AN EVENT-BASED LANGUAGE FOR PROGRAMMABLE DEBUGGING

by

Serguei Makarov

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2018 by Serguei Makarov
Abstract

An Event-Based Language for Programmable Debugging

Serguei Makarov

Master of Applied Science

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering

University of Toronto

2018

Conventional debuggers do not provide an efficient means to perform complex observations. This has motivated the development of tools for programmable debugging. Heavyweight instrumentation frameworks such as Valgrind and DynamoRIO can perform a wide variety of observations, but development of plugins for these frameworks is a difficult and low-level task. The event-based approach introduced in the DTrace and SystemTap languages allows observations to be expressed compactly, but these languages lack the flexibility to observe combinations of events required for debugging.

This thesis introduces E, an event-based language for debugging complex software problems, designed around a high-level model of the state transitions and events within a program’s execution. E provides a simple and expressive means for programming heavyweight instrumentation plugins. The thesis presents an E implementation based on heavyweight instrumentation frameworks and demonstrates how E can simplify the diagnosis process for complex real-world debugging scenarios.
Acknowledgements

I am deeply grateful to all of the people who made the completion of this thesis possible.

I would like to thank my advisor, Ding Yuan, for his invaluable support of this project, both for his skepticism which pushed me to search for compelling use cases and for his enthusiasm which has helped me push the work closer and closer to real-world use.

I would also like to thank my thesis committee members Tarek Abdelrahman, Faith Ellen, Michael Stumm and Ben Liang. In particular, Tarek Abdelrahman and Faith Ellen have gone above and beyond in providing valuable feedback for the improvement of this thesis in advance of the defence. I would like to especially thank Faith Ellen for providing additional detailed writing suggestions after the defence.

I would like to thank the engineering team at Red Hat. In particular, Frank Ch. Eigler supervised my work on the SystemTap project, creating a welcoming environment where I gained technical skills and accumulated the inspiration to build an event-based language that could be used for in-depth debugging. David Malcolm and his work on GDB scripting were a source of inspiration in themselves, and also became the source of a valuable motivating example for my own project.

I would like to thank Angela Demke Brown and Ashvin Goel for supervising my USRA undergraduate research project in the summer of 2014, when I prototyped some of the ideas that led to the present work.

I would like to thank members of my research group for the valuable discussions about this project and for providing feedback on my reading group presentations. In particular David Lion, Xiang Ren, Yu Luo, Kirk Rodrigues, Adrian Chiu and Xu Zhao have shown interest in my work and been valuable sources of support.

Finally, I would like to thank my family – my parents Vladimir Makarov and Nadia Makarov – for their absolutely crucial advice and encouragement, which have helped me persevere with my project.
# Contents

Acknowledgements ........................................ iii  
Table of Contents ........................................ iv  
List of Tables ........................................... vii  
List of Figures ........................................... viii

1 Introduction ........................................... 1

2 Background and Research Problem ................. 3  
   2.1 Interactive Debuggers ................................. 4  
   2.2 Event-based Observation ............................ 5  
   2.3 Heavyweight Instrumentation ....................... 8  
   2.4 Basic Insight ........................................ 10

3 Basic Concepts of Event-Based Tracing ............ 12  
   3.1 State Transitions, Events, and Probes ............. 12  
   3.2 Predefined Events ................................... 13  
   3.3 Event Expressions and Filtering Conditions ....... 15  
   3.4 Nested State Transitions ............................ 15  
   3.5 Probe Directives and Handler Blocks ............... 17  
   3.6 Function Declarations ................................ 18  
   3.7 Type System and Global Data Declarations ....... 18  
   3.8 Shadow Arrays ....................................... 19  
   3.9 Structure and Execution of an E Program ......... 19

4 Example Debugging Scenario ......................... 21  
   4.1 Original Debugging Process ......................... 21  
   4.2 Debugging the Same Problem with E ............... 24  
   4.3 Comparison to Other Debugging Approaches ....... 28

5 Language Specification ................................. 29  
   5.1 Syntax and Lexical Vocabulary ..................... 29  
   5.2 Designators, Values, and Types .................... 30  
   5.3 Expressions .......................................... 31  
      5.3.1 Literals, Designators, and Built-In Operations .. 32  
      5.3.2 Target Program Operations ..................... 33
5.3.3 Precedence Rules .................................................. 34
5.4 Events, Program State Values, and Context Values .............................. 35
5.5 Event Expressions ..................................................... 35
5.6 Statements ........................................................... 36
5.7 Declarations and Scope Rules ............................................ 37
  5.7.1 Probe Directive ................................................... 38
  5.7.2 Data Declaration .................................................. 38
  5.7.3 Function Declaration ............................................. 39
  5.7.4 Event Declaration ............................................... 39
5.8 Built-In Functions ..................................................... 40
  5.8.1 Format Strings and Interpolation .................................. 40
5.9 Predefined Events and Contexts ......................................... 41

6 Language Translator 43
  6.1 Bridging the E Language and Target Framework Functionality ............... 45
  6.2 Parsing Pass ................................................................ 46
  6.3 IR Construction Pass .................................................. 46
    6.3.1 Translation – Declarations .................................... 46
    6.3.2 Translation – Basic-Block IR ................................... 46
  6.4 Specialization Pass ..................................................... 49
    6.4.1 The Scope Labeling Step ........................................ 50
    6.4.2 The Inlining Step ................................................ 51
    6.4.3 The Type Inference Step ......................................... 51
  6.5 Plugin Structure and Runtime Support Layer ................................... 52
    6.5.1 Structure of the Generated Plugin .............................. 53
    6.5.2 Technical Details of Event Handling ............................ 55
    6.5.3 Locking .................................................................. 55
    6.5.4 Data Structures – Associative Arrays .......................... 55
    6.5.5 Data Structures – Shadow Arrays .............................. 57
  6.6 Code Generation Pass .................................................. 57

7 Evaluation 61
  7.1 Benchmark Programs ................................................... 61
  7.2 Experiments and Results .............................................. 66

8 Comparison to Related Work 70
  8.1 Model of State Transitions and Events .................................. 70
  8.2 Hierarchy of Context Data ............................................. 71
  8.3 Observation of Frequently Occurring Events ............................... 72

9 Discussion 73
  9.1 New Language Features ............................................... 73
  9.2 Optimizations .......................................................... 75
  9.3 Modular Observation Tools ............................................ 75
  9.4 Alternate Backends ...................................................... 76
List of Tables

6.1 Instruction set of the E intermediate representation. Terms in angular brackets denote instruction operands: ⟨bb⟩ denotes a basic block operand, ⟨val⟩ denotes a constant or variable operand, ⟨lval⟩ denotes a variable operand, ⟨fn⟩ denotes a function operand. ⟨event⟩ denotes the name of an event. 48
List of Figures

2.1 Defining a static tracepoint in a C program. .............................................. 6

3.1 State transitions and their related predefined events. Each state transition has two events identifying its beginning and end. A function call state transition is considered to consist of three phases, resulting in two additional predefined events indicating the start and end of execution of the function body. .............................................................. 13

3.2 Events which can access the context of a state transition. Context values for the function call state transition can be accessed from the start and end points of the state transition, as well as from an event nested within the function call. The context of a state transition is thus considered to exist throughout its entire duration. .............................................................. 16

3.3 Nested state transitions whose context can be accessed from one event. A handler for an event occurrence before the instruction state transition can access context values from the innermost containing statement and function call using the corresponding contexts stmt and fn. (Contexts marked * will be added in a future version of the language to provide access to additional outer state transitions.) .............................................................. 16

3.4 fcall-counts.ebt, a simple script which counts the number of function calls occurring in a file. .................................................................................. 20

4.1 An ma_table array containing active, dummy and unused entries, as well as a corrupted entry. ................... 22

4.2 A program in the E language written to track corrupted ma_table objects in the Python interpreter. ........... 25

6.1 use-after-free, a simple use-after-free detector written in E. ........................................ 44

6.2 A probe directive in E and its abstract syntax tree and basic-block IR representations. .......................... 47

6.3 Expansions of some E language constructs into equivalent code based on simpler operations.
These expansions are performed on the AST during generation of basic-block IR. %r0, %r1, ...
denote temporary variables. %assoc_iter_start, %assoc_iter_next, %assoc_iter_key,
%assoc_iter_val are built-in functions (provided by the runtime, but not exposed in the E lan-
guage) operating on an ‘iterator’ value of type opaque. ........................................ 49

6.4 Structure of the generated plugin, showing how elements of the plugin interact to observe a single predefined event A. Plugins observing multiple predefined events will contain more than one event handler. .............................................................. 54

6.5 A simplified version of the template used for generating a Valgrind plugin, with trigger points for the program.before, program.after, bb.instrument, obj.alloc and obj.access events. .............................................................. 56
6.6 An example of source code for a probe handler generated from the `obj.access` probe in the use-after-free detector from Figure 6.1. (The generated code has been edited slightly for the sake of brevity.) Note that the code for checking filtering conditions (`bb1,bb3` in the generated probe handler) and the code for the probe's body have been merged into a single function by the translation procedure. Temporary variables used to calculate the filtering conditions are 'mangled' by adding the `\_l13` prefix, while temporary and local variables used by the probe directive's body are prefixed with `\_l14`.

7.1 `fcalls`, a simple function call tracer written in E. ........................................ 62
7.2 `benchmark-fprobes`, a version of the function call tracer with a reduced number of `printf` calls. ............................................................... 62
7.3 `memwatch`, a simple use-after-free detector written in E which also counts memory object accesses. ........................................ 62
7.4 `benchmark-memprobes`, a simplified version of the use-after-free detector without `printf` calls. ............................................................... 63
7.5 `widget.c`, a simple C program which allocates memory objects and contains a use-after-free error. ............................................................... 63
7.6 A portion of the output produced by running the `fcalls` E program on `widget.c` . . . 64
7.7 The output produced by running the `memwatch` E program on `widget.c` . . . . . . . 65
7.8 Comparison of different observation tools for five SPEC2000 integer benchmarks. . . 67
7.9 The effect of reducing the number of watched objects on the overhead of `memwatch` using the Valgrind implementation of E, as measured on the `176.gcc` SPEC 2000 benchmark. “1/N-watchpoints” denotes that only 1 in every N objects allocated by the target program were added to the `watched` array. ............................................................... 68
7.10 Results for the watchpoints experiment after incorporating the shadow memory implementation from Memcheck. ............................................................... 69

8.1 An illustration of how SystemTap and E provide access to the target program’s state. E introduces a hierarchy of contexts which creates a logical place for information about the broader context of an event occurrence. ............................................................... 71
Chapter 1

Introduction

The tasks of diagnosing software defects, debugging, and performance analysis require the programmer to observe the behaviour of a program during its execution. Currently, there is a wide variety of tools intended for performing such observations. These tools are based on different models of the program’s execution and make use of different mechanisms for extracting information. There is a wide variety of existing interfaces and languages for describing program observations. Many observations which are necessary to perform during debugging and performance analysis are not easy to express using any of these interfaces, and carrying them out often requires tedious manual work.

For example, conventional debuggers such as GDB step through the program’s execution and allow the programmer to manually examine one program state at a time. This model makes it difficult to perform observations of program states across the entirety of a particular execution when such observations require information to be collected and combined from program states occurring at numerous points in time during that execution. Conventional debuggers also require the programmer to specify where to interrupt the program in terms of source code locations, but source code locations identify points in time very ambiguously – one source code location may be visited at numerous points in time during a particular execution, and therefore numerous interruptions of the program may be required in order to locate a particular point in time.

Programmable tracing frameworks such as DTrace and SystemTap are more suited to observing numerous program states, but their designs and capabilities have been limited by the goal of being safe to use on production software rather than in a debugging scenario.

Finally, heavyweight instrumentation frameworks such as Valgrind can perform extremely detailed observations, but require custom observations to be specified in terms of low-level program instrumentation. In practice, most developers only use prebuilt tools on top of these frameworks to detect a few commonly occurring error patterns such as memory errors or data races.

This thesis proposes a model for analyzing a program’s execution in terms of state transitions and events, and introduces E: a language which is designed for specifying program observations based on this model. E is designed to be independent of the underlying mechanisms used for observation, and allows implementation on top of most existing observation tools.

Chapter 2 of this thesis gives a detailed background of current observation tools and examines their limitations. Chapter 3 introduces our proposed model and shows how it is used in the design of the E language. Chapter 4 motivates the use of the E language for debugging by showing how it can simplify the diagnosis process in a complex debugging scenario. Chapter 5 contains a full specification of the E language, and Chapter 6 describes
an implementation of the E language based on the heavyweight instrumentation frameworks Valgrind and DynamoRIO, which have previously lacked an easy-to-use interface for describing custom observations. Chapter 7 contains a detailed performance evaluation of the E Tracer, while Chapter 8 compares the design of E with prior programmable tracing frameworks, Chapter 9 outlines further directions for the development of the language, and Chapter 10 gives concluding remarks.
Chapter 2

Background and Research Problem

Every stage of the software development cycle is imperfect and has the potential to introduce defects into a software system. Design and specification, development, testing, and even the process of fixing defects introduced by earlier stages of development [24] are all activities that include room for error and cannot be expected to produce ideal code [14]. Any software system contains defects that may surface in both testing and production as visible problems, either as failures affecting the correct behaviour of the system or as performance issues. The process of identifying and eliminating the defect that causes a particular problem is commonly known as debugging when the problem affects the correct behaviour of the program, or as performance analysis or performance debugging when the problem affects its performance. The boundary between ordinary debugging and performance debugging can be unclear, as performance issues can cause incorrect behaviour in a complex system [8].

The external symptoms of a problem can be extremely varied, and in a great many cases do not provide sufficient information to identify an underlying defect. Moreover, the external symptoms of a problem often do not suggest an obvious strategy for how to gather useful information about internal states of the program. Examples of problems that are difficult to diagnose include errors in complex output formats (such as machine code generated by a compiler), performance issues or resource leaks, and pervasive defects arising from design flaws introduced during early stages of the development cycle.

The diagnosis process for such problems tends to be an iterative process requiring continued human involvement, since the problems arise from defects in the program’s semantics – that is, deviations from the behaviour of the program as intended by the developer. During the diagnosis process, the human developer must actively carry out tasks such as examining the external symptoms of the problem and the program’s codebase, applying their judgment to determine which internal states of the program to observe, and identifying mistaken assumptions in the program’s design and implementation.

The variety of tasks performed during an investigation of a problem’s causes gives rise to a large variety of debugging tools, with no single tool applicable to all possible tasks. A variety of mechanisms have been proposed and implemented for collecting information from a program’s execution. These mechanisms are subject to different technical limitations on the information that can be collected and provide different means for controlling an observation.

Here, by ‘observation’ we mean any process of intervening in the program’s execution with the goal to extract information such as internal states of the program or the sequence of instructions executed. It is important to emphasize that observation of a program is not a passive process but can only be done by actively intervening in the execution of the program with some mechanism that alters or interrupts the program to collect additional
Debugging and performance analysis tools can be classified according to a wide variety of criteria, such as the underlying observation mechanism used to obtain information from the program or the manner in which the tool is expected to be used by the developer.

Different tools are based on different representations of the program, which determine the conceptual model used to control the observation. The conceptual model provided by a tool also determines how information about such things as memory layout and program state is communicated to the developer. This model may be based on the concepts of the high-level language used to write the program, or it may be defined in terms of the concepts of the hardware that the program runs on.

At the same time, different tools provide the developer with different means of requesting information. Depending on the tool, the developer can request information either in an interactive question-and-answer style or by formulating a batch request.

In addition, debugging tools vary in their manner of usage, from fully-interactive debuggers which require the developer to control the observation in detail through an interactive interface, to bug-finding tools that attempt to automatically detect a limited category of error conditions without any input from the developer.

2.1 Interactive Debuggers

Interactive debuggers are the most commonly used tools which follow a question-and-answer model. One of the most notable interactive debuggers is GDB [26], used for debugging programs in the C/C++ runtime environment. Many other language environments, including the Java Virtual Machine [34, 35] and higher-level scripting languages such as Python and Ruby [32, 33], provide debugging tools whose interface closely imitates the GDB model.

Interactive debuggers allow a developer to place breakpoints in order to pause the execution of a target program at a specified point in time. Once the program has been paused, the developer can use the debugger to interactively inspect its internal state using actions such as viewing the program’s call stack, querying the contents of variables and data structures, and manually stepping through the program’s execution one statement at a time in order to further pinpoint the occurrence of some event. Interactive debuggers also include capabilities for setting watchpoints, which notify the developer when a memory location is accessed or written [27]. There has also been research which aims to implement reversible debugging [18, 15]. A reversible debugger uses a record-and-replay runtime which records nondeterministic operations performed by the program, as well as periodic checkpoints of the program state, retaining sufficient information from the program’s execution to exactly reconstruct the program’s behaviour from any point in time. This allows the debugger to support commands that return the program to an earlier point of its execution. Such debuggers are rarely used in practice, as reliable record-and-replay runtimes are not available in many development environments.

The ability provided by an interactive debugger to pause the execution and manually inspect one state at a time leads to a significant advantage: the developer can observe one part of the program and then use that information in order to decide what to observe in a later portion of the program without starting a new execution.

Interactive debuggers are subject to a number of limitations. Although well-suited to sequential programs, the interactive model is less effective for studying the behaviour of parallel software, which has become far more common in recent years, because the debugger interface requires the programmer to manually keep track of and control multiple threads of execution. Interactive debugging is also slow and tedious for performing complex observations that require information to be collected from events occurring at numerous points in time. For
example, in order to understand some problems, the programmer may have to check a complex data structure at numerous points in time in order to locate the exact point in time when an element of the data structure was first corrupted. For example, a problem involving data corruption was found in an OS installation program [11]. The process of solving this problem required the programmer to repeatedly check a correctness invariant on a hash table data structure, and had to be partially automated. Without any automation, such complex observations can translate into hundreds or even thousands of debugger commands.

The desire to overcome these limitations has led to the development of sophisticated scripting functionality that aims to support programmable debugging [23]. The ideal of programmable debugging is that any complex observation should be possible to express as a script instead of needing to use an interactive interface to carry out individual steps of the observation by hand. Effectively, it should be feasible to extend the debugger’s functionality at any time to handle unexpected situations [30]. The resulting debugger extensions may query information (such as variables or correctness invariants) that is particular to a single program or even one specific problem [11], or they may query information about a widely used library or runtime feature and therefore be reusable across a range of situations [31].

There have been efforts to implement programmable debugging by extending a debugger’s command set with features that allow it to be used as a scripting language, or by providing access to the command set of a debugger from an existing full-featured scripting language. For example, GDB provides interfaces for writing extensions in Python and Guile Scheme [28], while also including programming-language-like constructs in its built-in command set [29].

However, there are a number of challenges to fully realizing the ideal of programmable debugging using interactive debuggers as a starting point. Because the command sets of most debuggers were originally designed for interactive use, they result in a programming interface that is unwieldy and difficult to write programs for. Any observation must be phrased in terms of procedural instructions to locate and inspect one state of the program at a time, which can result in complex sequences of actions being required to perform conceptually simple observations, such as finding which statement most recently wrote to a memory location. The performance of interactive debuggers such as GDB is limited by the fact that the debugger operates in a separate process and address space from the target program. Because of this design, every observation on the target program, even one as simple as reading a value from the target program’s memory, incurs a system call. In addition, every transfer of control between the target program and the debugger requires a context switch. While the cost of a context switch is negligible from the point of view of a user who interactively inputs one debugger command at a time, context switching causes a more serious slowdown for complex observations implemented as scripts, which require numerous interactions between the target program and the debugger before reporting a result back to the user.

2.2 Event-based Observation

Interactive debugging is not the only way to perform observations on a program. A different model for observing the behaviour of a program is based on the notion of an event. Conceptually, if a program’s execution is thought of as a sequence of state transitions, an event will identify a point in time at the beginning or at the end of a state transition. Events are used to select points in time where the program’s state will be observed.

(Note that the term ‘event’ as it is used in descriptions of existing debugging systems conflates the notions of an action and an occurrence. An event in the sense of an action, such as a function call, can have an arbitrarily long duration, and therefore cannot be used to unambiguously specify a point in time when it is appropriate to collect information about the event. On the other hand, an event in the sense of an occurrence is associated with
one particular point in time when some action has been completed. Existing event tracing systems specify well-defined points in time when it is appropriate to gather information about each event, which indicates that the term ‘event’ in descriptions of such systems is used to signify an occurrence rather than an action. For the sake of precision, in this thesis the term ‘state transition’ will be used to refer to actions that have a duration, and the term ‘event’ will be used solely to refer to occurrences that are located at a specific point in time.)

An event can have associated context data taken from the program’s state, as well as from additional sources of information such as the program’s original source code or the state of the operating system environment at the time of the event. The developer can request to perform an observation on a set of events in some target program, and collect context data associated with each event. The most basic state transition performed by a program is the execution of one machine instruction. However, larger-scale occurrences in the program can be thought of as state transitions. Events are usually identified in terms of state transitions corresponding to elements from the original program’s source code, such as statements or function calls. In contrast to an interactive debugger, the use of events as a basic abstraction frees the developer from having to describe the low-level operations required to capture information from each event.

This event-based model has been used in the design of tracing and dynamic-instrumentation frameworks intended for analyzing the behaviour of programs with minimal disruption to their execution [37]. Examples of such frameworks include DTrace [4], SystemTap [19], and LTTng [38].

The location in the program’s code at which an event can be captured is typically known as a tracepoint. Tracepoints can be added to a program either statically during compilation, or dynamically when a program has already been compiled.

Static tracepoints are included in the program’s code, and are typically used for frequently observed events. In order to define a static tracepoint, the developer must annotate the program’s source code with an explicit directive and recompile the program [39]. Figure 2.1 shows a fragment of an example program that includes a static tracepoint directive for use with DTrace. In this program, the DTRACE_PROBE2(...) directive will be compiled into a static tracepoint named myserv::query__receive, with the two values query->clientname and query->msg made available to any tracing framework that captures the corresponding event.

Dynamic tracepoints can be added to an already-compiled program during its execution. This allows the developer to avoid recompiling the program each time they need to add a new tracepoint. Dynamic tracepoints are implemented using a mechanism similar to breakpoints in an interactive debugger. In contrast to a debugger breakpoint, which pauses the execution for an indefinite period of time to allow the developer to interact with the program state, a tracing framework responds to a dynamic tracepoint by automatically capturing context data and
immediately resuming the program’s execution.

Tracing frameworks differ in terms of how the developer describes the selection of events to observe and in terms of how context data from each event is processed and presented to the developer.

LTTng runs continuously as a system service and can be controlled using a set of command line tools which enable and disable tracepoints within a target program. The LTTng service captures a set of attributes from each event that occurs at an enabled tracepoint. The developer can specify a filter expression to select a subset of the captured events to be recorded. The filter expression is a boolean predicate over attributes of the event, specified using a basic subset of the C expression syntax [40]. Events whose attributes match the filter expression are recorded to a trace file which the developer can later examine using separate tools.

LTTng is designed to allow the developer to study the behaviour of a running program over long periods of time. For this purpose, recording events to a trace file is a useful model, as it separates the process of analyzing the collected context data from the process of running the program and observing events.

DTrace and SystemTap have designs that aim for a greater degree of programmability. Both frameworks provide a very similar model for the developer to describe their observations, by incorporating a scripting language that allows the developer to specify how the framework should react to each observed event.

In contrast to the procedural scripting interfaces of interactive debuggers (which require the developer to provide a step-by-step description of how to control the program’s execution and collect data), scripts in DTrace and SystemTap consist of a list of probe declarations [41, 42]. A probe declaration consists of a reference to an event and a handler block containing statements to execute whenever the event occurs. Events are selected out of a large library of high-level events provided by the framework, which includes tracepoints as well as higher-level events which combine a tracepoint with additional processing code to capture less easily-accessible context data. (Higher-level events are used to simplify the process of observing complex events within the operating system such as system call invocations or networking events.) The statements inside a handler can output information to the developer, as well as record information in data structures which can be accessed by handlers for later events. This description model makes it particularly simple to collect statistics about a large number of events and generate a summary report at a later point in time, allowing the framework to be adapted to performing a wide range of observations [7, 43].

Scripts in the DTrace language are translated to a bytecode format and executed by an interpreter embedded within the Solaris or Darwin kernel. SystemTap incorporates a translator program that compiles the script into a native Linux kernel module. Because of their reliance on kernel internals for observation, both frameworks are tied to particular operating systems: DTrace supports the Solaris and Darwin kernels, while SystemTap supports the Linux kernel.

Unlike the scripting interface of an interactive debugger, the event-based languages used in SystemTap and DTrace remain conceptually simple even as their capabilities are increased by extending their libraries of built-in events. The richer the set of events that a given tracing framework can observe, the more flexibility there is to write custom tracing and profiling scripts adapted to a wide range of situations. The goal of these tracing frameworks is to replace a wide variety of more specialized tools with a single framework that the developer can adapt to the needs of any given situation. Thus, an event-based tracing language can be thought of as another attempt at realizing the ideal of programmable debugging.

While the language models of SystemTap and DTrace are potentially capable of being used to describe a wide range of observations, the implementations of these frameworks are limited in practice by restrictions arising from the target application of both frameworks. Rather than software developers, these tracing frameworks are mainly targeted at system administrators who need a means to observe the behaviour of software on a production
system. This target application requires an observation tool which is guaranteed not to disrupt the performance or behaviour of the program being observed. In order to provide this guarantee, each observation mechanism in these tracing frameworks triggers only a limited amount of data collection and processing before immediately returning control to the program. In addition, the frequency with which observation mechanisms can be triggered must be restricted. Static tracepoints can only be placed at a limited, explicitly specified subset of locations in a program. Dynamic tracepoints could be placed at a larger set of program locations; however, they are only efficient when placed infrequently within the program, and can cause visible and even prohibitive performance impacts if triggered too frequently during a program’s execution.

The prohibitive performance impact imposed by the tracepoint mechanism sets a practical limit on the scenarios where existing tracing frameworks can be used. For example, it is not possible to perform observations such as origin tracking, which require capturing every memory operation performed by the program. These limitations are inherent to any framework based on the tracepoint mechanism, and prevent the event-based language model built into these frameworks from being used to its full potential. These limitations can only be overcome by using a different observation mechanism.

2.3 Heavyweight Instrumentation

Some of the errors and incorrect states in a program can be classified into well-defined categories. Such categories of errors include memory errors (such as illegal reads and the use of uninitialized data in C/C++), concurrency errors (such as data races in multithreaded code) and invalid operations (such as arithmetic overflows). Many of these common error categories can only be detected by sampling the program state frequently and at numerous potential locations. For example, the runtime environment of a C program does not detect or prevent the use of uninitialized data, making it possible for a program to exhibit unexpected and incorrect behaviour when an uninitialized memory location turns out to contain unexpected ‘garbage data’ left over from a time when the same memory location was occupied by a different data structure. In order to accurately track which memory locations contain uninitialized data, it is necessary to observe all memory operations throughout the program.

Such pervasive observations can be performed using the techniques of instrumentation and shadow memory. Instrumentation is a technique where a program is modified on an instruction-by-instruction basis in such a way that it contains the code needed to perform an observation and executes that code without the transfers of control that would be required by the use of breakpoints or dynamic tracepoints. A shadow memory is a data structure that stores a shadow value for every location in the target program’s memory [16, 25]. A shadow value can be used to store information about the corresponding memory location, such as whether the location contains valid data or which threads are known to access the location [44]. Shadow memory can also be used to record when the data stored in each memory location was originally written, a technique known as origin tracking [22]. Origin tracking can help to identify the location in the program responsible for producing invalid data that subsequently caused an error.

There are a number of heavyweight instrumentation frameworks that combine instrumentation with shadow memory to perform complex observations on a program. An instrumentation framework provides functionality to control the execution of a target program, to add instrumentation to its code, and to create and access shadow memory data within the target program’s address space. The particular instrumentation to be performed by the framework is specified by a plugin written in a conventional programming language such as C. Each instrumentation framework includes a number of prebuilt plugins for detecting common patterns of error behaviour. If the developer suspects that a problem is caused by an underlying error that falls into one of the common categories,
they can run the framework with the corresponding prebuilt plugin against an execution of the program in order to collect information about all occurrences of the suspected error.

Two commonly used heavyweight instrumentation frameworks are Valgrind [17] and DynamoRIO [3]. Valgrind includes the widely used Memcheck memory error detection tool and the Helgrind concurrency error detection tool, as well as various other plugins [17, 21]. Due to the fact that memory errors cause a significant proportion of the problems found in software written in C and C++, Valgrind and its Memcheck plugin in particular have proven to be extremely popular across a wide range of projects written in those languages [45].

Both Valgrind and DynamoRIO follow a dynamic instrumentation model in which instrumentation is done on an already-compiled executable.

The approach to instrumentation used in Valgrind aims to be portable across many machine architectures. The Valgrind framework translates the target program’s machine code into an abstract, machine-independent intermediate representation. This intermediate representation is divided into basic blocks\(^1\) and instrumented one basic block at a time. To instrument a basic block, the framework invokes a callback function within the Valgrind plugin. The callback function modifies the contents of the basic block using an instruction manipulation library provided by Valgrind. The instrumented basic blocks are translated back into machine code and executed by a just-in-time compiler built into Valgrind. The instrumented target program executes with performance overheads from the just-in-time compiler as well as performance overheads from executing the additional instrumentation code. Just-in-time compilation with a machine-independent intermediate representation slows the target program’s execution down by a factor of about 3 to 8 times [17], but makes it possible for Valgrind and its plugins to operate on a wide range of processor architectures [46].

In comparison to Valgrind, DynamoRIO’s architecture emphasizes performance rather than portability, and does not use an abstract intermediate representation or just-in-time compiler. DynamoRIO plugins operate directly on the target program’s machine code. In other respects, DynamoRIO instruments the target program using a process similar to Valgrind. The target program’s machine code is divided into basic blocks. To instrument each basic block, the framework invokes a callback function within the DynamoRIO plugin. The callback function modifies the machine code instructions contained within the basic block, using an instruction manipulation library provided by DynamoRIO. Once a basic block has been instrumented, it is placed in a code cache managed by the DynamoRIO framework [3] and then executed directly by the hardware processor.

Both frameworks store instrumented basic blocks within a data structure called a code cache (in DynamoRIO) or translation cache (in Valgrind). Any branch instruction within the code cache that leads to an uninstrumented basic block is replaced with an instruction that transfers control to the instrumentation framework, which instruments the uninstrumented basic block, adds it to the code cache, and updates the branch instruction to point to the instrumented basic block. Instrumented code within the code cache can be executed without involvement from the framework as long as it does not branch to code within the target program that has not yet been instrumented. Branch instructions that lead outside the code cache are intercepted by the framework and trigger instrumentation of the next portion of the program to be performed. Once the majority of the program has been instrumented and placed in the code cache, the instrumented program will be executed with only occasional interruptions from the instrumentation framework.

Instrumentation can also be added to a program during compilation, in an approach known as static instrumentation.

The GCC and LLVM compilers can be used as static instrumentation frameworks. When a program is compiled,\(^1\) a basic block is the simplest unit of control-flow in a program. Typically, a basic block is a straight-line sequence of instructions, with no incoming branches except to the first instruction of the basic block, and no outgoing branches except from the very last instruction. Valgrind’s definition of basic blocks varies from the norm, allowing some outgoing branches from the interior of the basic block.
CHAPTER 2. BACKGROUND AND RESEARCH PROBLEM

When using the static instrumentation model, a separate version of the program executable must be compiled in order to add instrumentation. This is generally considered less convenient than using dynamic instrumentation to carry out observations on an already-compiled program. However, the ability to create an executable with built-in instrumentation is useful for heavily environment-dependent software (for example, desktop applications) which interact in a complex manner with the surrounding operating system and hardware. (For example, the automated build system of the Chromium web browser project produces a version of each executable that includes AddressSanitizer instrumentation [49].) If the occurrence of a problem depends on details of the surrounding environment that cannot be replicated on the developer’s machine, an instrumented executable can be placed in the environment where the problem occurs. The information collected from the instrumented executable could aid in fixing the problem, or be used to reproduce it in a different environment where other debugging tools are available.

As described above, the prebuilt tools provided by instrumentation frameworks are useful for debugging when the developer suspects that a problem is caused by a commonly occurring category of error. However, many debugging scenarios require the developer to define and search for error conditions that are specific to a particular program. For example, the developer may want to collect information about a set of events specific to the program (for example, operations on a specific data structure) while using origin tracking to find the locations in the program that wrote values accessed during each event.

To describe such observations as plugins for a heavyweight instrumentation framework, the developer must specify an instruction-by-instruction transformation of their program. The effort required to do so is nontrivial. The program to be observed is written in a high-level language, whereas the instrumentation framework operates on a representation of the low-level compiled machine code for the program. The developer must therefore be aware of how high-level concepts of the original program translate to the machine-code level. In general, when detecting errors that are unique to a particular program, the effort required to develop a customized plugin for a heavyweight instrumentation framework is not justified.

Thus, while extremely powerful and flexible in potential, heavyweight instrumentation frameworks are only applied in practice when there is an already-existing plugin that performs the exact observation needed by the developer.

2.4 Basic Insight

Conventional interactive debuggers do not provide an efficient means to perform complex observations which require information to be collected from events occurring at numerous points in time. Without any automation, such complex observations can require hours of programmer effort. The desire to find an efficient means to perform complex observations has led to the development of a variety of tools for programmable debugging, which aim to provide a means to express a complex observation as a program, eliminating the need to carry out individual steps of an observation through an interactive interface.

Heavyweight instrumentation frameworks can be seen as a powerful facility for programmable debugging,
which allows extremely frequent observation of the events and states within a program. However, the development of plugins for heavyweight instrumentation frameworks is a difficult and low-level task, limiting the use of heavyweight instrumentation to prebuilt plugins that detect a few commonly occurring categories of errors.

The DTrace and SystemTap tracing frameworks introduced a new approach which allows a large variety of observations to be expressed compactly. These frameworks are built around an event-based language which expresses observations in terms of high-level events. However, the designs of the DTrace and SystemTap languages do not provide sufficient flexibility to describe all combinations of events that could be required for expressing a detailed observation of a program for the purpose of debugging.

The purpose of this research is to extend and generalize the event-based language model and to develop an event-based language that can be used to debug complex software problems. This is the first project which uses heavyweight instrumentation to systematically overcome the limitations preventing existing event-based tracing languages from being applied to complex debugging tasks.

This work introduces the E language, an event-based language for implementing observations on a target program. E allows the programmer to identify events in an execution in terms of the conceptual model of the target program’s high-level language. This model is based on the state transitions and events occurring in the target program execution rather than the low-level details of the observation mechanism used to observe the target program. Because of this, E allows observations to be described independently of the underlying observation mechanism. In particular, the E language can be implemented on top of heavyweight instrumentation frameworks, which provide significantly more powerful observation capabilities than conventional debuggers or tracing frameworks.
Chapter 3

Basic Concepts of Event-Based Tracing

This chapter introduces the basic concepts of the E language. The E language is designed for the purpose of implementing observations on a target program. The language is built around the concept of an event. Observations are defined in terms of events occurring in the target program’s execution, as well as actions to be performed in response to different events. The design of E is inspired by the event concept introduced by the DTrace and SystemTap languages\(^1\). E introduces a model of the target program’s execution based on nested state transitions. This model includes a flexible mechanism for selecting event occurrences relevant to the debugging task as well as a scoping scheme for accessing context data from nested state transitions.

3.1 State Transitions, Events, and Probes

As discussed in Section 2.2, we regard a program’s execution as a sequence of state transitions. The most basic state transition that can be observed in a target program is the execution of a single machine instruction. Larger-scale occurrences in the program can also be thought of as state transitions. For example, an entire function call can be thought of as a state transition between a program state where the function is about to be called and a program state where the function has just finished executing.

State transitions form the basic vocabulary in terms of which a programmer using the E language analyzes a program’s behaviour. Based on the semantics of the target program, we distinguish different types of state transitions such as individual instructions, statements from the original program source, and function calls.

Each occurrence of a state transition begins at some point in time and ends at some point in time. We call each of these points an event occurrence in the target program’s execution. The E language provides predefined events which refer to the endpoints of certain state transitions. During any execution of the target program, a state transition of a given type occurs many times, resulting in many occurrences of the corresponding predefined events.

Each occurrence of an event has a corresponding program state. This program state can be thought of as the content of the program’s memory and registers at the moment when the event occurs. For each event, the E language provides associated program state values computed from the program state at the event’s occurrence, as well as context values containing information about the broader context within which the corresponding state transition occurs. For example, an event corresponding to the beginning of a function call will define a context

\(^1\)§8 contains a detailed comparison of the designs of the E and SystemTap languages.
value containing the name of the function, as well as program state values containing parameters given to the function.

In the previous chapter, we used the term ‘observation’ informally. In our discussion of the E language’s capabilities, we will define an observation to be the process of interrupting the program at a specified set of event occurrences and collecting information in the form of program state values and context values.

It is important to note that each particular execution of the target program contains many event occurrences of different types. Not all of these occurrences may be relevant to the debugging task. In order to perform an observation, the programmer needs to collect information about some subset of these event occurrences. The E language provides functionality to select a subset of event occurrences in the target program’s execution and a means to collect information relevant to the debugging task at each event occurrence.

In the E language, a set of event occurrences can be identified using an event expression (described in §5.5). An event expression is combined with a handler block containing a list of imperative statements to form a probe directive (described in §3.5). Probe directives allow the programmer to define actions that collect information in response to each event. We will call an E program consisting of these elements an observation program.

### 3.2 Predefined Events

Examples of predefined events supported by the E language include:

- `instruction.before` and `instruction.after`\(^2\) identify, respectively, the point immediately before and the point immediately after the execution of each instruction in the target program.
- `stmt.before` and `stmt.after` identify the points immediately before and immediately after the execution of each line of the target program’s original source code. Because debugging information identifies locations in the target program’s source code in terms of line numbers rather than syntactic statements, this does not correspond precisely to statements in the target program’s high-level semantics.
- `program.before` and `program.after` identify, respectively, the point at the very beginning of the target program’s execution and the point immediately after the execution terminates.

\(^2\)These events can also be referred to by their more compact alternate names `insn.before` and `insn.after`. 
CHAPTER 3. BASIC CONCEPTS OF EVENT-BASED TRACING

These events provide context values which contain information about the corresponding state transition:

- `instruction.before` and `instruction.after` both provide an opcode context value identifying the opcode of the instruction, as well as an addr context value identifying the address of the instruction in the target program’s machine code.

- `line.before` and `line.after` both provide a context value loc which contains the name of the source file `loc.file` as well as a line number in the file `loc.line_no`.

Large-scale state transitions during a program’s execution can be considered as indivisible actions, or as actions composed of smaller-scale state transitions. For example, a function call is divided into a sequence of three smaller-scale state transitions: the transfer of control to the function that is being called (the callee function), the execution of the callee function’s body, and the return of control from the callee to the calling function.

During each of these state transitions, a specific modification is made to the program’s memory. In particular, the transfer of control to the callee function results in a stack frame being created to store the callee’s local variables, while the return of control to the caller results in the stack frame being destroyed and the callee’s return value being passed to the caller. These state transitions are part of the target program’s calling convention and occur during all function calls.

Based on this analysis, we consider the execution of the function’s body as a separate state transition and identify four events that can be observed relative to a single function call:

- `fn.call` occurs within the calling function, immediately before the entire function call.
- `fn.entry` occurs immediately before the callee’s body executes (after the transfer of control to the callee).
- `fn.exit` occurs immediately after the callee’s body executes (before the return of control to the caller).
- `fn.return` occurs within the calling function, immediately after the entire function call.

The transfer of control to the callee happens between `fn.call` and `fn.entry`, the callee’s body executes between `fn.entry` and `fn.exit`, and the return of control to the caller happens between `fn.exit` and `fn.return`.

All four events provide the context value `name`, the name of the function being called. In addition, some of the events provide program state values:

- `fn.call` and `fn.entry` provide program state values corresponding to parameters of the function, identified by prefixing the name of the parameter (from the original program’s source code) with a “$” symbol. For example, if the function being called has a parameter “`param`”, the corresponding occurrence of the `fn.call` event will provide a program state value named `$param`.

- `fn.exit` and `fn.return` provide `return_val`, the return value of the function.

It is often useful to examine how the program reads and modifies its data structures. Nontrivial debugging problems often involve searching for incorrect interactions between different parts of the program that access the same data structure. Thus, we may be interested in examining all state transitions that involve modifications to a particular area of memory (which we will call a memory object). Such state transitions relate to the lifecycle of the memory object: its initial allocation, modifications and accesses to the object’s contents, and the eventual deallocation of the object. E includes a set of predefined events for this purpose:
• **obj.alloc** occurs immediately after a memory object is allocated by the program. This event provides program state values `addr` and `size`, the location and size of the memory object being allocated.

• **obj.free** occurs immediately before a memory object is deallocated. It provides a program state value `addr`, the address of the object.

• **obj.access** occurs immediately after the target program accesses a memory location (whether to read or to write it). This event provides a program state value `addr`, the address in memory which was read (or written).

### 3.3 Event Expressions and Filtering Conditions

Each predefined event occurs many times in any execution and a large proportion of these occurrences are likely to be irrelevant to the debugging or observation task. By default, when a predefined event is used in a probe directive, the corresponding handler block will trigger on every occurrence of that event. In order to restrict a handler block to triggering only on a subset of event occurrences, the E language supports combining a predefined event with filtering conditions to form an event expression.

A filtering condition is a boolean expression computed over the context values and program state values provided by the event. The handler for an event with a list of filtering conditions triggers when the values provided by the event satisfy all of the filtering conditions in the list.

For example, the event expression

```
instruction.before: opcode == "div" || opcode == "idiv"
```

includes a filtering condition that causes the handler for an `instruction.before` event to trigger immediately before any integer division instruction in the program. ("div" and "idiv" are opcodes corresponding to integer division instructions.)

The event expression

```
fn.call: name == "foo", $flags == 0
```

combines an `fn.call` event with a list consisting of two filtering conditions. The corresponding event handler will trigger at any call to a function named `foo` whose parameter `flags` is equal to 0.

### 3.4 Nested State Transitions

A state transition of one type can occur during a larger-scale state transition of a different type. For example, every instruction in the target program is executed during some function call. In this case, we say that the inner instruction state transition is nested within the outer function call state transition. A state transition can be nested within several state transitions of the same type: for example, an instruction occurs within a call to some function, which was called from another function. The instruction is therefore nested within every function call on the program’s current call stack. Whenever there are several outer state transitions of the same type, one of them will be the innermost.

It is useful to have a mechanism for requesting context variables of an outer state transition from a filtering condition or handler block observing an inner state transition. Unlike program state values, context values encode information about the context of a state transition which is available at any point during the state transition, rather than only at its beginning or end.
Figure 3.2: Events which can access the context of a state transition. Context values for the function call state transition can be accessed from the start and end points of the state transition, as well as from an event nested within the function call. The context of a state transition is thus considered to exist throughout its entire duration.

Figure 3.3: Nested state transitions whose context can be accessed from one event. A handler for an event occurrence before the instruction state transition can access context values from the innermost containing statement and function call using the corresponding contexts stmt and fn. (Contexts marked * will be added in a future version of the language to provide access to additional outer state transitions.)

To provide access to context variables from outer state transitions, the E language includes a set of special contexts corresponding to each type of state transition. A context is a data structure containing context values from an outer state transition. A value from a context is referenced by prefixing its name with the name of the context corresponding to the state transition and the '.' operator; for example, context value name in context fn is referenced as fn.name.

The following state transitions have context variables that can be accessed using a context:

- A function call state transition has a context variable name, which contains the name of the function being called. This context variable can be accessed using the context fn.

- A statement state transition has a context variable file identifying the filename and a context variable line_no identifying the line number of the statement’s location in the original source code of the target program. These context variables can be accessed using the context stmt.

3 Although there are future plans to support access to information from all nested state transitions containing an event, contexts in the current version of the language only support access to information from the innermost level of nesting – for example, the fn context is accessible but the fn.caller context is not.
For example, the event expression

```
instruction.before: opcode == "div" || opcode == "idiv",
        fn.name == "foo"
```

uses the `fn` context and `instruction.before` predefined event to cause the corresponding handler block to trigger only on integer division instructions occurring within the body of a function named `foo`.

The event expression

```
fn.exit: stmt.file == "dictobject.c"
```

will cause the handler to trigger at the exit point of any function in the file `dictobject.c`.

Because checking for a particular file and line number is a common use of filtering conditions, the E language provides an `is` operator, used in expressions of the form `stmt is <file>:<line_no>`, as an alternative convenience syntax for writing comparison expressions such as `stmt.file == <file> && stmt.line_no == <line_no>`. For example, the expression `stmt is dictobject.c:*` is equivalent to `stmt.file == "dictobject.c"` (the `*` symbol is a wildcard which matches any line number). Similarly, `stmt is file.c:(15,16)` would be equivalent to

```
stmt.file == "file.c" && (stmt.line_no == 15 || stmt.line_no == 16)
```

Thus, the event expression

```
instruction.before: stmt is myprogram:250
```

uses the `stmt` context to identify all instructions corresponding to line 250 of the target program’s original source code.

### 3.5 Probe Directives and Handler Blocks

In order to define actions that will be performed in response to each event, the programmer must combine an event expression with a `handler block` to form a `probe directive`.

A handler block is a list of imperative statements, which are used to specify an action to perform in response to each event occurrence matching the probe directive’s event expression.

The E language supports variable assignment statements, function calls, and standard control flow constructs such as `if` statements as well as `for`, `foreach` and `while` loops.

The statements in a handler block have access to context values and program state values of the event occurrence which triggered the handler, as well as globally defined variables. For example, the probe directive

```
probe fn.call: name == "foo" {
    log("foo(%s) was called", $param)
    counter++
}
```

describes an observation on calls to the function `foo`. Whenever the function `foo` (with parameter `param`) is called, the handler block increments the global variable `counter` (used to track the total number of calls) and prints a message containing the parameter of the function call.
3.6 Function Declarations

In addition to the probe directives described previously, E supports function declarations. Function declarations define subroutines that can be called from within handler blocks.

For example, the function

```c
func consistent(me) {
    me_key = ((PyDictEntry)me)->me_key
    me_value = ((PyDictEntry)me)->me_value
    return me_key != 0 && me_value == 0
}
```

computes a boolean condition on fields from the target program data structure `me`. Within the function’s body, expressions such as `((PyDictEntry)me)->me_key` are used to extract fields of the data structure: the integer value `me` is treated as an address within the target program which contains a data structure of type `PyDictEntry`. The function can be called within any handler block. For example, the following probe directive checks the consistency of parameter `ep` in any call to function `build_indices`:

```c
probe fn.entry: name == "build_indices" {
    if (!consistent($ep)) log("not consistent!"))
}
```

3.7 Type System and Global Data Declarations

Data types supported by E include values of unsigned integer or string type, records and associative arrays. Unsigned integers are used to represent both numbers as well as addresses within the target program’s memory. String values are character strings represented using ASCII. Records are data structures containing named subfields. Different subfields of a record may be of different types.

Associative arrays map a set of integer or string indices to a set of entries. Entries can be values of any type, including array or record type. Associative arrays are homogeneous: all keys of an array must be of the same type, and all entries of an array must be of the same type.

Associative arrays can be used to collect information about different elements of the target program. For example, the following code

```c
array called

probe fn.entry: loc is dictobject.c:* {
    called[name]++
}
```

consists of a data declaration defining an associative array, and a probe directive which uses the array to store a running count of function calls for each of the functions in `dictobject.c`.

E supports global and local variables. Global variables must be explicitly declared by global data declarations, which define variables that can be accessed by any handler block in the E program. Global variables allow the program to combine and summarize information from multiple event occurrences. E also supports local variables within probe handlers and functions: any variables that are referenced within a block and are not declared as global variables are implicitly treated as local variables.
E is a statically typed language: the type of every variable is fixed at translation time, with the E translator performing type inference to determine the types of variables, records, and arrays, as well as the set of subfields present in each record and the set of local variables present in each probe handler or function, based on how these variables and subfields are used throughout the program. The presence of type inference in the E translator keeps the programmer from having to declare the type of each variable explicitly, simplifying the process of writing programs in E.

3.8 Shadow Arrays

In addition to the above data types, E supports shadow arrays, a subtype of associative arrays where the array is indexed by target program addresses and can store entries of any type. Target program addresses are represented using integer values. Unlike other associative arrays, whose internal representation is based on hash tables, the internal representation of a shadow array is based on a shadow memory data structure provided by a heavyweight instrumentation framework.

The entries in a shadow array can be used to efficiently store information about data at different locations in the target program’s memory. In the following example, a shadow array is declared and used to observe and count accesses to a subset of the locations in the program’s memory:

```
shadow watched

probe fn.return: name == "f" {
    watched[return_val].access_count = 0
}

probe obj.access: addr in watched {
    watched[addr].access_count++
}
```

We assume that f is the name of a function in the target program which returns an address each time it is called. Whenever a call to function f occurs, the handler block for fn.return adds the address returned by f as a key in the watched array, initializing the access_count subfield of the corresponding entry to zero. The probe directive for obj.access is triggered whenever the address of the observed memory access exists as a key in the watched array, incrementing the access count in the corresponding entry.

3.9 Structure and Execution of an E Program

An E program consists of a list of declarations and probes. When the E program is run, each occurrence of an event matching one or more of the probe directives will trigger the corresponding handler block or blocks in a sequence which matches the order of the probe directives in the program. Statements in handler blocks can store information in global variables and data structures or produce output by calling predefined functions such as log and printf.

For example, the program fcall-counts.ebt (in Figure 3.4) counts the number of times each function in a particular file has been called. It consists of two probe directives. One probe collects a count of calls to different functions in the file dictobject.c, while the second probe outputs a summary of the collected data after the completion of the target program’s execution.
An E program can be invoked from the Linux command line with the `ebt` command:

```
$ ebt fcall-counts.ebt -- myprogram
```

The above command launches `myprogram` as the target program, and perform the observation specified in `fcall-counts.ebt`. By default, the output of an E program is added to the standard output of the target program being observed. However, the output can also be redirected to a separate log file for later analysis by specifying the `--redirect` command line option.
Chapter 4

Example Debugging Scenario

This chapter illustrates the use of the E language. For our illustration, we have selected a real-world problem reported in Red Hat Enterprise Linux (RHEL). A customer reported to the Red Hat support staff that the RHEL operating system installer program was crashing on startup. The support staff at Red Hat found that this problem occurred only on a specific hardware architecture, but was consistently reproducible on that architecture. The crash report was urgently escalated to the development team with a request to identify and fix the underlying problem. The process of diagnosing the problem was not straightforward; indeed, it was sufficiently complex that it was used in a presentation at the PyCon 2011 conference to illustrate the use of advanced features of GDB [11].

We will first describe how the underlying cause of the crash was diagnosed by Red Hat developers using GDB. Then, we will describe a strategy for diagnosing the same problem using E, illustrating how event-based tracing could have been used to diagnose the same problem with a significantly reduced amount of manual work.

4.1 Original Debugging Process

(The full details of the crash report and debugging process are recorded on a private Red Hat bug tracker. Thus, the debugging process described here is a reconstruction based on the details available in the presentation, and may differ from the real debugging process in minor details.)

The crashing RHEL installer program was written in Python. The Python interpreter, a program written in C, was terminating abruptly with an ‘assertion failure’ message, rather than cleanly exiting after reporting an error within the Python program it was running. Based on this fact, it was concluded that the problem was not contained in the installer program itself, but rather originated somewhere within the Python language interpreter. GDB was used to inspect the interpreter’s state at the point of the crash, and it was found that data structures being used to represent Python’s “dictionary” datatype contained corrupted data.

Within the Python interpreter, dictionaries are represented as hash tables based on an open addressing scheme with lazy deletion\(^1\). The following C data structures are used to represent a dictionary object within the interpreter’s source code [50]:

\(^1\)Open addressing is a hash table representation in which each slot of the table stores at most one value. In the event of a hash collision, one of the colliding values is stored in an alternate table slot selected according to a formula called a probe sequence. An empty slot signifies that there are no elements matching the corresponding hash key. Therefore, when an element is deleted, its slot must be marked with a special ‘dummy’ value to indicate that there may be an additional element at the next slot in the probe sequence. This technique is known as lazy deletion.
A PyDictObject contains an `ma_table`, which stores an array of dictionary entries of type `PyDictEntry`. According to a basic invariant documented in the Python interpreter’s source code [50], the fields in a `PyDictEntry` must satisfy one of the following states:

- **Unused**: in this case, `me_key == me_value == NULL`. This represents an empty slot in the hash table.
- **Active**: in this case, `me_key != NULL` and `me_value != NULL`. This state represents a hash table slot which is storing a value.
- **Dummy**: in this case, `me_key == -1` (a “dummy” value). This state is used to mark a hash table slot which stored a value that was subsequently deleted from the table.

Inspecting the contents of an `ma_table` array in the crashing program showed that some of the `PyDictEntry`es stored in the `ma_table` had a nonzero, non-dummy numerical value in their `me_key` and a NULL pointer in their `me_value`. Since this situation does not correspond to any of the three permitted states for a `PyDictEntry`, it is clearly the result of data corruption. Encountering the NULL pointer stored in one of these
corrupted me_values was the immediate cause of the crash. (The presence of corrupted data was caught by an assertion. Even without the assertion, the program would have crashed at the same location in the code, since the next operation after the assertion is a pointer dereference operation.)

Figure 4.1 shows an example of an ma_table array containing both valid and corrupted entries.

The next step in the diagnosis process was to determine when a corrupted PyDictEntry first appeared in an ma_table, and which operation in the interpreter caused the data corruption.

In order to scan all entries of an ma_table and check them for corruption, the following script was written using GDB’s built-in Python support:

```python
val3 = gdb.parse_and_eval('((PyDictObject*)interned)->ma_table')
```

```python
table_size = long(gdb.parse_and_eval('((PyDictObject*)interned)->ma_mask')) + 1
```

```python
print [i for i in range(table_size) if long(val3[i]['me_key']) != 0 and long(val3[i]['me_key']) != -1 and long(val3[i]['me_value']) == 0]
```

The first command in the script selects an ma_table visible at the current location in the program. The expression used to identify the ma_table must be varied depending on the point in the execution being examined, since the C language permits aliasing and therefore a reference to an ma_table object may be stored under different variable names at different points in the program. For example, the same ma_table object could be referenced by the expression mp->ma_table within the function PyDict_New(), by the expression d->ma_table within the function dict_new(), or by newtable within the dictresize() function.

The second command iterates through the PyDictEntry objects in the ma_table and reports the indices of any entries violating the basic invariant.

A tedious “bisection search” process was required to narrow down the source of the problem, according to the following strategy:

- Select a point in the execution at which to interrupt the program using a breakpoint.
- Use the above script to examine the contents of an ma_table visible at the selected point in the execution and determine if it contains corrupted indices.
- If the table is ‘clean’ (not yet corrupted), use the debugger to advance to a slightly later point in the execution where the same PyDictObject is visible. Check the table again and repeat the process.
- Once the table is found to be corrupted, it is necessary to backtrack to an earlier point in the execution. This requires starting a repeated run of the execution and identifying a breakpoint that will trigger sometime between the last-known point when the table was ‘clean’ and the first-known point when the table was corrupted.

Such manual trial-and-error strategies tend to be required when using conventional debuggers to diagnose complex problems. In cases where the target program is nondeterministic (for example, due to multithreading)
and different instances of the data structure may become corrupted during different executions, even a bisection search strategy may become impractical to apply.

Because breakpoints are identified by giving points within the program’s source code rather than points in time during an execution, the bisection search process required a lot of manual work to carefully identify the correct breakpoints to place in order to visit various points in time during the execution. At the end of the process, it was determined that the contents of a table were corrupted during a dictresize operation. That is, immediately before the dictresize operation the table did not contain corrupted entries, and after being resized the table contained corrupted entries.

After using GDB to step through the implementation of dictresize and examine its behaviour in greater detail, it was discovered that the cause of the corruption was incorrect behaviour by a memset() operation intended to zero out a newly allocated ma_table array (setting all fields in all entries to NULL). The memset() operation left a portion of the requested memory region unmodified, causing the me_key field of one of the ma_table entries to contain garbage data from an unrelated object that had been deallocated earlier during the execution. This could be determined by comparing the contents of the newly allocated ma_table immediately before and immediately after the memset() operation.

Thus, the underlying problem was discovered to originate in an implementation of memset. Since the GNU C library included in Red Hat Enterprise Linux used a separate implementation of memset for each hardware architecture, this also explained why the interpreter crash only occurred on one architecture.

While GDB was eventually effective in diagnosing the problem, it required a lot of manual work in terms of setting breakpoints and stepping through the program’s execution.

### 4.2 Debugging the Same Problem with E

Now we outline how to use the E language and its shadow memory datatype to diagnose the same problem using a single observation. We will write a script that captures the origins of corrupted data in any ma_table in the program.

Our strategy will be to capture origin information for every entry of every ma_table in the program, in order to guarantee that we have origin information for the particular entry that turns out to be corrupted later on. In order to do so, our script (given in Figure 4.2) includes the following components:

- On lines 1-2, we declare two associative arrays watched and tables. The watched array is a shadow memory; conceptually, it is an associative array indexed by address. For each address a in watched, we track two things: watched[a].start_addr, the address of the ma_table currently containing a, and watched[a].origin, the origin of the last known write to a.

  The tables array will be used to record appropriate information about each ma_table as a whole (rather than individual memory locations within the table). In this case, we use tables[t].where_created to record the location where a table at address t was originally created.

- The watch_add function (lines 4-9) designates a run of addresses (addr, addr+1,...,addr+size-1) to be watched by adding them as keys to the watched array. The watched[start:end] = ... syntax is a shorthand for assigning the same value to a consecutive run of indices in an array (that is, to indices from start through end-1 inclusive).

- The probe declarations on lines 11-19 capture all of the locations where an ma_table is created by the Python interpreter. The locations are specified in terms of line numbers in the file dictobject.c. The
01 shadow watched
02 array tables
03
04 func watch_add(loc, addr, size) {
05 // Assign a consecutive run of addresses to be watched:
06 watched[addr:addr+size].start_addr = addr
07 watched[addr:addr+size].origin = "unknown"
08 tables[addr].where_created = loc
09 }
10
11 // Capture the ma_table assignment in dictsize():
12 probe line.after: loc is dictobject.c:652
13 { watch_add(loc, &mp->ma_table, sizeof($PyDictEntry) * $newsize) }
14
15 // Capture all uses of INIT_NONZERO_DICT_SLOTS macro:
16 probe line.after: loc is dictobject.c:(264,268,280,966,975)
17 { watch_add(loc, &mp->ma_table, sizeof($mp->ma_smalltable)) }
18 probe line.after: loc is dictobject.c:2407
19 { watch_add(loc, &d->ma_table, sizeof($d->ma_smalltable)) }
20
21 // Record the origin of any address in watched:
22 probe obj.write: addr in watched {
23 watched[addr].origin = fn.name + " " + line.loc
24 }
25
26 // Scan an ma_table and report corrupted values:
27 func check_corruption(addr) {
28 // Get the table at addr:
29 start_addr = watched[addr].start_addr
30 if (!(start_addr in watched)) {
31 log("no record of a table at %x{addr}!"); return
32 }
33
34 // Check the table for corrupted entries:
35 require_debuginfo("dictentry.h") // <- info on PyDictEntry data structure
36 first = true
37 for (ptr = start_addr;
38 ptr < start_addr+size;
39 ptr += sizeof($PyDictEntry)) {
40 me_hash = ((PyDictEntry)ptr)->me_hash
41 me_key = ((PyDictEntry)ptr)->me_key
42 key_loc = watched[&((PyDictEntry)ptr)->me_key].origin
43 me_value = ((PyDictEntry)ptr)->me_value
44 value_loc = watched[&((PyDictEntry)ptr)->me_value].origin
45
46 if (me_key != 0 && me_value == 0) {
47 if (first) { // -- only print a header when there’s corruption.
48 log("table at %x[start_addr], " +
49 "created at %s[tables[addr].where_created]"
50 first = false
51 }
52
53 log("- corrupted entry at %x[ptr], me_hash=%x[me_hash], " +
54 "me_key=%x[me_key] (written at %s[key_loc]), " +
55 "me_value=%x[me_value] (written at %s[value_loc])")
56 }
57 }
58 }
59
60 probe line.after: loc is dictobject.c:526, // in insertdict_by_entry
61 $ep->me_key != NULL && $ep->me_key != -1 {
62 check_corruption($ep->me_value) // -- address of me_value
63 }
64
65 // Also check for corruption whenever an ma_table is resized:
66 probe line.after(loc is dictobject.c:621) {
67 check_corruption($oldtable)
68 }

Figure 4.2: A program in the E language written to track corrupted ma_table objects in the Python interpreter.
line numbers were found by doing a text search on the source code for statements which assign a value to the `ma_table` field of a `PyDictObject`.

For each `ma_table` whose existence we discover, we use `watch_add` to designate the addresses contained in it to be watched. We make use of the following context values provided to the probe handler by E:

- `loc` is the source code location where the event occurred.
- `$mp->ma_table`, `sizeof($mp->ma_smalltable)`, `sizeof($PyDictEntry)`, as well as `$newsize` are all *program state values* obtained from the program itself at the location of the event.

• Once an address has been added to `watched`, the probe on lines 22-24 will capture all write operations which modify that address. For each write operation to some address `addr`, we record the function and line number at which the operation occurred in `watched[addr].origins`. If an address is not present as a key in `watched`, write operations to that address will be ignored. We make use of the following context values:
  - `addr` is the address being written to (provided by the `obj.write` predefined event).
  - `line.loc` and `fn.name` are the source code location and the name of the function where the event occurred, respectively.

• The function `check_corruption()` (on lines 27-58) scans the contents of an `ma_table` and checks each entry for corruption. This part of our script basically performs the same function as the GDB-Python script used by the developers originally diagnosing the problem. However, our version of this functionality also retrieves origin information from `watched` and prints a detailed report describing each corrupted entry in the table.

In order to examine the contents of a data structure, the script uses *typed pointer dereference* operations such as `((PyDictEntry)ptr)->me_hash` (on lines 40-44). These operations make use of type information obtained from the program (explicitly requested on line 35 using the `require_debuginfo` predefined function – see §5.2 for more information). In addition to reading the fields of a `PyDictEntry`, we can also obtain the address of a field using the address-of operator `&`. For example, `&((PyDictEntry)ptr)->me_key` is the address of the `me_key` field of the `PyDictEntry` stored at `ptr`.

In addition, the script uses *string interpolation* to insert data into message strings. For example, the statement `log("no record of a table at %x{addr}!"))` replaces the character sequence `%x{addr}` with the value of the `addr` variable, formatted as a hexadecimal number, and then prints the resulting string. (String interpolations are described in detail in §5.8.1.)

• The probe on lines 60-63 calls `check_corruption()` on the `ma_table` found at the location of the crash. In this case, the crash happens because of an assertion failure in `insertdict_by_entry()`, which checks the basic invariant for an existing entry in the table. We make use of the context value `loc` (the source code location of the event) as well as the program state values `$ep->me_key` and `$ep->me_value`, which are fields of a variable `ep` within the program that is visible at the location of the event (i.e. line 526 of `dictobject.c`).

(Within this probe, it would also be possible to only check the single entry being accessed from `insertdict_by_entry()` rather than all entries in the `ma_table`. However, in order to avoid having to write code that is redundant between this probe and the probe on lines 66-68, this probe also calls `check_corruption()` to check the entire table.)
• It is possible that checking the table at the origin point will not capture the event we are interested in. A Python dictionary has a complex lifecycle which may involve several ma_table objects. Initially, the dictionary is created with a minimal ma_table; whenever the dictionary runs out of space, it is automatically resized by creating a larger ma_table and copying the data from the old table into the new one. It is conceivable that corrupted data is introduced into one ma_table, copied into another one, and only then becomes the cause of the crash. In this case, the origin recorded for the newer table will refer to the point where the dictionary was resized, rather than the point where the corrupted data was first introduced.

The probe on lines 66-68 solves this problem by also checking for corruption immediately before an ma_table is resized (in the dictresize() function). We call check_corruption() on the old version of the table (stored in the program state value $oldtable). This ensures that the original operation which corrupted the table will also be reflected in the script output.

(If an ma_table is resized too many times during the execution and this approach generates too much output to be useful, we could instead have this probe respond to a dictresize() operation by copying origin tracking information about the old ma_table to corresponding entries for the new ma_table.)

After running our origin tracking script on the program, we should observe a message such as the following:

ma_table at 0x045a, first created at dictobject.c
  - corrupted entry at 0x0470, me_hash=0xd33d,
    me_key=0x45 (written at <unknown>),
    me_value=0x0 (written at memset memset.S:90)

In this entry, the me_value was clearly written by a call to memset() (a standard C library function), since the origin point of the me_value is reported to occur in memset.S. On the other hand, the value contained in me_key was already present in the ma_table when we initialized its origin tracking information, since the initial <unknown> value stored in the origin array was not overwritten by any subsequent write operation. Looking at the code used to initialize a new ma_table in the dictresize() function, it is possible to pinpoint the problem:

```c
643     newtable = PyMem_NEW(PyDictEntry, newsize);
644     if (newtable == NULL) {
645         PyErr_NoMemory();
646         return -1;
647     }
648     /* Make the dict empty, using the new table. */
649     assert(newtable != oldtable);
650     mp->ma_table = newtable; // <-- HERE is where we start origin-tracking!
651     mp->ma_mask = newsize - 1;
652     memset(newtable, 0, sizeof(PyDictEntry) * newsize);
653     mp->ma_used = 0;
654     i = mp->ma_fill;
655     mp->ma_fill = 0;
656     ...
```
Since we started origin-tracking for mp->ma_table at line 652, we can conclude that the call to memset at line 654 was not zeroing out the entire table. (If we want to confirm this, we could extend our script with a probe to call check_corruption() after line 654 and re-run the program.) We have thus arrived at the same diagnosis as the original Red Hat developers, through a significantly less tedious debugging process.

Besides the functionality outlined in our basic language design, the above scenario suggests that the following additional functionality would be useful:

- We should also be able to capture a backtrace (which consists of the contents of the program’s call stack) at each origin point, in order to provide more context about the location where the target program is calling memset. (When debugging this particular error, the location quickly becomes obvious without any backtrace, but this is not likely to be the case for every problem.) In order to efficiently assemble and store a backtrace for every origin point, we would need to develop ‘shadow stack’ functionality to store multiple backtraces in a compressed representation similar to the one used by CCTLib [5] or CallGrind [51].

- To further investigate what happened at the origin of the corrupted data, it may be useful extend our script to count the number of times dictresize is visited, and record the index of the call which created a corrupted ma_table. This information can be used to set a conditional breakpoint in GDB which will stop the program at the correct invocation of dictresize, allowing the programmer to interactively examine the state of the program using GDB. (Note that this counting strategy only applies if the program does not exhibit nondeterminism.)

### 4.3 Comparison to Other Debugging Approaches

**Interactive Debuggers.** When investigating a complex problem with GDB, the developer may need to spend significant time setting breakpoints and restarting the program’s execution in order to traverse from a problem to its causes. This is due to the need to guess the location of each cause, and then narrow down the guess using strategies such as bisection search. In addition, GDB does not work well in cases where the program is multithreaded, since the programmer is required to manually keep track of and control several threads of execution.

**Direct Use of Heavyweight Instrumentation Frameworks.** Heavyweight instrumentation frameworks such as Valgrind or DynamoRIO include prebuilt tools that can do origin tracking. However, these tools only report value origins in response to certain predefined types of errors. In addition, the prebuilt tools capture the origins of all data in the program (instead of only tracking relevant data structures), incurring a significant and unnecessary overhead.

It may be possible to modify the prebuilt tools to track and report origins in response to the particular pattern of corruption seen in this program. However, this requires understanding and altering a multi-thousand-line-of-code program written in terms of low-level concepts such as machine instructions and instrumentation, rather than in terms of high-level concepts such as events and associative arrays. The developer is unlikely to have enough familiarity with low-level instrumentation techniques to be able to make such modifications within a reasonable timeframe.
Chapter 5

Language Specification

This chapter contains a complete specification of the E language. The syntax of the language is described in extended Backus-Naur form, with the notation \{elt\} denoting zero or more occurrences of elt, the notation [elt] denoting zero or one occurrences of elt, brackets (...) denoting grouping and alternate possibilities separated by |.

5.1 Syntax and Lexical Vocabulary

The lexical vocabulary of the E language consists of identifiers, keywords, operators and delimiters, string literals, and numerical literals.

Identifiers are sequences of letters, digits, and the underscore character \_, which must begin with a letter or underscore. Some identifiers are reserved keywords used only in specified syntax elements; the current list of keywords comprises:

array break continue else emit event false for foreach func global if in is probe record return shadow true while

Operators and delimiters include the following syntactically meaningful symbols:

{ } [ ] ( ) ; : <= $ @ * / % + - ! \~ >> <<
< > <= >= == != & ^ | && || ? = += -= *= /= %= &= ^= |= <<= >>= ++ --

Numerical literals are identical to integer constants in the C language, and consist of:

• an optional sign character: + or -, followed by
• an optional prefix indicating octal or hexademical base: 0 or 0x, followed by
• a sequence of decimal digits 0, ..., 9 (if no base was specified) or 0, ..., 7 for octal or 0, ..., 9, a, ..., f (or A, ..., F) for hexadecimal base.

Examples:

-516
0777
0xf34d
String literals are enclosed in double quotes " and may contain the usual backslash-escaped special characters (\n, ", \, and so forth) that are permitted in string constants in the C programming language. Example:

"Hello, World!\n"

Filename patterns are sequences of letters, digits, the underscore _, the hyphen --, and the wildcard character *. Examples:

my-file.c
file*_big.c
*file*.*
0file

Finally, comments are ignored by the E translator and can follow C/C++ or shell style. Examples:

/* This is a comment. */
// And this is a comment.
# This is also a comment.

5.2 Designators, Values, and Types

E is statically-scoped and statically typed. The scope of each language construct is fixed at compilation time. Within each scope, a variable can only reference values of a fixed type. The data types in E comprise word, str, assoc, record, and opaque:

- A word is an unsigned 64-bit integer, used to represent both numbers as well as addresses within the target program’s memory. (Note that signed integer constants and signed integer values from the target program are converted to equivalent unsigned values using two’s complement representation.)
- A str is an ASCII character string.
- An assoc is an associative array. The type of each assoc’s keys and elements is fixed at compilation time. (Thus, an assoc can be indexed by word keys or by str keys, but the same assoc cannot be indexed by a mixture of the two.)
- A record contains named subfields. Different subfields may have different types. The type of each subfield is fixed at compilation time.
- The opaque data type is reserved in the future for values which cannot be represented by any other data types in E. (For example, a reference to an object in a high-level language such as Java could be represented by an opaque. In the future, opaque values may be returned from calls to some built-in functions, and passed as arguments to other built-in functions.)

Variables and values in the E language are referred to by designators. The simplest designator is a single identifier, referring to a value available in the current scope:

basic-designator ::= IDENTIFIER

Entries in an associative array can be referenced by indexing the array with an expression that evaluates to a key of the corresponding type. For example a["field"] refers to the entry with key "field" within the array named a. The entries of an array can themselves be of assoc type, allowing multiple indices (as in b[c][d]). Indices to a nested array can also be given by a comma-separated list (as in b[c,d]).
Subfields in a record can be accessed using the . operator. For example, subfields \( x \) and \( y \) of a record \( r \) can be accessed as \( r.x \) and \( r.y \).

A record can have assoc and record subfields, and likewise an assoc can have record elements, allowing nesting to an arbitrary depth.

If an element of an array or a subfield of a record has not been assigned a value, an access to it yields the default value of the corresponding type (e.g. 0 for \( \text{word} \) or "" for \( \text{str} \)).

A designator can be prefixed using the @ sigil, to indicate that it is being used to define a program state value. This is typically used in assignment operations that define program state values inside an event declaration.

Finally, an identifier prefixed with the $ sigil references a target program variable. Target program variables provide access to variables within the target program.

The set of target program variables that may be accessed at runtime is determined by the event occurrence currently being handled. An event occurrence in the target program may be associated with a location in the target program’s source code, as described in §5.9. For example, the predefined event \( \text{insn.before} \) is associated with the target program statement containing the instruction being executed, while the predefined event \( \text{fn.entry} \) is associated with the entry point of the function being called at the time of the event occurrence. Within the handler block handling the event occurrence, or within any function called (directly or indirectly) from the handler block, the set of program variables that may be accessed is the set of program variables visible at the source code location associated with the event occurrence.

In addition, it is possible to refer to global variables within any module (source file, executable, or shared library) of the target program by calling the built-in function \( \text{require.debuginfo} \) with the name of the module, then accessing the global variables as target program variables. For example, the statement

\[
\text{require.debuginfo("dictentry.h")}
\]

will make available the global variables in the source file \( \text{dictentry.h} \).

When a target program variable is referenced at runtime, an actual variable with the specified name may not exist within the target program relative to the location of the current event occurrence, or within any modules loaded by \( \text{require.debuginfo} \). (For example, the handler for an \( \text{fn.entry} \) event may trigger during calls to different target program functions which have different local variables.) If a nonexistent target program variable is evaluated within a filtering condition, the entire filtering condition evaluates to 0. In other contexts, evaluating a nonexistent target program variable causes the E program to terminate with an error.

### 5.3 Expressions

E supports many of the same operators and expression syntax as C [10]. Some examples of E expressions:
\[(n \& (n-1)) == 0\]
\[\text{div\_count}[$\text{name}$]++\]
\[\text{fn}.\text{name} == "myfunc" \&\& \text{fn}.\text{args}[2] <= 256\]
\[\text{is\_powerof2}(x) ? 3 : 4\]

### 5.3.1 Literals, Designators, and Built-In Operations

The simplest expressions in E are **literals** (numbers, strings, and booleans) and **designators** (described in the previous section).

\[
\text{expr ::= NUMBER} \\
\text{expr ::= STRING} \\
\text{expr ::= "true"} \\
\text{expr ::= "false"} \\
\text{expr ::= handler-literal} \\
\text{expr ::= designator}
\]

Within an expression, the keywords `true` and `false` stand for 0 and 1, respectively. The `word` datatype is used to represent boolean values.

More complex expressions are built up using unary, binary, and ternary operators similar to those of C, as well as function calls. Brackets () can be used to override standard operator precedence:

\[
\text{expr ::= "(" expr ")"} \\
\text{expr ::= UNARY expr} \\
\text{expr ::= expr ("++" | "--")} \\
\text{expr ::= expr BINARY expr} \\
\text{expr ::= expr "?" expr ":" expr}
\]

The binary operations `&&` and `||` have short-circuit semantics: `&&` does not evaluate its second argument if its first argument evaluates to a false value, and `||` does not evaluate its second argument if its first argument evaluates to a true value.

E also supports a shorthand notation for checking whether a value belongs to an explicitly defined set of values. The syntax `a in (b, c, d)` is treated as being equivalent to `a == b || a == c || a == d`.

\[
\text{expr ::= expr "in" "(" expr "," expr "}" "")"}
\]

Function calls consist of the name of the function and a parameter list in brackets:

\[
\text{expr ::= IDENTIFIER "(" [expr "," expr] ")"}
\]

The left-hand operand of an assignment operator (such as `a in a = b`) must be a designator of a scalar (`word`, `str`, or `opaque`) type. (The semantics of assignment with an `assoc` or `record` left-hand operand will be specified in a future version of the language.)

E also supports a shorthand notation for setting a range of array entries to the same value. The syntax `a[b:c] = d` assigns the value of expression `d` to the array entries `a[b], a[b+1], ..., a[c-2], a[c-1]` \(^1\).

\[
\text{expr ::= designator "[" expr ":" expr "]" ":=" expr}
\]

\(^1\)Note that `a[c]` is not included in the range. This behaviour is consistent with array slicing semantics in other commonly used languages such as Python.
5.3.2 Target Program Operations

E includes a number of operations used to access information about the target program, such as the source line number where an event occurs or the contents of a target program variable.

**File and line number matching.** E supports a shorthand notation for matching the source file and line number of an event occurrence against a pattern. If `stmt` is a value of type `record` with subfields `stmt.file` (of type `str`) and `stmt.line_no` (of type `word`), then the expression `stmt is <file>:<line_nos>` compares `stmt.file` against `<file>` and `stmt.line_no` against `<line_nos>`.

In the right-hand side of the `is` operator, `<file>` must be either a string literal (which matches itself) or a filename pattern (similar to a UNIX shell pattern) consisting of alphanumeric characters, the underscore `_`, the hyphen `-`, the period `.`, and the wildcard character `*`. All characters in a wildcard pattern match themselves except for the wildcard character, which matches an arbitrary substring. The line number specifier `<line_nos>` must be either a single line number (which matches itself), a bracketed list of line numbers such as `(25,27)` (which matches any line number in the list), or a wildcard `*` (which matches any line number).

Examples of file and line matching:

```
stmt is file.c:15
stmt is file.c:(25,27)
stmt is file_*.c:*
stmt is "*special*":*
```

Syntax for the `is` shorthand:

```
expr ::= expr "is" is_expressions
is_expressions ::= is_expression
is_expressions ::= (" is_expression ("," is_expression) ")
is_expression ::= ( WILDCARD_PATTERN | STRING )
                |
                (" line_nos | (" line_nos ")")
line_nos ::= ";"
line_nos ::= INTEGER |
```

**Target program operations.** E also supports a variety of C-style dereference operations (`.` , `->` and address-of `&`), as well as a `sizeof()` operator and the `(type)` typecasting operator. These operations can be used to access the contents of target program variables and data structures.

The typecasting operator `(type)` is used to interpret a `word` value as a pointer to a data structure of the specified type. It can be applied to a variable of `word` type. It returns the same `word` value that was passed to it. However, this return value can be used as the operand to any operation that expects a target program variable. If the data type specified in the typecasting operator does not exist in the target program, the E program signals an error.

For example, `(PyDictEntry)me` takes the address stored in the variable `me` and interprets it as a pointer to a data structure of type `(PyDictEntry)`.

The dereferencing operations `.` and `->` are used to obtain the value of a data structure subfield. The `->` operator must be applied to a target program variable (such as `$value`), or to the result of a dereferencing or typecasting operation that accesses a target program variable of C pointer type. The `.` operator must be applied to the result of a dereferencing or typecasting operation that accesses a target program variable of C structure type. Both operations return a `word` value, either the value of the subfield (if the subfield is of a scalar or pointer type), or the address of the data structure stored in the subfield (if the subfield is of a structure or array type).
For example, $\text{var} \rightarrow \text{val}$ and $((\text{PyDictEntry}) \text{me}) \rightarrow \text{me_key}$ return subfields of the data structures whose addresses in the target program are given by $\text{var}$ and $\text{me}$ respectively.

The address-of operator & can be applied to a target program variable, or a data structure subfield returned from a dereferencing operation, and returns the address of the variable or subfield.

The \text{sizeof} operator takes a target program variable and returns its storage size, behaving analogously to the \text{sizeof} operator in C. The \text{sizeof} operator can also take the name of a type declaration, returning the storage size of the type. For example \text{sizeof} (\text{PyDictEntry}) returns the storage size of the \text{PyDictEntry} datatype in the target program.

Similarly to target program variables, type declarations belonging to a target program module can be made available for use in the \text{(type)} typecasting operation and the \text{sizeof} operation using the \text{require_debuginfo()} built-in function.

Syntax for the target program operations:

- \text{expr ::= "sizeof" \("" \$" IDENTIFIER ")"}
- \text{expr ::= \" IDENTIFIER \")" expr}
- \text{expr ::= \text{expr} \\leftrightarrow \" IDENTIFIER}
- \text{expr ::= \" IDENTIFIER}
- \text{expr ::= \" expr}

5.3.3 Precedence Rules

A table of available \text{UNARY} and \text{BINARY} operators, as well as their relative precedence level and associativity, is given below. Operators are binary except where stated otherwise:

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 left</td>
<td>\text{postfix ++ --}</td>
<td>unary post-increment, post-decrement</td>
</tr>
<tr>
<td>1 left</td>
<td>\text{ . } \rightarrow</td>
<td>structure access</td>
</tr>
<tr>
<td>2 right</td>
<td>\text{unary + - ! ~}</td>
<td>unary positive sign, negative sign, not, bitwise not</td>
</tr>
<tr>
<td>2 right</td>
<td>\text{prefix ++ --}</td>
<td>unary pre-increment, pre-decrement</td>
</tr>
<tr>
<td>2 right</td>
<td>\text{prefix {(type) &amp;}</td>
<td>typecast, address-of</td>
</tr>
<tr>
<td>3 left</td>
<td>\text{* / %}</td>
<td>multiplication, division, modulo</td>
</tr>
<tr>
<td>4 left</td>
<td>\text{binary + -}</td>
<td>addition, subtraction</td>
</tr>
<tr>
<td>5 left</td>
<td>\text{\langle \rangle}</td>
<td>(logical) bit shift</td>
</tr>
<tr>
<td>6 left</td>
<td>\text{\langle \langle &gt; &gt; =}</td>
<td>integer comparison operations (for \text{word})</td>
</tr>
<tr>
<td>7 left</td>
<td>\text{== !=}</td>
<td>equality comparisons (for \text{word and string})</td>
</tr>
<tr>
<td>8 left</td>
<td>\text{binary &amp;}</td>
<td>bitwise and</td>
</tr>
<tr>
<td>9 left</td>
<td>\text{^}</td>
<td>bitwise xor</td>
</tr>
<tr>
<td>10 left</td>
<td>\text{</td>
<td>}</td>
</tr>
<tr>
<td>11 left</td>
<td>\text{in}</td>
<td>Is a key present in an assoc?</td>
</tr>
<tr>
<td>11 left</td>
<td>\text{is}</td>
<td>Does a string match a filename pattern?</td>
</tr>
<tr>
<td>12 left</td>
<td>\text{&amp;&amp; and}</td>
<td>logical (short-circuit) and</td>
</tr>
<tr>
<td>13 left</td>
<td>\text{</td>
<td></td>
</tr>
<tr>
<td>20 right</td>
<td>\text{ternary ?:}</td>
<td>ternary if-then-else operator</td>
</tr>
<tr>
<td>21 right</td>
<td>\text{= += -= *= /= %=}</td>
<td>assignment operators</td>
</tr>
<tr>
<td>22 left</td>
<td>\text{,}</td>
<td>evaluate expressions in sequence</td>
</tr>
</tbody>
</table>
The \( , \) operator cannot appear at the top level of an expression being used as a filtering condition (see §5.5), as a function argument, or as the element of a set in the \( \text{in} \) shorthand.

### 5.4 Events, Program State Values, and Context Values

An E program specifies an observation to be performed on some other target program. This observation is described in terms of events occurring in the target program. Each event defines a set of program state values obtained from the target program’s state at the time of the event’s occurrence. For example, the \( \text{fn.entry} \) event occurs at the entry point of every function call, and provides the name program state value, giving the name of the function being called at the time of the event’s occurrence.

In addition, context values provide additional information about the context of the current event occurrence. The context of an event occurrence consists of state transitions (such as function calls, execution of source lines, and execution of threads) occurring simultaneously with the event. For example, the context values \( \text{stmt.file} \) and \( \text{stmt.line.no} \) give the source file name and line number corresponding to the source line currently being executed.

E provides a built-in set of predefined events (see §5.9), mechanisms for defining new events (see §5.7.4), and mechanisms for filtering these events using event expressions. In addition, E provides a built-in set of predefined contexts, which are globally visible variables of type record providing access to various context values.

### 5.5 Event Expressions

An event expression identifies a subset of event occurrences in the target program. It consists of a named event together with an optional list of filtering conditions:

\[
\text{event-expression ::= named-event [:} \text{filtering-conditions}]
\]

\[
\text{named-event ::= IDENTIFIER}
\]

\[
\text{filtering-conditions ::= expr \{ ‚‚ expr \}}
\]

Some examples of named events:

\[
\text{insn.before}
\]

\[
\text{bb.instrument}
\]

\[
\text{fn.entry}
\]

Named events are either predefined events, or are defined elsewhere in the program by an event declaration (see §5.7.4).

The filtering conditions attached to a named event are boolean expressions which can refer to data (program state values and context values) provided by the event. An event expression identifies the set of occurrences of the named event whose data satisfies the filtering conditions.

The filtering conditions attached to a named event are boolean expressions. These expressions can make use of program state values and context values provided by the named event. An event expression with filtering conditions represents the set of occurrences of the named event for which the provided values satisfy the filtering conditions.
5.6 Statements

Executable code in E is contained in blocks, enclosed in curly brackets {}.

Empty statement. Statements in a block do not need to be separated by semicolons. For example, the block

```
{x = x + y ; z = 2 * x}
```

contains two statements, \( x = x + y \) and \( z = 2 \times x \).

However, a semicolon can be used to denote an empty statement.

```
;
```

Syntax:

```
stmt ::= ";"
```

Expression statement. An expression can act as a statement. The expression is evaluated for its side effects, including any assignment operations and function calls:

```
x = x + 1
```

Syntax:

```
stmt ::= expr
```

If statement, While-loop, For-loop. These control-flow statements have syntax which matches the syntax of the corresponding statements in C:

```
if (x) return y else return z
while (k < bb.insns.size) { k++; l-- }
for (i = 0; i < n; i++) printf("%d: %s\n", i, a[i])
```

Syntax:

```
stmt ::= "if" "(" expr ")" stmt ["else" stmt]
stmt ::= "while" "(" expr ")" stmt
stmt ::= "for" "(" [expr] ";" [expr] ";" [expr] ")" stmt
```

Block statement. Curly brackets can be used to group a sequence of statements. A block statement does not introduce a new scope for local variables (unlike a handler block or function body).

Syntax:

```
stmt ::= block-stmt
block-stmt ::= "{ "stmt" "}"
```

Foreach-loop. The foreach-loop visits all keys and values of an associative array:

```
foreach (k, v in a) { /* Obtain keys and values contained in a. */ }
foreach (k in a) { /* Obtain keys only. */ }
```

The foreach-loop also supports iteration of nested arrays. For example, if \( b \) is an array with elements \( b[...][...], \) the following foreach-loops are valid:

```
foreach (k1, k2, v in b) { /* Obtain keys and values of b. */ }
fforeach (k1, k2 in b) { /* Obtain just the keys. */ }
```
Chapter 5. Language Specification

Syntax:

stmt ::= "foreach" "(" IDENTIFIER {"," IDENTIFIER} "in" expr ")" stmt

Break, continue, return. These statements behave identically to the ones in C. return takes an optional return value.

break
continue
return x+1

Syntax:

stmt ::= "break"
stmt ::= "continue"
stmt ::= "return" [expr]

Emit. The emit statement takes the name of a user-defined event and signals the occurrence of that event. Any event handlers for the user-defined event will be executed immediately. Any values within the current scope that were defined with an @ sigil become visible in the event handlers as program state values.

An emit statement can only occur within the body of an event declaration defining the corresponding named event. For example, the following emit statement may occur within the body of a declaration of the form event fn.entry <- ... { ... }:

emit fn.entry

Syntax:

stmt ::= "emit" emit-event-name
emit-event-name ::= IDENTIFIER {"." IDENTIFIER}

5.7 Declarations and Scope Rules

An E program consists of a sequence of declarations, which may include module import declarations, function declarations, probe directives (requests to observe events in the target program), event declarations, and global data declarations:

program ::= { declaration }

E is a statically-scoped language. The set of variables that can be accessed from an expression or statement is determined by that expression or statement’s position in the syntactic structure of the program.

The following variables can be accessed from the body of a probe directive or event declaration:

• local variables within the body,
• local variables defined in filtering conditions of the event expression of the declaration containing the body,
• program state values and context values provided by the named event in the event expression of the declaration containing the body,
• global variables.
The following variables can be accessed from the body of a function or from the initializer expression of a global data declaration (see §5.7.2):

- local variables of the body or expression (including arguments of a declared function),
- global variables.

The following variables can be accessed from a filtering condition within an event expression:

- local variables of the expression,
- program state values and context values provided by the named event in the event expression,
- global variables.

The E translator automatically infers the set of variables that are local to each body or filtering expression in the E program. If a variable is not present in any of the outer scopes of a body or filtering expression, it is considered to be implicitly declared as a local variable.

Before a variable is first assigned, it is initialized with a default value. The default values for local variables of \texttt{word} and \texttt{str} types are 0 and "" respectively. If an unassigned variable is evaluated within a filtering expression, the E program issues a warning.

### 5.7.1 Probe Directive

A \texttt{probe} directive is a declaration that requests observation of a specified event. It consists of an event expression and a handler block containing statements to execute at every event occurrence matching the event expression. For example:

```e
probe fn.call: name=="foo" { printf("Starting foo()") }
```

**Syntax:**

```
declaration ::= "probe" event-expression block-stmt
```

### 5.7.2 Data Declaration

A \texttt{data} declaration describes a global data structure that is visible to all probes, events, and functions in the program. Global data can be a scalar value, a record, an associative array, or a shadow array.

A \texttt{shadow array} is an associative array indexed by integer keys representing target program addresses. It is represented in the E language implementation using an efficient \texttt{shadow memory} data structure.

Scalar value declarations can include an optional initialization expression:

```
global x // Declares a scalar value.
global y = 0
global z = y + 1
```

```
record r // Declares a record value.
array a // Declares an associative array.
shadow s // Declares a shadow array.
```
Syntax:

```plaintext
declaration ::= "global" IDENTIFIER ["=" expr]
declaration ::= "record" IDENTIFIER
declaration ::= "array" IDENTIFIER
declaration ::= "shadow" IDENTIFIER
```

### 5.7.3 Function Declaration

A **function** declaration defines a function which can be called from the body of another declaration or from within a filtering condition. It is possible (but not required) to explicitly specify the input values and return type of a function.

```plaintext
func is_powerof2(n:word):word {
    return (n & (n-1)) == 0
}
```

If the last statement of a function is an expression, the return value of the function is the value of the expression:

```plaintext
func inc(x) { x + 1 }
```

Syntax:

```plaintext
declaration ::= "func" func-definition
func-definition ::= IDENTIFIER "(" params-spec ")"
                   [":" type] block-stmt
params-spec ::= [ param-spec {"," param-spec} ]
param-spec ::= IDENTIFIER [":" definite-type]
definite-type ::= "word" | "str" | "assoc" | "record" | "opaque"
type ::= definite-type | "none"
```

### 5.7.4 Event Declaration

An **event** declaration defines an event whose occurrences can be observed within the target program. It consists of a name for the newly defined event, an event expression, and a handler block.

If the newly defined event is not used in any probe directives or event declarations, the event declaration is ignored by the translator. Otherwise, the handler block of the event declaration executes for every event occurrence matching the event expression, similarly to a probe directive. Within the handler block, an `emit` statement can be used to generate an occurrence of the newly defined event. For example:

```plaintext
event insn_instrument <- bb.instrument {
    foreach (i, @insn in bb.insns) {
        @insn.num = i
        emit insn_instrument
    }
}
```

Any variables with an `@` sigil in the handler block are made available as program state values for the event occurrence. In addition, program state values provided by the declaration’s event expression are made available to the event occurrence. In the above example, each occurrence of the `insn_instrument` event will provide a
program state value `insn` (which will be a value of type record containing a subfield named `insn.num`) as well as a program state value `bb` which was made available by the `bb.instrument` event.

All of the values that are made available to the event occurrence can be accessed by a probe directive whose event expression references the declared event. For example, the following probe directive references the `insn.instrument` event:

```c
probe insn_instrument {
    printf("%d %d %s\n", insn.num, i, insn.opcode)
}
```

The call to `printf()` in this probe will print strings such as "5 0 idiv". Within the probe’s handler block, the values `insn.num` and `insn.opcode` are inherited from the event declaration of `insn.instrument`, while the variable `i` is local to the probe. If `i` within the event declaration was defined using an `@` sigil, the call to `printf()` would print strings such as "5 5 idiv".

Syntax:

```
declarion ::= "event" emit-event-name
    "<-" event-expression block-stmt
```

## 5.8 Built-In Functions

E includes support for the following built-in functions:

- `log(FORMAT, args...)` – print the values specified in `args` according to the format string `FORMAT`, then print a newline. `FORMAT` must be a format string, as described in §5.8.1.
- `printf(FORMAT, args...)` – print the values specified in `args` according to the format string `FORMAT`.
- `println(args...)` – print the values specified in `args`, then print a newline.
- `require_debuginfo(module)` – here, `module` is a value of type `str` giving the name of a module (source file, executable or shared library) loaded by the target program. This function makes the type declarations present in `module` available for typecasting and `sizeof` operations, and makes the global variables present in `module` available as target program variables in the current handler block.

### 5.8.1 Format Strings and Interpolation

The built-in functions `log` and `printf` take as their first argument a *format string* which specifies how the subsequent arguments of the function should be formatted.

A format string can contain *format specifiers*. A format specifier consists of the ‘%’ character followed by one or more characters comprising a *conversion specifier*. E supports the same conversion specifiers as the `printf` function in the C language [52]. For example, the call `log("format %d", 40+2)` contains the format specifier `%d` (with the conversion specifier ‘d’ indicating a base 10 signed integer). This call will print the string "format 42", with the result of the expression `40+2` being formatted as a signed integer in base 10.

In addition, format strings in E support *interpolated specifiers*. An interpolated specifier consists of the ‘%’ character, an optional conversion specifier, and an E expression enclosed in curly brackets `{...}`. For example,
the sequence of statements \texttt{str="format"; log("%{
str} \%d(40+2)")} will print the string "format 42".

A format string must be a string literal, rather than an expression returning a string. Thus, \texttt{log("format \%d \%d", a, b)} is a valid call to \texttt{log()}, whereas \texttt{log(str, a, b)} is not.

\section*{5.9 Predefined Events and Contexts}

\textit{E} includes support for the following predefined events (a detailed description was given earlier in §3.2):

- \texttt{bb.instrument} – for every basic block in the program, \texttt{bb.instrument} is guaranteed to occur once before the first time that block is executed. (If the block is never executed, an \texttt{bb.instrument} event may not occur for that block.)

- \texttt{begin} – occurs once at the beginning of the target program’s execution. Equivalent to \texttt{program.before}.

- \texttt{end} – occurs once at the end of the target program’s execution. Equivalent to \texttt{program.after}.

- \texttt{fn.call} – occurs before a function call, within the caller.

- \texttt{fn.entry} – occurs at the beginning of a function call, within the callee.

- \texttt{fn.exit} – occurs immediately before returning from a function call, within the callee.

- \texttt{fn.return} – occurs immediately after returning from a function call, within the caller.

- \texttt{instruction.after} or \texttt{insn.after} – occurs immediately before an instruction is executed.

- \texttt{instruction.before} or \texttt{insn.before} – occurs immediately after an instruction is executed.

- \texttt{obj.access} – occurs immediately after the target program accesses a memory location.

- \texttt{obj.alloc} – occurs immediately after a memory object is allocated by the target program.

- \texttt{obj.free} – occurs immediately before a memory object is deallocated by the target program.

- \texttt{program.before} – occurs once at the beginning of the target program’s execution.

- \texttt{program.after} – occurs once at the end of the target program’s execution.

A table of program state values provided by the various predefined events is given below:

<table>
<thead>
<tr>
<th>Program State Value</th>
<th>Type</th>
<th>Provided By</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>word</td>
<td>obj.access, obj.alloc, obj.free</td>
<td>memory address</td>
</tr>
<tr>
<td>bb</td>
<td>record</td>
<td>bb.instrument</td>
<td>basic block</td>
</tr>
<tr>
<td>bb.insns[N]</td>
<td>record</td>
<td>bb.insns[N], insn.after, insn.before</td>
<td>\textit{Nth} instruction in basic block value of \textit{Nth} destination operand</td>
</tr>
<tr>
<td>dest[N]</td>
<td>word</td>
<td>insn.after, insn.before</td>
<td>\textit{Nth} value of \textit{Nth} source operand</td>
</tr>
<tr>
<td>name</td>
<td>str</td>
<td>fn.(call, entry, exit, return)</td>
<td>name of function</td>
</tr>
<tr>
<td>opcode</td>
<td>str</td>
<td>insn.insns[N], insn.(after, before)</td>
<td>opcode of instruction</td>
</tr>
<tr>
<td>returnval</td>
<td>target program variable</td>
<td>fn.exit, fn.return</td>
<td>return value of function</td>
</tr>
<tr>
<td>size</td>
<td>word</td>
<td>obj.alloc</td>
<td>size of memory object</td>
</tr>
<tr>
<td>src[N]</td>
<td>word</td>
<td>insn.after, insn.before</td>
<td>value of \textit{Nth} source operand</td>
</tr>
</tbody>
</table>

\textit{E} also includes support for the following predefined contexts:
• \texttt{fn} – the current function call in which the event occurs.

• \texttt{stmt} – the source code statement at which the event occurs.

• \texttt{thread} – the current thread in which the event occurs.

A table of context values provided by each predefined context is given below:

<table>
<thead>
<tr>
<th>Context Value</th>
<th>Type</th>
<th>Provided By</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{fn.name}</td>
<td>\texttt{str}</td>
<td>\texttt{fn}</td>
<td>name of function</td>
</tr>
<tr>
<td>\texttt{fn.caller}</td>
<td>nested context (\texttt{fn})</td>
<td>\texttt{fn}</td>
<td>caller of function</td>
</tr>
<tr>
<td>\texttt{fn.callers}[N]</td>
<td>nested context (\texttt{fn})</td>
<td>\texttt{fn}</td>
<td>Nth caller up the stack</td>
</tr>
<tr>
<td>\texttt{fn.callsite}</td>
<td>nested context (\texttt{stmt})</td>
<td>\texttt{fn}</td>
<td>\texttt{stmt} where function was called</td>
</tr>
<tr>
<td>\texttt{stmt.file}</td>
<td>\texttt{str}</td>
<td>\texttt{stmt}</td>
<td>source file of \texttt{stmt}</td>
</tr>
<tr>
<td>\texttt{stmt.line.no}</td>
<td>word</td>
<td>\texttt{stmt}</td>
<td>source line of \texttt{stmt}</td>
</tr>
</tbody>
</table>

A more detailed explanation of the nested contexts \texttt{fn.caller} and \texttt{fn.callsite} was given in §3.4.
Chapter 6

Language Translator

In order to carry out an observation, an E program must first be translated into an observation plugin for a heavy-weight instrumentation framework (the target framework). A plugin for a heavyweight instrumentation framework is typically a program in a high-level language such as C built on top of an interface provided by the target framework. The E implementation targets existing frameworks instead of incorporating its own custom-built instrumentation mechanism for two reasons. First, existing heavyweight instrumentation frameworks include sophisticated capabilities which the E implementation can take advantage of. For example, Valgrind incorporates an optimizing JIT compiler which is able to reduce the execution overhead for instrumented code. Second, the design of the E language is intended to act as a simple, general interface for existing observation mechanisms. In order to evaluate whether this design goal has been accomplished, it is necessary to develop an implementation which targets already existing frameworks.

The target framework’s interface does not support the entire functionality of the E language. In general, frameworks such as Valgrind and DynamoRIO only provide a callback mechanism that allows a plugin to instrument the target program’s code. Support for the predefined events in the E language must be implemented on top of this callback mechanism. In order to provide this support, for each target framework the E implementation includes a runtime support layer and a target framework description. The runtime support layer consists of routines implemented on top of the target framework’s library interface (§6.5). The target framework description contains information specifying functions, program data and context data supported by the functionality of the target framework and the runtime support layer built on top of it.

The E language translator is organized into several passes. During the parsing pass, the E program is translated into an abstract syntax tree (AST) (§6.2). During the intermediate representation construction pass, the abstract syntax tree is translated into a basic-block intermediate representation (IR) (§6.3). During the specialization pass, the information in the target framework description is used to convert the IR into a form that is specified in terms of functionality provided by the target framework and the associated runtime support layer (§6.4). During the code generation pass, the specialized intermediate representation is used to generate source code for a plugin (§6.6). Finally, the generated plugin is compiled, linked with the target framework, and executed.

The scope of the current work is to evaluate the feasibility and efficiency of implementing E on top of different heavyweight instrumentation frameworks. The current implementation is able to produce plugins for the Valgrind and DynamoRIO target frameworks. This implementation includes support for language features sufficient to implement the benchmark programs needed for evaluating language performance (as described in Chapter 7), while omitting support for such language features as context and event declarations, nested contexts (such as
the fn.caller, fn.caller[N] hierarchy), features for retrieving target program data which rely on debug information, as well as features related to explicit instrumentation of the target program such as handler literals and the instrument.before()/instrument.after() built-in functions.

The set of currently implemented E language features nonetheless provides sufficient functionality to support useful observations. For example, the E program shown in Figure 6.1 implements a use-after-free error detector for C/C++ programs which utilize the memory management functionality from the standard library (such as the malloc and free functions). The use-after-free program observes the target program according to the following strategy:

- Whenever a new memory object is allocated, an obj.alloc probe directive records the object in a shadow array watched tracking the set of watched addresses (lines 4-7).

- Whenever a memory object is deallocated, an obj.free probe directive records this fact in the array obj.freed (lines 16-18).

- Whenever an address in the shadow array watched is accessed by the target program, an obj.access probe outputs a message reporting a use-after-free error if the address is part of freed memory corresponding to a previously deallocated object, as determined by checking the obj.freed array (lines 9-14). Memory accesses performed by the memory management functions in the C standard library (consisting of functions in malloc.c) are excluded from this check by the second filtering condition on line 9, since the memory management system has a valid reason to access freed memory.

The output of the use-after-free program contains a message for each event occurrence where the target program accesses freed memory. The overall structure of the use-after-free program is similar to that of the origin-tracking program described in an earlier chapter (see Figure 4.2 in Chapter 4).

The following section gives a high-level overview of how features of the E language are implemented by adding runtime support layer functionality to the target framework and by transforming the E program. Subsequent sections describe, in sequence, the transformations performed by each of the passes in the E translator.
6.1 Bridging the E Language and Target Framework Functionality

Although the functionality of a heavyweight instrumentation framework supports the full range of observations that can be described in the E language, there is a significant difference between the Valgrind and DynamoRIO frameworks and the E language in terms of the provided means of description. In order to translate E programs into plugins for Valgrind or DynamoRIO, differences between the E language’s event model and the target framework’s instrumentation interface are bridged by implementing an additional layer of functionality on top of the target framework, called a runtime support layer. The runtime support layer translates the high-level events supported by the E language to the low-level instrumentation supported by the target framework. The E translator transforms an E program into a form specified solely in terms of the functionality provided by the runtime support layer.

There are a number of differences which must be bridged by the E translator and runtime support layer.

Firstly, in the E language, an observation is specified by probe directives which declaratively identify event occurrences in the target program, while an observation based on a heavyweight instrumentation framework must be programmed by invoking target framework routines which instrument the target program with the addition of instructions for performing the observation. The runtime support layer provides a set of higher-level instrumentation routines implemented on top of this low-level instrumentation interface. Each routine generates instrumentation to observe a particular built-in event by inserting instructions which invoke an event handler. An event handler is a function which contains code to handle an event occurrence. An event handler processes an event occurrence using the following sequence of steps. The event handler collects the program state values and context values visible at the point of the event occurrence and stores them in a context data structure which is made available to the E program. Then, the event handler executes the statements in the handler blocks of all probe directives whose event expressions are based on the event. Finally, the event handler frees the memory associated with the context data structure.

Secondly, an E program can contain high-level data structures such as associative arrays, and can access context and program state values simply by referencing them as variables. In contrast to this, obtaining program data in a target framework plugin requires carrying out an explicit procedure for accessing the target program’s memory. The runtime support layer bridges this difference by including routines which perform the procedures necessary for obtaining program data. The runtime support layer also provides implementations for high-level E language data structures.

Thirdly, the translation process must also generate variable declarations and infer data type information which is not explicitly declared in the E program, but which must be specified explicitly in the C source code of the plugin. These declarations are generated by a type inference procedure (described in §6.4.3).

The target framework description specifies how each of the available events, contexts, and built-in functions is implemented using invocations to target framework and runtime support layer routines. Once the transformed E program is specified in terms of functionality listed in the target framework description, every operation in this program can be directly translated to C source code which invokes functionality in either the target framework or the runtime support layer. This makes it possible to generate the entire source code of the final plugin.

Here is an example which illustrates how the transformation of probe directives is performed. The following is a probe directive which must be translated into a plugin for the Valgrind framework.

```
probe obj.access: stmt is myprogram.c:*, addr in watched {
  base_addr = watched[addr]
  printf("obj.access addr=%x, base_addr=%x @myprogram.c:%d\n", 
     addr, base_addr, stmt.line_no)
}
```
This probe directive contains filtering conditions which restrict the scope of the observation to a subset of all \texttt{obj.access} events. The target framework description for Valgrind includes the \texttt{obj.access} event, which has occurrences at every memory access during an execution of the target program.

The translator will add a prefix to the basic-block intermediate representation of the probe. The instructions in this prefix will check the filtering conditions and, if any of them evaluate to false, exit the probe body. Later, the translator will generate a C function with a body which executes the statements of the basic probe. This function will be included in the plugin and designated as a probe handler (see \S 6.5.2) which will be invoked on every occurrence of the \texttt{obj.access} event.

### 6.2 Parsing Pass

During the parsing pass, the E program’s source code is parsed into an abstract syntax tree representation (AST). Each declaration in the program is parsed into a separate subtree of the AST representation of the program. Figure 6.2 shows a probe directive (taken from the \texttt{use-after-free} example in Figure 6.1) and the corresponding AST after parsing.

### 6.3 IR Construction Pass

During the IR construction pass, an intermediate representation is constructed from the AST subtree corresponding to each declaration. The intermediate representation for a declaration is a data structure consisting of attributes describing the declared entity, together with an executable part represented using a sequence of elementary instructions organized into basic blocks.

#### 6.3.1 Translation – Declarations

The attributes stored in the intermediate representation for a declaration vary depending on the type of entity being declared.

The representation for a probe directive contains a unique identifier for the probe and a representation of the event expression. The event expression is represented by the name of the base event, together with a list of filtering conditions. The executable code of each filtering condition is represented by a separate data structure containing instructions and basic blocks.

A function declaration is represented by an identifier naming the function and a function signature listing the names of the function’s parameters.

A global data declaration is represented by an identifier naming the global data structure.

Finally, an event declaration is represented by an identifier naming the event being defined and a representation of an event expression.

#### 6.3.2 Translation – Basic-Block IR

The executable part of a declaration’s intermediate representation is represented using the simple instruction set shown in Table 6.1. Each instruction has an opcode and a list of operands.

In order to simplify analysis of the E program’s control flow, the instructions in a declaration’s executable part are grouped into elementary units of control flow called basic blocks. A basic block consists of a sequence of non-control-flow instructions (MOV, CALL) terminated by a control-flow instruction (BR, BR_IF, RETURN).
probe obj.alloc {
    for (i = addr; i < addr+size; i++) watched[i] = addr
    obj_freed[addr] = 0
}

// AST representation:

// Basic-block IR representation:

Figure 6.2: A probe directive in E and its abstract syntax tree and basic-block IR representations.
Operation Description

<table>
<thead>
<tr>
<th>BR (bb)</th>
<th>unconditional branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR_IF ⟨val⟩, ⟨bb₁⟩, ⟨bb₂⟩</td>
<td>conditional branch</td>
</tr>
<tr>
<td>⟨lval⟩ ← MOV ⟨val⟩</td>
<td>move operation</td>
</tr>
<tr>
<td>⟨lval⟩ ← CALL ⟨fn⟩ ((⟨val₁⟩, ⟨val₂⟩), . . .)</td>
<td>function call or primitive operation</td>
</tr>
<tr>
<td>RETURN ⟨val⟩</td>
<td>return from a function or handler block</td>
</tr>
<tr>
<td>EMIT ⟨event⟩</td>
<td>signal event occurrence (within an event declaration)</td>
</tr>
</tbody>
</table>

Table 6.1: Instruction set of the E intermediate representation. Terms in angular brackets denote instruction operands: ⟨bb⟩ denotes a basic block operand, ⟨val⟩ denotes a constant or variable operand, ⟨lval⟩ denotes a variable operand, ⟨fn⟩ denotes a function operand, ⟨event⟩ denotes the name of an event.

The branch instructions BR and BR_IF represent a transfer of control to the first instruction of another basic block. As a result of these rules, each basic block has exactly one entry point and exactly one exit point. This scheme for representing a program’s control flow is standard in most compilers [1].

Figure 6.2 shows a probe directive (taken from the use-after-free example in Figure 6.1) and the corresponding basic-block representation after IR construction.

The executable part of a declaration’s intermediate representation is generated by translating each statement from the AST declaration into a sequence of instructions. This translation is done according to the translation algorithm detailed in [12]. This translation algorithm recursively processes the subtrees of each statement or expression, generating instructions for the subtrees and then generating instructions for the outermost operation in the statement or expression. For example, a subtree representing a function call is processed by generating instructions which compute each argument of the function call and then generating a CALL opcode whose operands include the computed arguments.

Below is a description of how the translation algorithm handles different E language elements, translating each element into the equivalent intermediate representation. Depending on the language element, the result of translation could be an instruction operand, a single instruction, a sequence of instructions or a more complex structure of instructions and basic blocks.

String and integer constants in the E program are translated into constant operands, while variable accesses are translated into variable operands. Variable operands are represented by an access path consisting of a variable access followed by an optional sequence of field accesses and array accesses. Within an access path, a field access or a variable access is represented by an identifier naming the field or variable and an array access is represented by an identifier referencing a local variable which holds the value of an array index. For example, a, b[c], and d[e].f are valid access paths which can be included in the intermediate representation as variable operands. An expression with a complex array index such as a[b+c] cannot be represented by an access path and is instead translated into two instructions: an instruction that stores the index expression (b+c) in a temporary variable %r, and an instruction that references the array element as a variable operand a[%r].

Other operations in the E language are translated as follows:

- Function calls are translated into CALL instructions containing a function operand and arguments. A function operand is represented by an identifier naming the function being called.
- Primitive operations such as +, -, *, /, ... are translated into CALL instructions invoking corresponding built-in functions.
- The assignment operator = is translated into a MOV instruction.
a += b
// expanded to:
a = a + b

a[b:c] = d
// expanded to:
%r0 = b; %r1 = c; %r2 = d
for (%r3 = %r0; %r3 < %r1; %r3++)
a[%r3] = %r2

foreach (key, val in assoc) body
// expanded to:
%r0 = %assoc_iter_start(assoc)
while (%assoc_iter_next(%r0, assoc)) {
  key = %assoc_iter_key(%r0)
  val = %assoc_iter_val(%r0)
  body
}

Figure 6.3: Expansions of some E language constructs into equivalent code based on simpler operations. These expansions are performed on the AST during generation of basic-block IR. %r0, %r1, ... denote temporary variables. %assoc_iter_start, %assoc_iter_next, %assoc_iter_key, %assoc_iter_val are built-in functions (provided by the runtime, but not exposed in the E language) operating on an 'iterator' value of type opaque.

- Boolean operations such as && and || have short-circuit semantics: && does not evaluate its second argument if its first argument evaluates to a false value, and || does not evaluate its second argument if its first argument evaluates to a true value. These operations are translated into corresponding structures of basic blocks and control-flow instructions.

- Complex expressions are translated into sequences of instructions. Each instruction stores its result in an automatically generated temporary variable. The algorithm avoids computing the same value repeatedly by reusing the results of side-effect-free primitive operations whenever possible. For example, the expression (a+b) < c || (a+b) > d will produce a sequence of instructions that computes the subexpression a + b only once, storing the result in a temporary variable %r0 that can be used to compute both %r0 < c and %r0 > d.

- Control-flow structures such as if statements and loops are transformed into equivalent structures of basic blocks and control-flow instructions. The branch instructions BR and BR_IF take one or more basic block operands. A basic block operand identifies a basic block occurring in the same declaration as the branch instruction.

A number of more complex language constructs such as operator assignment (+=), range assignment (a[b:c] = ...) and the foreach loop are expanded into equivalent language constructs, which are then transformed into basic-block IR. Examples of such expansions are shown in Figure 6.3.

### 6.4 Specialization Pass

During the specialization pass, the intermediate representation of each declaration is transformed into a form that directly invokes functionality provided by the target framework and the associated runtime support layer. This
transformation is organized into a sequence of smaller steps and is performed according to information provided by the target’s framework description.

The target framework description contains the set of built-in functions, events, and contexts supported by a target framework. For each built-in function, the target framework description contains a function signature specifying the name of the function as well as its parameter types and return types. For each built-in event, the target framework description contains the name of the event together with a symbol table whose entries identify the program state values provided by the event. For each built-in context, the target framework description contains the name of the context together with a symbol table whose entries identify provided context values. These symbol tables are used by the subsequent steps of the translation procedure to identify which instruction operands reference program state values and context values.

Symbol tables are created during the specialization pass. The translator creates a local symbol table for every intermediate representation of a declaration that has an executable part. This local symbol table is initially empty. Later, it will be populated with entries for variables whose scope is local to this particular declaration. In addition, the translator creates a global symbol table which will be populated with entries for globally declared data structures. Finally, the target framework description provides symbol tables which contain entries for program state values and context values defined by the E language’s predefined events and contexts.

During the scope labeling step, the specialization pass creates variable entries in the local and global symbol tables. During the type inference step, the specialization pass infers a valid type for each variable entry according to the rules of the E language’s static type system.

### 6.4.1 The Scope Labeling Step

The purpose of the scope labeling step is to identify the scope of each variable in the program and to create corresponding symbol table entries. According to the scoping rules of the E language, the statements within a declaration may reference local variables, global variables, program state values and context values. Symbol table entries for these variables are stored, respectively, in local symbol tables, global symbol tables, and symbol tables included in the target framework description.

A symbol table contains a list of entries representing variables. The symbol table entry for each variable identifies the name of the variable and its type. The variable is identified by an access path consisting of field accesses and array accesses. The types of variables are initially unknown and are filled in during the later type inference step.

The translator uses the following rules to determine the scope of a variable referenced by a variable operand in an instruction. If there is an explicit global data declaration for the variable, then it is a global variable. If the instruction referencing the variable occurs within a probe declaration and a corresponding entry is present in the target framework description’s symbol table for a context or event referenced by the probe, then the variable is a context or program state value. Finally, if none of the above rules apply, then the variable is a local variable and the translator creates a corresponding entry in the local symbol table whose scope includes the instruction which references the variable.

Symbol table entries are created for scalar variables, records and arrays.

A scalar variable is represented by a single symbol table entry whose identifier is the name of the variable.

A record is represented by a base entry as well as subfield entries which store the types of the record’s subfields. The base entry of a record is identified by the name of the record and contains the type record. Subfield entries are identified by access paths which include the name of the record together with the name of the
subfield. For example, a record \( r \) with subfields \( x \) of type \( \text{str} \) and \( y \) of type \( \text{word} \) will be represented by a base entry with identifier \( r \) of type \( \text{record} \) as well as subfield entries \( r.x \) of type \( \text{str} \) and \( r.y \) of type \( \text{word} \).

Similarly, an array is represented by a base entry of type \( \text{assoc} \) as well as a subfield entry which stores the type of the array’s elements. The type of the array’s indices is encoded within the access path used to identify the array’s subfield entry. Initially, the type of the array’s indices is unknown, and the identifier of the array’s subfield entry contains the array access ‘\([\text{unknown}]\)’. Thus, an array \( a \) will be represented by a base entry with identifier \( a \) as well as a subfield entry with identifier \( a[\text{unknown}] \). When the type of the array’s indices is determined during the type inference step, the identifier of the array’s subfield entry is modified to match the inferred type of the indices. The array access ‘\([\text{unknown}]\)’ is replaced with ‘\([\text{word}]\)’ or ‘\([\text{str}]\)’ depending on whether the array is indexed by integer or string keys respectively.

These rules apply recursively to records nested in arrays and arrays nested in records. For example, an array \( r[x] \) indexed by keys of type \( \text{word} \) whose elements are records with subfields \( r[x].a \) of type \( \text{word} \) and \( r[x].b \) of type \( \text{str} \) will be represented by the following entries:

- \( r \) with type \( \text{assoc} \)
- \( r[\text{word}] \) with type \( \text{record} \)
- \( r[\text{word}].a \) with type \( \text{word} \)
- \( r[\text{word}].b \) with type \( \text{str} \)

### 6.4.2 The Inlining Step

The purpose of the inlining step is to transform each probe declaration into a basic probe whose event expression consists of a predefined event without any filtering conditions. The inlining step adds a prefix to the body of each probe. The instructions in the prefix check the filtering conditions and, if any of them evaluate to false, exit the probe body.

### 6.4.3 The Type Inference Step

\( E \) is a statically typed language. The type of every variable and expression is determined at translation time, and any use of a variable in violation of the type system’s rules is detected and reported as an error. Because data type information is not explicitly declared in the \( E \) program, a type inference procedure is required to compute a valid type for each variable.

**Type Information.** The intermediate representation supports the basic types described in the \( E \) language specification. The types \( \text{word} \) and \( \text{str} \) are \( \text{scalar} \) types, which identify unsigned integers and strings respectively. The type \( \text{record} \) identifies records, \( \text{assoc} \) identifies associative arrays, and \( \text{shadow} \) identifies arrays declared with the \text{shadow} keyword.

In addition, the special types \( \text{unknown} \), \( \text{scalar} \), and \( \text{error} \) are used before and during the type inference process. The \( \text{unknown} \) type indicates that the type of a variable has not yet been inferred. The \( \text{scalar} \) type indicates that a variable is either of \( \text{word} \) or of \( \text{str} \) type, but the type inference algorithm has not yet encountered sufficient information to determine the exact type. The \( \text{error} \) type is recorded in a situation where different uses of a variable imply contradictory types.

The data types supported by the intermediate representation are partially ordered into a hierarchy of compatible subtypes. The \( \text{word} \) and \( \text{str} \) types are subtypes of \( \text{scalar} \), and every type is a subtype of \( \text{unknown} \).

**Type Inference Algorithm.** The purpose of the type inference algorithm is to compute a valid type for every variable in the program. The type inference algorithm is based on the fact that the built-in functions of
the E language are subject to type constraints: they require values of specific types as their parameters and are guaranteed to return values of specific types.

The type inference algorithm iterates over every instruction of the program. If an instruction is an operation with type constraints, the type inference algorithm checks the symbol table entry for each of the instruction’s operands. The type stored in the symbol table entry is compared to the type required by the instruction’s type constraints. If the type required by the instruction is a subtype of the type stored in the symbol table entry, the subtype is recorded in the symbol table entry. If the type required by the instruction is not compatible with the type stored in the symbol table, the type of the symbol table entry is set to error and the translator reports a type mismatch error.

For example, when the type inference algorithm checks the instruction `a <- CALL +(b,c)`, the type required by the instruction for `a`, `b` and `c` is `word`. The type `word` is a valid subtype of `scalar` and `unknown`. If the symbol table entries for `a`, `b` and `c` are of either `scalar` or `unknown` type, these entries will be updated to type `word`.

The type constraints on some operands of an instruction may vary depending on which types are currently recorded for other operands of the same instruction. Therefore, when the type stored in a symbol table entry is modified while processing an instruction, this modification may also affect the type constraints on operands of other instructions in the program which reference the same symbol table entry. Because of this, the type inference algorithm must repeatedly process the instructions of the program until no more symbol table entries can be modified.

For example, when the currently recorded type for both `a` and `b` is `unknown` the instruction `a <- MOV b` does not require a specific type for either of its operands. Once a different type has been recorded for either `a` or `b`, the instruction requires the other operand to be of the same type.

The type inference algorithm can only replace the type stored in a symbol table entry with its valid subtype or with the `error` type. The longest possible sequence of modifications is `unknown → scalar → word/str → error`. Therefore, each symbol table entry can only be modified a finite number of times and the type inference algorithm is guaranteed to terminate. This method of guaranteeing termination is standard across a wide range of analysis algorithms used in compilers [13].

### 6.5 Plugin Structure and Runtime Support Layer

A heavyweight instrumentation framework controls a target program’s execution according to the scheme described earlier in §2.3. The purpose of the instrumentation framework is to create and execute a version of the target program which has been modified by the addition of code which executes an observation. This is called an instrumented version of the target program. Both Valgrind and DynamoRIO generate the instrumented version of the target program on-demand as the target program’s control-flow visits previously unexecuted code.

On-demand instrumentation of the target program proceeds as follows. The target program’s executable code is considered to be divided into small units of code called basic blocks. The target framework manages a code cache containing instrumented basic blocks. Initially, this code cache is empty. Because the code cache is initially empty, the target framework begins the target program’s execution by locating and instrumenting the first basic block of the target program. The instrumentation is done by passing the uninstrumented basic block to a plugin. The plugin instruments the basic block by inserting instructions which execute an observation. The plugin returns the instrumented basic block to the framework. The framework adds the basic block to the code cache and resumes the target program’s execution at that basic block. Executable code that has been placed within the code cache
can be executed without involvement from the instrumentation framework as long as it does not branch to an uninstrumented basic block. Within the code cache, any branch instruction leading to an uninstrumented basic block is replaced by an instruction that transfers control back to the instrumentation framework. After receiving control from such a branch instruction, the instrumentation framework scans the target program’s executable code, locates the uninstrumented basic block originally targeted by the branch instruction and repeats the process of passing the basic block to the plugin for instrumentation. The instrumented basic block is placed within the code cache and the target program’s execution is once again resumed.

A plugin is a program which implements a particular observation and is written according to requirements imposed by the target framework. Both the Valgrind and DynamoRIO frameworks impose very similar requirements on the structure and functionality of a plugin. The plugin must provide a function, known as a basic block callback, which performs instrumentation on each basic block of the target program. In addition, the plugin is required to provide an initialization function. The framework calls this function before beginning the target program’s execution. Actions pertaining to configuration of the target framework can only be performed within the initialization function, before the target program begins executing. These actions include creating a shadow memory, as well as designating additional callback functions for responding to occurrences such as thread creation or shared library loading. In addition, the initialization function provides a convenient place for initializing the plugin’s own global data structures.

In order to bridge the functionality of the general, framework-independent E language and the functionality and requirements of a particular framework, almost all of the framework-dependent aspects of the E implementation have been separated from the core translator implementation and contained within the plugin template and runtime support layer. To ensure that the E implementation is easy to adapt to different target frameworks, it is crucial that logic which is highly specific to the target framework be organized in a self-contained manner.

The plugin template, included in the target framework description, is an incomplete skeleton of a plugin for the target framework written in C. The plugin template contains the minimal set of routines required to generate a working plugin for the target framework, as well as a number of empty sections which are filled in by the code generation pass. These sections are located in a way which satisfies the requirements of the target framework. For example, a section for initializing global data structures is located within the initialization function.

The runtime support layer is a library of routines implemented on top of the target framework’s library interface. The runtime support layer implements the standard set of predefined events and data structures available in the E language. The runtime support layer for every target framework implements the same set of routines. This allows the code generation pass to produce code that invokes runtime support layer routines and is reusable across different target frameworks.

### 6.5.1 Structure of the Generated Plugin

As described earlier, the structure of the generated plugin is subject to requirements from the target framework. The E translator generates the plugin based on a plugin template containing elements required by the target framework such as a basic block callback and an initialization function. In addition to these required elements, the plugin also includes an event handler for each predefined event and a probe handler for each of the probe declarations in the E program.

The diagram in Figure 6.4 illustrates the interactions between the target framework and elements of the plugin.

During an observation of the target program, the target framework controls the target program’s execution. The target program’s executable code is divided into minimal units of control flow called basic blocks. During the execution, if the target program is about to transfer control to a new, previously unexecuted basic block, the
target framework calls the plugin’s basic block callback. The basic block callback instruments the basic block by modifying its instructions and then returns control to the target framework. The target framework then resumes the target program’s execution. Each basic block in the target program is instrumented at most once. During the target program’s execution, calls by the framework to the basic block callback are interweaved with the execution of previously instrumented basic blocks.

The basic block callback is a function provided by the runtime support layer. The basic block callback instruments a basic block as follows: at any instruction, for every predefined event that occurs at that instruction, the basic block callback inserts a call to the event handler for that predefined event.

Each predefined event in the E language has a corresponding event handler function within the runtime support layer. The purpose of an event handler is to collect information about an event occurrence in the form of program state values and context values. Program state values and context values visible at the point of the event’s occurrence are stored in a context data structure. Multiple probe directives in a E program may use the same predefined event. The event handler function invokes the probe handlers corresponding to all probe directives which use the event.

A probe handler is a function whose purpose is to execute code translated from the statements within the body of a particular probe declaration. Thus, each probe declaration from the E program is translated to a corresponding probe handler, which takes as its parameter a context data structure.

The plugin template also includes an initialization function, which is called by the target framework before the target program begins executing. The initialization function designates a set of probe handlers for each predefined event by calling routines within the runtime support layer. If a predefined event is not used by any probe in the E program, the plugin will not designate any probe handlers for that event, and the runtime support layer’s basic block callback will not insert any calls to the corresponding event handler.

Figure 6.5 shows a simplified version of the plugin template used for the Valgrind framework. This plugin template includes a number of callbacks required by the Valgrind framework:

- `et_pre.clo_init()` and `et_post.clo_init()` contain initialization code for the plugin,
- `et_instrument()` is a basic block callback,
- `et_fini()` contains code which executes after the target program completes.
6.5.2 Technical Details of Event Handling

Both the Valgrind and DynamoRIO target frameworks require the plugin to include a basic block callback which performs instrumentation of the target program. This callback is implemented in the runtime support layer. For both frameworks, the basic block callback instruments a basic block as follows: for every instruction in the basic block, the callback considers the set of predefined events which occur at that instruction.

For example, let’s consider two events. Let’s take the insn event which occurs at every instruction and the obj.access event which occurs only at instructions which contain a memory access. The basic block callback will insert a call to the event handler for insn at every instruction, but will only insert a call to the handler for the obj.access event if the instruction contains a memory access.

In principle, almost all predefined events could be supported with this direct instrumentation approach. However, certain predefined events can be more effectively implemented with other functionality provided by the target framework. In particular, the Valgrind framework provides function wrapping support, which allows a plugin to specify a wrapper function for a selected function in the target program. The wrapper function contains additional code to be executed before and after each invocation of the target program function. The obj.alloc event is implemented by a wrapper function for the C standard library malloc() function. After the malloc() call returns, this wrapper function computes program state values and context values and invokes the probe handlers for the obj.alloc event. Similarly, the obj.free event is implemented by a wrapper function for free().

The runtime support layer includes routines to designate probe handlers for each of the predefined events of the E language. For example, the runtime support layer routines et_set_obj_alloc_handler() and et_set_obj_access_handler() designate probe handlers for the obj.alloc and obj.access events respectively, as seen in Figure 6.5. These routines must be invoked from the generated plugin’s initialization function (et_pre_clo_init() in the case of Valgrind).

6.5.3 Locking

The target program can be multi-threaded, with event occurrences triggering the execution of probe handlers in simultaneously running threads. In order to coordinate access to global data structures from probe handlers executing in separate threads, a simple multi-phase locking scheme is implemented. Multi-phase locking is a standard scheme for coordinating access to multiple shared resources, typically used to coordinate transactions in databases [20]. Each global data structure has an associated lock. At the beginning of every probe handler, the probe handler uses the global_lock() and global_unlock() macros provided by the runtime support layer to request the locks for all of the global data structures referenced in the probe handler. All probe handlers request the locks in a standard order to prevent the possibility of a deadlock. The multi-phase locking ensures that every probe handler runs as an atomic operation. Since no two probe handlers can run simultaneously, this makes the semantics of an observation in E more predictable and understandable.

6.5.4 Data Structures – Associative Arrays

The E program can include operations on associative arrays. Associative arrays are implemented using hash tables. Both Valgrind and DynamoRIO provide hash table libraries. The runtime support layer implements a table_t data type based on these hash table libraries, and provides a common interface consisting of the functions assoc_init(), assoc_contains_key(), assoc_get(), and assoc_set(). These functions initialize an associative array, test element inclusion, and read and modify its elements respectively.
Figure 6.5: A simplified version of the template used for generating a Valgrind plugin, with trigger points for the
program.before, program.after, bb.instrument, obj.alloc and obj.access events.
An associative array whose entries are of scalar type is represented by a single hash table. An associative array storing entries of record type is represented by several hash tables storing scalar values, with one hash table per record subfield.

### 6.5.5 Data Structures – Shadow Arrays

As described in §3.8, the E language provides associative arrays indexed by target program addresses, called shadow arrays. Shadow arrays are implemented using the target framework’s shadow memory functionality, which supports faster lookup than an ordinary hash table. There is a significant gap in functionality between a shadow array in the E language and shadow memory in the Valgrind and DynamoRIO frameworks: a shadow array can store entries of any type, whereas a shadow memory can only store a small integer value for every target program address. On commonly used architectures such as x86 and ARM, each target program address corresponds to one byte of memory [9, 2]. The size of the values stored in the shadow memory is configured during the plugin’s initialization, and can range from one bit of shadow memory data stored for every byte of target program memory to two bytes of shadow memory data for every byte of target program memory. None of these configurations allow an E language value to be stored directly within the shadow memory.

To bridge this gap, the runtime support layer represents a shadow array with two data structures: a shadow memory provided by the target framework as well as an ordinary hash table. For each address of the target program memory, the shadow memory stores one bit of data, which is set to 1 if the shadow array contains an entry indexed by that address, and set to 0 otherwise. If the shadow array contains an entry indexed by an address, the value of that entry is stored in the hash table.

The interface to access a shadow array consists of the same functions as the interface for an ordinary associative array, namely `assoc_contains_key()`, `assoc_get()` and `assoc_set()`. The implementations of these functions automatically distinguish a shadow array from a regular array. The functions `assoc_contains_key()` and `assoc_get()` access the shadow memory before the hash table. If the shadow memory of the address contains the value 0, the hash table will not be accessed. This improves the efficiency of reading the array or testing for inclusion, since the presence or absence of a key can be checked without needing to calculate its hash function. In order to ensure that the shadow memory correctly reflects the state of the shadow array, the `assoc_set()` function writes the value 1 to the shadow memory for an address when it writes a new entry for that address to the shadow array.

The initialization code for a shadow array consists of a call to `shadow_init()`, a runtime support layer function which initializes a shadow memory in addition to a hash table. A flag variable is set within the initialized `array_t` data structure indicating the array to be a shadow array.

### 6.6 Code Generation Pass

The code generation pass generates an observation plugin. As mentioned earlier, the plugin is a program in C built on top of interfaces provided by the target framework.

The plugin source code is generated from the specialized intermediate representation. This intermediate representation matches the functionality available in the target framework. Because of this, the code generation procedure is a straightforward mapping of intermediate representation constructs into equivalent C source code constructs. The generated code is used to fill in the empty sections of the plugin template.

C source code corresponding to each declaration in the E program is generated as follows:
The following probe directive:

```c
probe obj.access: addr in watched, !(stmt is malloc.*) {  
  base_addr = watched[addr]  
  if (obj_freed[base_addr])  
    printf("use-after-free on %x (accessed %x) at %s:%d
",  
      base_addr, addr, stmt.file, stmt.line_no)  
}
```

... produces the probe handler:

```c
void _g11_probe2_resolved(void *ctx)  
{
  unsigned long _l8_addr = (unsigned long) ((ctx_ev_obj_access*)_ctx)->addr;
  unsigned long _l14_r11;
  char *_l14_r12;
  unsigned long _l14_r14;
  unsigned long _l14_r5;
  unsigned long _l14_base_addr;
  unsigned long _l14_stmtline = (unsigned long) ((ctx_ev_obj_access*)_ctx->stmtline;

  // from S13::handler  
  unsigned long _l13_r10;
  unsigned long _l13_r7;
  char *_l13_r8;
  unsigned long _l13_r9;
  char *_l13_stmtfile = (char *) ((ctx_ev_obj_access*)_ctx)->stmtfile;

  global_lock(_g11_watched_lock); global_lock(_g11_obj_freed_lock);
  bb1:  
    _l13_r7 = _assoc_contains_key(_l8_addr, &(_g11_watched));
    _l13_r10 = _l13_r7;
    if (_l13_r7) goto bb8; else goto bb9;
  bb8:  
    _l13_r8 = "malloc.*";
    _l13_r9 = (strglob(_l13_stmtfile, _l13_r8) != 0);
    _l13_r10 = _l13_r9;
    goto bb9;
  bb9:  
    _l14_r14 = _l13_r10;
    goto bb3;
  bb3:  
    if (_l14_r14) goto bb4; else goto bb2;
  bb4:  
    _l14_r5 = _l8_addr;
    _l14_base_addr = _assoc_get(&(_g11_watched), _l14_r5);
    _l14_r11 = _l14_base_addr;
    if (_assoc_get(&(_g11_obj_freed), _l14_r11)) goto bb5; else goto bb6;
  bb5:  
    _l14_r12 = "use-after-free on %x (accessed %x) at %s:%d\n"
    dr_fprintf(STDERR, _l14_r12, _l14_base_addr, _l8_addr, _l13_stmtfile, _l14_stmtline);
    goto bb6;
  bb6:  
    goto bb2;
  global_unlock(_g11_obj_freed_lock); global_unlock(_g11_watched_lock);
}
```

Figure 6.6: An example of source code for a probe handler generated from the `obj.access` probe in the use-after-free detector from Figure 6.1. (The generated code has been edited slightly for the sake of brevity.) Note that the code for checking filtering conditions (`bb1, bb3` in the generated probe handler) and the code for the probe’s body have been merged into a single function by the translation procedure. Temporary variables used to calculate the filtering conditions are ‘mangled’ by adding the `_l13_` prefix, while temporary and local variables used by the probe directive’s body are prefixed with `_l14_`.
• For each **global data declaration**, a C declaration of the corresponding data structure is generated. Declarations of scalar variables result in declarations of C global variables. A declaration of a record results in a series of C variable declarations, one for each subfield of the record. A declaration of an associative array results in a declaration of a data structure of the `table_t` data type provided by the runtime support layer. (The `table_t` type is typically an alias for a hashtable type provided by the target framework.)

Code for initializing each global data structure to its default value is added to the corresponding section of the plugin template’s initialization function (`et_pre_clo_init()` in the case of the Valgrind template in Figure 6.5).

• For each **function declaration**, a corresponding declaration of a C function is generated. The body of a generated C function begins with a list of **local data declarations**. For every variable in the function declaration’s local symbol table, the translator outputs a local data declaration. Initialization code for each local variable is included immediately after its declaration. After initializing local variables, the generated C function executes code translated from the statements of the function declaration’s body.

• For each **probe directive**, the translator generates a declaration of a probe handler function which executes code translated from the statements of the probe directive’s body.

A probe handler takes a reference to a *context data structure* as its parameter, and has return type `void`. A context data structure contains context and program state values which may be accessed by the probe handler. For every context or value accessed within a probe directive, the translator adds a local variable to the probe handler. The initialization code for the variable copies the appropriate value from the probe directive’s context data structure.

Figure 6.6 shows an example of a probe handler generated for the `obj.access` probe from line 9 of the **use-after-free** program discussed at the beginning of the chapter (see Figure 6.1).

The statements within each probe directive or function body are translated to C statements as follows:

• Constant operands are translated into equivalent constant expressions.

• Variable operands referring to simple variables or record subfields are translated into equivalent C variable references.

• Variable operands referring to elements of an associative array are translated into calls to the runtime support layer’s `_assoc_get()` function.

• Instructions that incorporate a destination operand (CALL and MOV) are translated into assignment statements if the destination operand is a simple variable or record subfield, and into calls to the runtime support layer’s `_assoc_set()` function if the destination operand is an associative array element.

• CALL operations to built-in functions are translated either into built-in C operations, or into calls to functions provided by the runtime support layer. For example, a call to the `+` built-in function with operands `a` and `b` is translated into an expression `a + b`, while a call to the `printf` built-in function is translated into a call to a C `printf` function provided by the runtime support layer.

• CALL operations to user-defined functions are translated into calls to the C functions generated from the corresponding function declarations.
• Control-flow instructions are translated into equivalent structures of labels and goto statements, producing a C function whose control-flow structure mirrors the structure of the basic-block intermediate representation.

E language identifiers (i.e. the names of local variables, global variables, functions, and probe handlers) that appear in generated code may have identical names to identifiers declared by the target framework and runtime support layer. To avoid naming conflicts, E language identifiers in generated C code are combined with a prefix corresponding to the scope of the identifier. This technique is known as mangling, and is commonly employed in language translation [6]. For example, a local variable called var will be named in the generated source code by an identifier such as _l14_var. Here, the automatically generated prefix _l14_ corresponds to the original declaration where var has its scope.
Chapter 7

Evaluation

This chapter presents a performance evaluation of the E implementation on programs from the SPEC2000 benchmark suite. In addition, this chapter presents a comparison of the performance of E with the performance of the existing Valgrind Memcheck instrumentation plugin. Memcheck is a prebuilt plugin for Valgrind which observes memory operations within the target program in order to detect a variety of common memory errors (such as use-after-free errors, use of uninitialized memory, and memory leakage). Memcheck is extremely widely used and may therefore be considered a baseline for acceptable performance of an instrumentation plugin.

Any observation of a target program introduces a degree of slowdown (execution time overhead) in the target program's execution as well as a degree of memory overhead. While the target program executes, additional computing resources are required to perform the observation, store the observed data and report it to the developer. The E language allows the developer to easily express complex observations such as origin tracking. These observations are useful for debugging, but may add significant overhead to the target execution. In a scenario where the overhead is very large or the target program is very complex, it may no longer be practical to run an observation on the target program. Therefore, performance overhead is a major factor determining whether or not an observation tool is useful in a realistic scenario, and it is important to quantify this overhead and find ways to reduce it.

7.1 Benchmark Programs

The goal of this evaluation is to measure the performance overhead introduced by observation programs written in E. As discussed below, the E programs chosen for the evaluation perform observations whose overheads are likely to be representative of a wide variety of E observations written for the purpose of debugging.

Figure 7.1 contains fcalls, a function call tracing program which prints a message at the entry and exit points of each function call. To illustrate the operation of the fcalls program, Figure 7.5 contains widget.c, a simple C program, and Figure 7.6 shows a portion of the output which results from running fcalls with widget.c as a target program.

Figure 7.3 contains memwatch, a variation of the use-after-free program from the earlier chapter (§6) which also counts the number of accesses to every memory object allocated by the target program. The memwatch program follows the same strategy as the use-after-free program, as described at the beginning of §6. Figure 7.7 shows the output which results from running the memwatch program with widget.c as a target program.
global level = 0

probe fn.entry {
    printf("--> %d %s\n", level, name)
    level++
}

probe fn.exit {
    printf("<-- %s\n", name)
    level--
}

Figure 7.1: fcalls, a simple function call tracer written in E.

global count = 0
global return_count = 0

 probe fn.entry { count++ }
 probe fn.exit { return_count++ }

probe end {
    printf("calls %d\n", count)
    printf("returns %d\n", return_count)
}

Figure 7.2: benchmark-fprobes, a version of the function call tracer with a reduced number of printf calls.

shadow watched
array obj_accesses
array obj_freed

probe obj.alloc {
    for (i = addr; i < addr+size; i++) watched[i] = addr
    obj_accesses[addr] = 0
    obj_freed[addr] = 0
}

probe obj.access: addr in watched {
    base_addr = watched[addr]
    obj_accesses[base_addr]++
    if (obj_freed[base_addr])
        printf("use-after-free on %x at %x!\n", base_addr, addr)
}

probe obj.free {
    obj_freed[addr] = 1
}

probe end {
    foreach (addr in obj_accesses) {
        printf("%x -- %d accesses", addr, obj_accesses[addr])
        if (!obj_freed[addr])
            printf("", not freed!")
        printf("\n")
    }
}

Figure 7.3: memwatch, a simple use-after-free detector written in E which also counts memory object accesses.
shadow watched
array obj_accesses
array obj_freed

probe obj.alloc {
   for (i = addr; i < addr+size; i++) watched[i] = addr
   obj_accesses[addr] = 0
   obj_freed[addr] = 0
}

probe obj.access: addr in watched {
   base_addr = watched[addr]
   obj_accesses[base_addr]++
}

probe obj.free {
   obj_freed[addr] = 1
}

Figure 7.4: benchmark-memprobes, a simplified version of the use-after-free detector without printf calls.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
typedef struct {
   int x;
   int y;
} widget_t;

widget_t *create_widget(int x, int y) {
   widget_t *w = (widget_t *) malloc(sizeof(widget_t));
   w->x = x; w->y = y;
   return w;
}

#define WNUM 32
widget_t *line[WNUM];

void poke_1(int n, int k) {
   fprintf(stderr, "%d poke %d --> %d %d\n",
           k, n, line[n]->x, line[n]->y);
}

void main()
{
   srand(time(NULL));
   fprintf(stderr, "sizeof widget is %d\n", sizeof(widget_t));
   int i, k = 0;
   for (i = 0; i < WNUM; i++)
      line[i] = create_widget(rand() % 100, rand() % 100);
   for (i = 0; i < WNUM; i++)
      if (line[i]->x > 50) {
         poke_1(i, k); k++;
      }
   // Now free the widgets:
   for (i = 0; i < WNUM; i++) {
      free(line[i]);
      if (i == 3) // generate a use-after-free error:
         fprintf(stderr, "access %d @%x\n", line[i]->x, &line[i]->x);
   }
}

Figure 7.5: widget.c, a simple C program which allocates memory objects and contains a use-after-free error.
Chapter 7. Evaluation

--1707-- ETrace-gen, Valgrind client automatically generated by ETrace
--1707--
--1707-- Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
--1707-- Command: ./widget
--1707--
--1707-- --> 0 _dl_start
--1707-- --> 1 _dl_setup_hash
--1707-- <-- _dl_setup_hash
--1707-- --> 1 _dl_sysdep_start
--1707-- --> 2 brk
...
--1707-- --> 50 poke_1
--1707-- --> 51 fprintf
--1707-- --> 52 vfprintf
--1707-- --> 53 buffered_vfprintf
--1707-- --> 54 vfprintf
--1707-- --> 55 strcnnul
--1707-- <-- strcnnul
--1707-- --> 55 __IO_default_xspu
--1707-- <-- __IO_default_xspu
--1707-- --> 55 _itoa_word
--1707-- <-- _itoa_word
--1707-- --> 55 __IO_default_xspu
--1707-- <-- __IO_default_xspu
...
--1707-- <-- vfprintf
--1707-- --> 54 __IO_file_xspu@@GLIBC_2.2.5
--1707-- --> 55 __IO_file_overflow@@GLIBC_2.2.5
--1707-- --> 56 __IO_do_write@@GLIBC_2.2.5
--1707-- <-- __IO_do_write@@GLIBC_2.2.5
--1707-- --> 56 __IO_file_write@@GLIBC_2.2.5
--1707-- --> 57 write
--1707-- --> 58 __write_nocancel
0 poke 0 --> 74 20
--1707-- <-- __write_nocancel
--1707-- <-- __IO_file_write@@GLIBC_2.2.5
--1707-- <-- __IO_file_xspu@@GLIBC_2.2.5
--1707-- <-- buffered_vfprintf
--1707-- <-- vfprintf

Figure 7.6: A portion of the output produced by running the fcalls E program on widget.c.
Figure 7.7: The output produced by running the `memwatch E` program on `widget.c`.
The `memwatch` program observes every memory access in the target program. The overhead of this observation represents a realistic upper bound on overheads of debugging observations, because memory operations are likely to be the most frequently-occurring events observed during debugging, and therefore the most computationally expensive operations to observe. Many debugging observations, such as the origin-tracking example program in §4.2, will only observe a small subset of the memory operations occurring in the target program. (In the case of the origin-tracking program, only writes to `ma_table` objects were observed.)

Because both `fcalls` and `memwatch` output a large amount of information, a large proportion of the overhead they introduce will come from formatting and printing messages. Therefore, in addition to `fcalls` and `memwatch`, the evaluation includes the programs `benchmark-fprobes` and `benchmark-memprobes`, shown in Figures 7.2 and 7.4. These programs observe the same events and maintain the same external data structures as `fcalls` and `memwatch` respectively, but do not output any messages except at the every end of the program’s execution.

Comparing the overhead of both `memwatch` and `benchmark-memprobes` on the same execution allows the overhead of printing messages to be separated from the overhead of performing the actual observation. The same is true for `fcalls` and `benchmark-fprobes`.

Note that the `fcalls` program produced an unreasonably long running time for large target programs. A comparison to the runtime for `benchmark-fprobes` on the same programs showed that this was solely due to the extremely large number of messages printed by `fcalls`, and not due to the overhead of instrumenting the function calls themselves. Thus, results for `fcalls` are omitted from the evaluation.

The evaluation measures the execution time overhead introduced by the E example programs `memwatch`, `benchmark-fprobes`, and `benchmark-memprobes` to the executions of integer programs from the SPEC2000 suite.

### 7.2 Experiments and Results

The computer used for the evaluation was an Intel Core i5-3750k CPU with four 3.4GHz cores\(^1\) and 8GB of RAM. The target programs for the evaluation were integer benchmarks from the SPEC 2000 suite. The following observation tools and settings were used for the evaluation:

- **baseline.** The target program was executed without any observation tools in order to a baseline to estimate overhead against.
- **vg-none.** The target program was executed under Valgrind with the ‘nulgrind’ plugin, which does not perform any instrumentation. This measurement gives the absolute smallest runtime possible for the target program under Valgrind.
- **memcheck.** The target program was observed with the Memcheck memory debugger, a prebuilt instrumentation plugin for Valgrind.
- **dr-fprobes.** The target program was observed with the `benchmark-fprobes` observation tool using the DynamoRIO implementation of E.
- **vg-fprobes.** The target program was observed with the `benchmark-fprobes` observation tool using the Valgrind implementation of E.

\(^1\) Since the SPEC 2000 programs are single-threaded, only one processor core was used to run the benchmarks.
Figure 7.8: Comparison of different observation tools for five SPEC2000 integer benchmarks.

- **vg-memwatch.** The target program was observed with the memwatch E observation tool using the Valgrind implementation of E.

- **vg-memprobes.** The target program was observed with the benchmark-memprobes E observation tool using the Valgrind implementation of E.

Valgrind’s Memcheck plugin is an important baseline for comparison because it is very widely used, which implies that its performance overhead is considered by developers to be reasonably small for most target programs. In addition, because the Memcheck implementation observes every memory access by the target program, the nature of the observation is roughly similar to that of the memwatch program.

These observation tools and settings were run on the SPEC2000 integer benchmarks 164.gzip, 176.gcc, 181.mcf, 256.bzip2, 300.twolf. When a benchmark consisted of multiple executions of the target program on different inputs, the different executions were performed in sequence for each observation tool. The runtimes of the different executions were added together to produce a total benchmark runtime for each observation tool. Figure 7.8 shows the resulting runtimes.

In general, the memwatch script introduces a several-fold additional execution time overhead in comparison to Memcheck. Because both memwatch and Memcheck must observe every memory access, they instrument the target program with similar frequency and can be considered to perform similar observations. There is thus a significant gap between the performance of Memcheck and the Valgrind observation plugin generated from memwatch, even though both use the same underlying framework. This indicates the potential to improve the performance of memwatch by incorporating additional optimizations into the translator or runtime support layer.
Figure 7.9: The effect of reducing the number of watched objects on the overhead of memwatch using the Valgrind implementation of E, as measured on the 176.gcc SPEC 2000 benchmark. “1/N-watchpoints” denotes that only 1 in every N objects allocated by the target program were added to the watched array.

In order to determine the source of the additional overhead for memwatch, the following modified versions of the memwatch observation were executed using the 176.gcc benchmark as the target program:

- **empty-handler**: The same memwatch program was used with a modified runtime support layer whose event handler for obj.access was replaced by an empty function. Thus, the runtime support layer does not call the probe handler for obj.access at any point in time.

- **no-watchpoints**: A modified version of the memwatch program was used which instruments every memory access, but does not observe any memory objects. This was done by replacing the obj.alloc probe handler with an empty block. Thus, no values were added to the watched array and the filtering condition for the obj.access probe always returned false.

- **1/2-watchpoints**: A modified version of the memwatch program was used which only adds to the watched array 1/2 of all the objects allocated by the target program. This was done by declaring a global variable num_watched and adding the following code to the start of the obj.alloc probe handler:

  ```
  num_watched++; if (num_watched % 2 != 0) return
  ```

- **1/10-watchpoints, 1/100-watchpoints, 1/1000-watchpoints**: Modified versions of the memwatch program were used which only add to the watched array 1/10, 1/100, and 1/1000 of the objects allocated by the target program.

In addition, the Valgrind memcheck tool and an unmodified version of the memwatch program were executed.

The results of the experiment are given in Figure 7.9. When the number of objects added to the watched array was reduced to 1/2, 1/10, 1/100, and 1/1000 of the total number of objects allocated by the target program,
the overhead of the \texttt{memwatch} observation approached the overhead of the version of \texttt{memwatch} which does not set any watchpoints. This suggests that observations which observe writes to a subset of memory objects for the purpose of debugging (such as the origin tracking program in §4) will experience a much smaller overhead compared to the \texttt{memwatch} observation.

There is still a performance gap between the \texttt{no-watchpoints} program and Valgrind Memcheck. This occurs because the E runtime support layer for Valgrind uses a relatively simplistic version of Valgrind’s shadow memory implementation. On the other hand, Valgrind Memcheck uses a separate shadow memory implementation which incorporates additional optimizations not present in other Valgrind plugins [16]. Some of these optimizations, such as the use of a compressed shadow-memory representation, apply to all shadow memory operations and could be incorporated into the E implementation. Other optimizations are specific to the analysis performed by Memcheck. One Memcheck-specific optimization allows a range of consecutive entries in shadow memory to be efficiently set to the same value, in order to allow Memcheck to model operations such as memory allocation and copying of memory objects without needing to perform many individual shadow memory read and write operations. This optimized range-setting functionality could be exposed directly in the E language in the form of the array range-setting operation \texttt{a[b:c] = val} (discussed in §5.3.1).

To investigate the possibility of additional optimizations, the E runtime support layer was modified to include a shadow array implementation adapted from Memcheck’s shadow memory. This implementation was based on Memcheck’s compressed shadow memory representation, but excluded optimizations such as fast range-setting support which are specific to the Memcheck analysis. The experiment of Figure 7.9 was repeated with this modified runtime support layer, yielding the results in Figure 7.10. The original version of the \texttt{memwatch} program only experienced a modest speedup due to the optimized shadow memory representation. On the other hand, versions of \texttt{memwatch} which created few or no watchpoints experienced a significant speedup which resulted in execution time overhead only slightly greater than that of Memcheck. There are likely to be further possibilities to improve the performance the E language.
Chapter 8

Comparison to Related Work

A detailed description of existing debugging approaches was given in an earlier chapter of this work (§2). This chapter contains a further discussion of the differences between the designs of the E and SystemTap languages. SystemTap has been chosen as a basis for comparison because in terms of design concepts SystemTap is the observation tool most closely related to E. The design of E is based on syntax elements similar to SystemTap, namely events and probe declarations. However, a comparison of the two languages will highlight the fact that E introduces ideas that are not present in existing observation tools.

8.1 Model of State Transitions and Events

One of the major differences between the SystemTap and E languages lies in their approach to the concept of events and to the expression of this concept to the developer. SystemTap and E are both designed around the concept of probe declarations which associate an event with a handler block. Events in SystemTap correspond closely to observation mechanisms available in the Linux kernel and do not introduce any additional abstraction. Events in E are designed and expressed to the developer based on the semantics of the target program. This design is independent of the underlying observation mechanisms. The implementation, which is based on heavyweight instrumentation, is concealed from the developer. Events in the E language are expressed to the developer in terms of the semantics of the target program’s state transitions rather than in terms of the requests required to observe these state transitions.

SystemTap includes a concept called a probe point which is roughly equivalent to an event expression in E. A probe point is an identifier combined with a set of configuration parameters. SystemTap includes a large library of probe point identifiers, each of which corresponds to a particular observation mechanism available in the kernel. For example, the SystemTap probe point `process("...").function("...")` identifies the beginning of a function call just as the beginning of a function call is identified by the built-in events `fn.call` or `fn.entry` in the E language. The probe point `process("...").function("...")` requires two configuration parameters (indicated in our example by "..."). When this probe point is used in a SystemTap probe declaration, the first and second configuration parameters must contain the name of a program and the name of a function within that program, respectively\(^1\). SystemTap implements observation of such a probe point by placing a breakpoint at the beginning of each function identified by the probe point. Because each probe point

\(^1\)A configuration pattern can also be a wildcard pattern which matches the names of several programs or functions. For example, the probe point `process("a.out").function("foo_*")` indicates all functions whose name starts with `foo` in the program `a.out`.
includes a limited number of configuration parameters, a large library of probe points is required to cover the various combinations of event occurrences that could occur in the target program.

In contrast to SystemTap, E provides a small number of built-in events which correspond to categories of state transitions defined according to the semantics of the target program. The E language provides the capability to combine built-in events with filtering conditions to identify combinations of event occurrences. Many of these combinations of event occurrences cannot be identified using a SystemTap probe point. To illustrate this, consider the event expression

\[ \text{fn.entry: module == "a.out", name is foo_*} \]

which is equivalent to the SystemTap probe point \( \text{process("a.out").function("foo_*")} \). This event expression could be extended with additional filtering conditions to become an event expression

\[ \text{fn.call: module == "a.out", name is foo_*,}$flags != 0, \text{stmt is worker.c:*} \]

which only matches calls to functions named \( \text{foo_*} \), where the function is called from a statement in \( \text{worker.c} \) and where the function parameter \( \$flags \) satisfies a particular condition. The corresponding SystemTap probe point does not contain any additional configuration parameters that could be used to specify these restrictions as part of the probe point.

### 8.2 Hierarchy of Context Data

The SystemTap and E languages differ in the means by which they provide access to the current state of the program.
Each SystemTap probe point provides an associated set of variables. These variables correspond to program state values computed from the program state at the event’s occurrence. Information about the broader context within which an event occurs must be obtained through calls to library functions provided by SystemTap. For example, the library function ppfunc() gives the name of the current function, similarly to the E context value fn.name. Variables provided by probe points are computed at the start of the handler block’s execution, while values provided by library functions are computed each time the library function is called.

In contrast to SystemTap, E uses variables to expose information about both the program state and the broader context within which an event occurs. This enables information about outer state transitions to be accessed in a manner that is logical and uniform with how information about the current event is accessed. For example, in an insn event, program state values such as opcode provide information about the current instruction. Information about the function containing the current instruction can be accessed through the fn context, containing values such as fn.name. This scheme can be naturally extended to access information about outer function calls: the fn.caller context contains information about the next outer function call containing the current function call, fn.callers[2] contains information about the next outer function call after that, and so forth, arbitrarily far up the target program’s call stack.

Figure 8.1 illustrates the differences between how SystemTap and E provide access to the program state.

### 8.3 Observation of Frequently Occurring Events

The most crucial distinction in terms of practical use between E and SystemTap consists in the differing frequencies with which event occurrences may be observed.

SystemTap only supports observation of events that occur relatively infrequently. The kernel backend of SystemTap observes events by inserting breakpoints into the target program. The probe points that are based on this observation mechanism can only be used to observe events at a comparatively small proportion of the possible locations in the target program, and cannot be used to implement observation of events such as insn and obj.access that occur at every instruction or every memory access.

In contrast to this, E supports observation of events that occur at every memory access or even every instruction. The E language implements such frequent events using heavyweight instrumentation, which directly inserts code into the target program. This capability to observe frequently occurring events allows E to express observations of complex dependencies within the target program, and allows the development of E programs such as the origin tracking program used to solve the example scenario in §4 or the use-after-free example program given in §6. Supporting such observations allows E to be applied to a range of debugging scenarios not handled by SystemTap.
Chapter 9

Discussion

This chapter discusses further directions for the development of the E language and concludes the thesis by summarizing research findings obtained from the present work.

9.1 New Language Features

Libraries of Observations. The process of debugging using the E language could be made more convenient by implementing common observations in a reusable form as libraries of functions and events.

For example, the E program used to debug corrupted hashtables in the Python interpreter (Figure 4.2 in §4) has comparatively few parts which are specific to the problem being debugged (namely, the `check_corruption` function and the probe declarations that identify where memory objects of type `ma_table` are created). The remainder of the program consists of a set of declarations that are used to implement an origin-tracking scheme which could be adapted to a wide variety of purposes.

It should be possible to develop a language library which implements this origin-tracking scheme in a reusable fashion. After the E language has been used in practice to solve a wider range of debugging problems, it should be possible to identify and develop other observations whose components could be implemented as libraries.

User-Defined Events. In order to support implementation of common patterns of observation as reusable libraries, the E language must be extended with a declaration which allows new events to be defined in addition to the already existing predefined events.

A design for an event declaration is described in the language specification chapter (§5.7.4). An event declaration should contain the name of a newly defined event, an event expression in terms of an already defined (predefined or user-defined) event, and a handler block which can contain one or more `emit` statements. The event expression and handler block generate occurrences of the newly defined event as follows: at every occurrence of the already defined event which matches the event expression, the handler block is executed. Each execution of an `emit` statement within the handler block generates an occurrence of the newly defined event.

For example, the following event declaration defines a new event `access_watched`. An occurrence of the `access_watch` event will be generated at every `obj.access` event where the address of the memory access is present as a key in the `watched` array and the location of the memory access is not within the `malloc` library:

```
1 event access_watched <- obj.access:
2   !(stmt is malloc.*:*), addr in watched {
3     @base_addr = watched[addr].start_addr
```
This event can be used within a probe declaration just like a predefined event. For example, the following probe will observe all occurrences of access_watched that are located in the file myprogram.c:

```
probe access_watched: stmt is myprogram.c:* { 
    printf("obj.access addr=%x, base_addr=%x @myprogram.c:%d\n", 
            addr, base_addr, stmt.line_no) 
} 
```

The access_watched event could be provided as part of a library that also includes the following declarations:

```
shadow watched 
array objects 

cfunc watch_add(loc, addr, size) { 
    // Assign a consecutive run of addresses to be watched:
    watched[addr:addr+size].start_addr = addr 
    watched[addr:addr+size].origin = "<unknown>" 
    objects[addr].where_created = loc 
}
```

The library containing the declarations of access_watched, watched, objects and watch_add could then be used to write a variety of custom scripts that select memory objects and observe accesses to them, without needing to duplicate the library code in every script.

The E translator will process the event and probe declarations in the above example as follows. Suppose that the program has to be translated into a plugin for the Valgrind framework. The target framework description for Valgrind includes the obj.access event, which occurs at every target program memory access.

In the above example, the probe directive on line 7 is based on the user-defined event access_watched, and the runtime support layer for Valgrind does not include access_watched as a predefined event. The translator will combine the probe directive with the event declaration on line 1 by substituting the emit statement on line 4 of the event declaration with the entire body of the probe directive. The result will be a basic probe whose event expression only references the predefined event obj.access and does not contain any filtering conditions:

```
probe obj.access { 
    if ((!stmt is malloc::*)) && addr in watched) { 
        base_addr = watched[addr] 
        if (stmt is myprogram.c:*)
            printf("obj.access addr=%x, base_addr=%x @myprogram.c:%d\n", 
                   addr, base_addr, stmt.line_no) 
    }
}
```

This basic probe can be processed by the code generation pass in the same manner as other probes in the E language.
9.2 Optimizations

In comparison to prebuilt instrumentation plugins such as Memcheck, the current implementation of the E language introduces a significant slowdown. Prebuilt instrumentation plugins reduce the overhead of the observation using hand-designed instrumentation or custom-designed data structures.

This performance gap can be closed by incorporating optimizations into the E translator. Numerous possibilities exist for such optimizations. For example, the following two optimizations are likely to produce significant improvements:

1) **Static Filtering-Condition Extraction.** This optimization aims to improve the performance of probe directives whose filtering conditions can be evaluated statically at the time the target program is instrumented. It will generate an additional *static check function* for each probe directive that computes any filtering conditions which can be determined for an instruction prior to its execution and returns *true* if and only if all of these filtering conditions evaluate to *true*. This static check function will be used to determine which instructions of the target program should be instrumented.

For example, in the event expression `insn: fn.name == "foo", opcode == "idiv"`, the two filtering conditions can be computed prior to the execution of an instruction, since they check attributes of the instruction (such as the opcode of the instruction and the name of the containing function) which can be determined statically by examining the target program's code.

For each instruction in the target program, the generated plugin's basic block callback will execute the static check functions of all probe directives whose predefined events occur at that instruction. If all the static check functions return *false*, the basic block callback will not insert any instrumentation at that instruction. The result will be a reduction of the instrumentation that is added to the target program, and a corresponding decrease in the overhead of the observation.

2) **Dead Code Elimination.** A standard compiler optimization which removes dead code can be applied. Dead code is code that computes values that are not used by any subsequent statements. This optimization aims to improve the performance of code which contains event declarations. In situations where an event declaration computes program state values that are not used by a probe directive observing the declared event, the code computing unused program state values is dead code and can be removed.

9.3 Modular Observation Tools

One eventual goal of the E language design is to make possible the development of prebuilt observation tools which are structured modularly as libraries of event definitions. The developer would then be able to write event expressions that restrict the scope of the observations performed by these tools to a particular subset of the program.

Such modular instrumentation plugins may be useful in debugging large software programs which experience significant slowdown when subjected to comprehensive error checking across the entire execution, but may run much faster when the scope of the error checking is restricted, providing a performance advantage over prebuilt instrumentation plugins.

For example, when run on a compiler, Memcheck will instrument and report errors from all passes of the compiler. However, a modular analogue of Memcheck with settings which restrict the observations performed to only one pass of the compiler. This will reduce the amount of instrumentation and the overall time required to execute the compiler with the observation of the selected pass.
9.4 Alternate Backends

The event-based tracing model used in E allows observations to be specified independently of the underlying mechanism used to perform the observation. The implementation of E presented in this work is based on dynamic heavyweight instrumentation frameworks.

However, the E implementation could also be modified and extended to make use of other types of observation mechanisms.

**Static Instrumentation.** One alternate observation mechanism that could be used by the E implementation is static instrumentation in the context of a compiler. As discussed in the earlier background chapter (§2.3), static instrumentation tools such as AddressSanitizer and ThreadSanitizer run as compiler passes, inserting observations into the program as part of the compilation process. Such tools allow information about the program’s behaviour to be collected in production environments where setting up a dynamic instrumentation framework would be difficult. For example, dynamic instrumentation is not possible to perform on code that executes on GPUs, but instrumentation can be added statically while GPU code is being compiled.

In other respects, static instrumentation provides similar capabilities to dynamic instrumentation, and it would be relatively straightforward to adapt the E implementation to generate static instrumentation tools.

**Backends for Higher-Level Languages.** The basic model used by the E language is based on a set of state transitions which are present in almost all commonly-used programming languages. Even languages with unusual semantics (for example, Haskell) still have program executions that can be described in terms of state transitions such as function calls. It is likely to be possible and worthwhile to adapt the design and implementation of the E language to support observation of programs written in languages other than C/C++.

**Backends for Multilayer Systems.** Complex software systems may be composed of several layers, written in different programming languages based on different runtimes. Analysis and debugging of such systems is made more difficult by the fact that most observation mechanisms target only a single language or runtime layer, and can therefore only observe a subset of the events and state transitions in a multilayer system. For example, the Java Virtual Machine provides observation mechanisms such as JVMTI and agent instrumentation [34, 36], which are distinct from analogous mechanisms used to observe programs in the C/C++ runtime. It may be worth exploring whether it is possible to develop a backend for the E translator which produces a combined observation tool that can observe events within several layers of a system, using the appropriate observation mechanism for each component of the system depending on the language and runtime used to implement it.
Chapter 10

Concluding Remarks

This thesis introduced the event tracing language E, compared E to a range of existing debugging tools (as discussed in Chapters 2 and 8) and demonstrated the utility of the language for solving complex software problems (Chapter 4). The design of E introduces a new event tracing model based on an analysis of the structure of a program’s execution in terms of a small number of state transition categories (Chapter 3). The high-level design of the language and framework-independent design of the implementation open up a number of opportunities for making the E language applicable to an even wider variety of scenarios (Chapter 9). It will be interesting to polish and complete the implementation and put the work presented in this thesis into real-world practice.
Bibliography


