exponent & 1) == 1) {
    result = (result * base) % PRIME;
  }
  exponent >>= 1;
  base = (base * base) % PRIME;
```

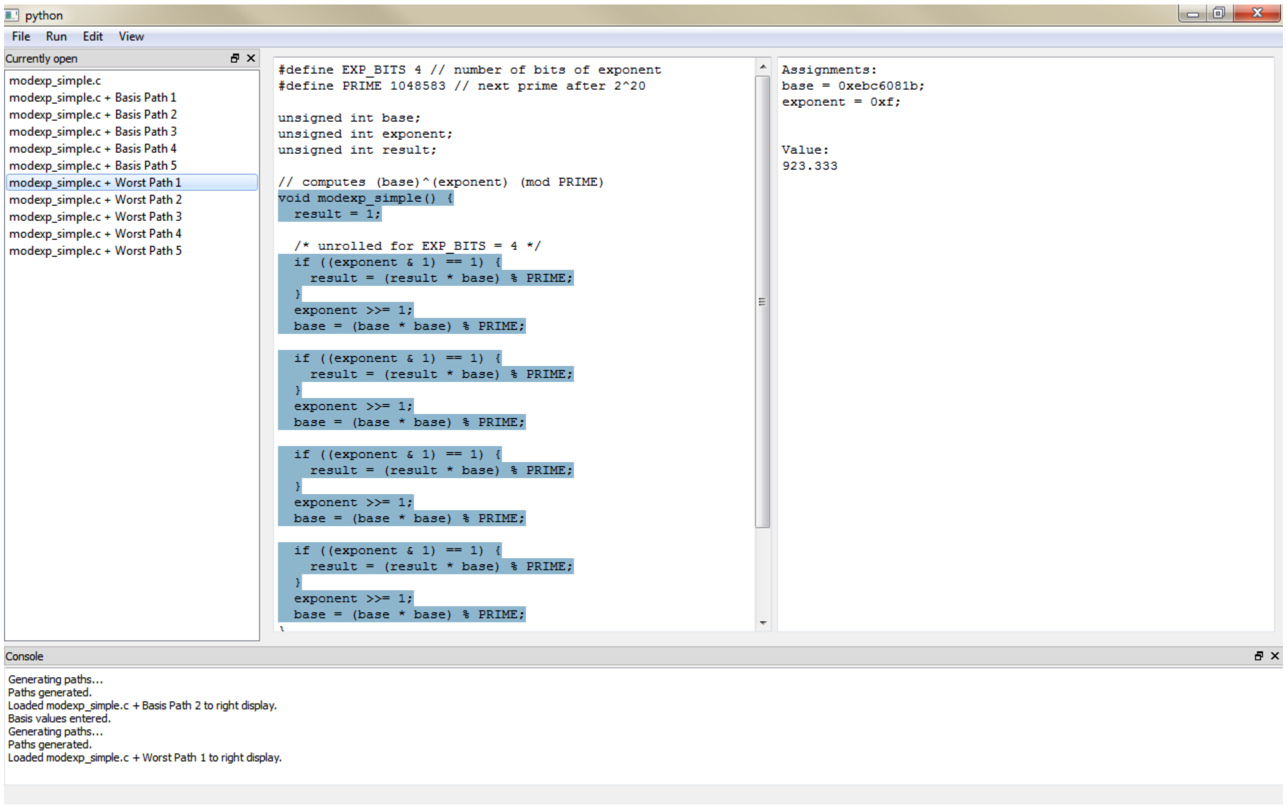
```
Assignments:
base = 0x89688a9;
exponent = 0x4;

Value:
0
```

Fill in the basis values below, or

Path 1	764
Path 2	684
Path 3	684
Path 4	847
Path 5	840

```
Generating paths...
Paths generated.
Loaded modexp_simple.c + Basis Path 2 to right display.
```

Figure B.3: **GUI demonstration: Measurements of the test cases are entered.**



modexp_simple.c
modexp_simple.c + Basis Path 1
modexp_simple.c + Basis Path 2
modexp_simple.c + Basis Path 3
modexp_simple.c + Basis Path 4
modexp_simple.c + Basis Path 5
modexp_simple.c + Worst Path 1
modexp_simple.c + Worst Path 2
modexp_simple.c + Worst Path 3
modexp_simple.c + Worst Path 4
modexp_simple.c + Worst Path 5

```
#define EXP_BITS 4 // number of bits of exponent
#define PRIME 1048583 // next prime after 2^20

unsigned int base;
unsigned int exponent;
unsigned int result;

// computes (base)^(exponent) (mod PRIME)
void modexp_simple() {
  result = 1;

  /* unrolled for EXP_BITS = 4 */
  if ((exponent & 1) == 1) {
    result = (result * base) % PRIME;
  }
  exponent >>= 1;
  base = (base * base) % PRIME;

  if ((exponent & 1) == 1) {
    result = (result * base) % PRIME;
  }
  exponent >>= 1;
  base = (base * base) % PRIME;

  if ((exponent & 1) == 1) {
    result = (result * base) % PRIME;
  }
  exponent >>= 1;
  base = (base * base) % PRIME;

  if ((exponent & 1) == 1) {
    result = (result * base) % PRIME;
  }
  exponent >>= 1;
  base = (base * base) % PRIME;
```

```
Assignments:
base = 0xebc6081b;
exponent = 0xf;


Value:
923.333
```

```
Generating paths...
Paths generated.
Loaded modexp_simple.c + Basis Path 2 to right display.
Basis values entered.
Generating paths...
Paths generated.
Loaded modexp_simple.c + Worst Path 1 to right display.
```

Figure B.4: **GUI demonstration: Worst-case path has been determined.**

Bibliography

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. “Detecting equality of values in programs”. In: *Conference Record of the 15th ACM Symposium on Principles of Programming Languages*. 1988, pp. 1–11.

[2] B. Awerbuch and R. D. Kleinberg. “Adaptive routing with end-to-end feedback: distributed learning and geometric approaches”. In: *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing (STOC)*. 2004, pp. 45–53.

[3] Thomas Ball et al. *Efficient evaluation of pointer predicates with Z3 SMT Solver in SLAM2*. Tech. rep. Citeseer, 2010.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. URL: `http://smtlib.cs.uiowa.edu/logics/QF_AUFBV.smt2`.

[5] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. 2010. URL: `http://www.SMT-LIB.org/`.

[6] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. “RWset: Attacking path explosion in constraint-based test generation”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 351–366.

[7] Robert Brummayer and Armin Biere. “Boolector: An efficient SMT solver for bit-vectors and arrays”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, pp. 174–177.

[8] Randal E. Bryant, Shuvendu Lahiri, and Sanjit A. Seshia. “Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions”. In: *Proceedings of the 14th International Conference on Computer Aided Verification*. 2002, pp. 78–92.

[9] COIN-OR. *CBC (Coin Branch and Cut)*. 2013. URL: `https://projects.coin-or.org/Cbc/`.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs”. In: *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. 2008, pp. 209–224.

[11] David R Cok. *The SMT-LIB v2.0 language and tools: A tutorial*. Tech. rep. Technical report, GrammaTech, Inc, 2011.

[12] iRobot Corporation. *iRobot Create User's Manual.* 2006. URL: http://www.irobot.com/filelibrary/create/Create%20Manual_Final.pdf.

[13] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[14] Prabal Dutta et al. "Energy metering for free: Augmenting switching regulators for real-time monitoring". In: *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on*. IEEE. 2008, pp. 283–294.

[15] GNU. *GNU ARM Toolchain for Cygwin, Linux and MacOS*. 2006. URL: http://www.gnuarm.com/.

[16] David Gay et al. "The nesC language: A holistic approach to networked embedded systems". In: *ACM Sigplan Notices*. Vol. 38. 5. ACM. 2003, pp. 1–11.

[17] Graphviz. *DOT Documentation*. 2013. URL: http://www.graphviz.org/Documentation.php.

[18] Graphviz. "Graphviz - Graph Visualization Software". In: (2013). URL: http://www.graphviz.org/.

[19] Jan Gustafsson et al. "The Mälardalen WCET Benchmarks: Past, Present And Future". In: *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*. 2010, pp. 137–147.

[20] Ben Hardekopf and Calvin Lin. "Flow-Sensitive Pointer Analysis for Millions of Lines of Code". In: *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE. 2011, pp. 289–298.

[21] David Harel et al. "Statemate: A working environment for the development of complex reactive systems". In: *Software Engineering, IEEE Transactions on* 16.4 (1990), pp. 403–414.

[22] Michael Hind. "Pointer Analysis: Haven't We Solved This Problem Yet?" In: *Proceedings of PASTE'01*. 2001, pp. 54–61.

[23] Center for Hybrid and Embedded Software Systems (CHESS). *PTARM Simulator v1.0 Documentation*. 2013. URL: http://chess.eecs.berkeley.edu/pret/src/ptarm-1.0/ptarm_simulator.html.

[24] Center for Hybrid and Embedded Software Systems (CHESS). *Precision Timed (PRET) Machines*. 2013. URL: http://chess.eecs.berkeley.edu/pret/.

[25] IBM. *CPLEX Optimizer*. 2013. URL: http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html.

[26] Texas Instruments Incorporated. *MSP430^{TM} Ultra-Low-Power Microcontrollers*. 2013. URL: http://www.ti.com/lit/sg/slab034v/slab034v.pdf.

[27] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Second. Prentice Hall Software Series, 1988.

[28] Jonathan Kotker, Dorsa Sadigh, and Sanjit A. Seshia. "Timing Analysis of Interrupt-Driven Programs under Context Bounds". In: *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 2011, pp. 81–90.

[29] Miads Lab. *RTE-2000H-TP Real Time Evaluator Product Information*. 2013. URL: http://www.midas.co.jp/products/rte-2000h-tp.htm.

[30] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. UC Berkeley: LeeSeshia.org, 2011.

[31] Yau-Tsun Steven Li and Sharad Malik. *Performance Analysis of Real-Time Embedded Software*. Khuwer Academic, 1999.

[32] Isaac Liu et al. "A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance". In: *Proceedings of International Conference on Computer Design (ICCD)*. 2012.

[33] Microsoft. "Phoenix Documentation". In: (2008).

[34] Microsoft. *Phoenix software optimization and analysis framework*. 2007. URL: http://connect.microsoft.com/Phoenix/.

[35] Leonardo de Moura. *Support for Multi-Dimensional Arrays in QF_AUFBV?* 2012. URL: http://stackoverflow.com/q/13443259/1834042/.

[36] George C. Necula. *CIL - Infrastructure for C Program Analysis and Transformation (v. 1.3.7)*. 2013. URL: http://www.cs.berkeley.edu/~necula/cil/.

[37] George C. Necula et al. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs". In: *Proceedings of Conference on Compiler Construction (CC)*. 2002, pp. 213–228.

[38] Fadia Nemer et al. "Papabench: a free real-time benchmark". In: *WCET Workshop*. 2006.

[39] Nokia. *PySide: Python for Qt*. 2013. URL: http://qt-project.org/wiki/PySide/.

[40] Gurobi Optimization. *Gurobi Optimizer*. 2013. URL: http://www.gurobi.com/.

[41] GNU Project. *GLPK – GNU Linear Programming Kit*. 2013. URL: http://www.gnu.org/software/glpk/glpk.html.

[42] Wei Qin and S. Malik. "Flexible and formal modeling of microprocessors with application to retargetable simulation". In: *Design, Automation and Test in Europe Conference and Exhibition, 2003*. 2003, pp. 556–561. DOI: 10.1109/DATE.2003.1253667.

[43] Red Hat, Inc. *Cygwin*. 2013. URL: http://www.cygwin.com/.

[44] Red Hat, Inc. *SimIt-ARM*. 2007. URL: http://simit-arm.sourceforge.net/.

[45] Sanjit A. Seshia and Jonathan Kotker. "GameTime: A Toolkit for Timing Analysis of Software". In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2011, pp. 388–392.

[46] Sanjit A. Seshia and Alexander Rakhlin. “Game-Theoretic Timing Analysis”. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE Press, 2008, pp. 575–582.

[47] Sanjit A. Seshia and Alexander Rakhlin. “Quantitative Analysis of Systems Using Game-Theoretic Learning”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 11.S2 (2012), p. 55.

[48] TinyOS. *TinyOS*. 2013. URL: `http://www.tinyos.net/`.

[49] University of Illinois at Urbana-Champaign. *The LLVM Compiler Infrastructure*. 2013. URL: `http://llvm.org`.

[50] Reinhard Wilhelm. “Determining bounds on execution times”. In: *Handbook on Embedded Systems* (2005).

[51] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. “Pointer Analysis for Programs with Structures and Casting”. In: (1999).

[52] Stuart Mitchell et al. *PuLP: An LP modeler in Python*. 2013. URL: `https://code.google.com/p/pulp-or/`.