Tracerory - Dynamic Tracematches and Unread Memory Detection for C/C++

by

Jonathan Eyolfson

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Dynamic binary translation allows us to analyze a program during execution without the need for a compiler or the program's source code. In this work, we present two applications of dynamic binary translation: tracematches and unread memory detection.

Libraries are ubiquitous in modern software development. Each library requires that its clients follow certain conventions, depending on the domain of the library. Tracematches are a particularly expressive notation for specifying library usage conventions, but have only been implemented on top of Java. In this work, we leverage dynamic binary translation to enable the use of tracematches on executables, particularly for compiled C/C++ programs.

The presence of memory that is never read, or memory writes that are never read during execution is wasteful, and may be also be indicative of bugs. In addition to tracematches, we present an unread memory detector. We built this detector using dynamic binary translation.

We have implemented a tool which monitors tracematches on top of the Pin framework along with unread memory. We describe the operation of our tool using a series of motivating examples and then present our overall monitoring approach. Finally, we include benchmarks showing the overhead of our tool on 4 open source projects and report qualitative results.

Acknowledgements

First, I would like to thank my supervisor Dr. Patrick Lam. I am grateful for the opportunity to learn from him. He has an incredible amount of knowledge and experience which was always available to me. I couldn't have done it without his guidance.

I would like to thank my thesis committee members Dr. Derek Rayside and Dr. Lin Tan for reading my thesis and providing valuable feedback along with suggestions during their busy schedules.

Finally, I would like to thank my colleagues Aakarsh Nair, Xavier Noumbissi and Hang Chu for helping me improve my work.

Dedication

This thesis is dedicated to my parents, who have supported me throughout all my years.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this work, we use dynamic binary translation [7, 17, 19] to enable runtime monitoring of programs. Dynamic binary translation inspects a program's binary code as its loaded and allows changes in the code before execution. We present two applications of runtime monitoring: tracematches and unread memory detection.

Runtime monitoring enables developers to specify and monitor a wide range of program properties, including library usage constraints. An important class of library usage constraints can be expressed in terms of sequences of events on library objects. Tracematches [1] are a powerful notation for expressing runtime monitoring constraints over Java programs and libraries. In particular, they permit the specification of constraints in terms of regular expressions over events, defined in terms of AspectJ pointcuts [2], and enable developers to provide notification and recovery code to handle tracematch property violations.

Current implementations of tracematches (or properties similar to tracematches) monitor programs provided in the form of Java bytecode. Typically, they rewrite the Java bytecode to include monitoring code, which is then executed at runtime alongside the original program. Broadening the applicability of tracematches beyond Java programs would enable their use in many more situations than are currently possible. Traditionally, they allow users to easily specify safety properties about programs. In this work, we propose the use of dynamic binary translation to implement tracematches, and explain our prototype implementation. Dynamic binary translation enables the monitoring of sophisticated properties like tracematches, even if programs are written in languages such as C++ and only available in binary form. In addition, we allow 3 different operating modes. The first is the traditional mode, which allows users to easily specify satefy properties. The other

modes are more suited for usage properties.

Valgrind [19] is a dynamic binary translator which includes a tool that checks for common memory errors. These errors are as follows: accesses to unallocated memory, uninitalized memory, memory leaks, double frees and overlapping memory. In this work, we analyze another property—unread memory. This can come in two forms: memory which is completely unread and writes to memory which are unread. Unread memory may not indicate a bug in the software. However, they are another source of wasted resources. Our tool detects and reports unread memory for any binary/library given by the user.

1.1 Approach

Our dynamic binary translation approach has several advantages over program-rewriting approaches. First, whereas Java is generally relatively straightforward to compile, build systems for C++ and other languages can be quite difficult to work with. Our approach requires neither a compiler nor the program's source code. Note that tracematches are useful even in the absence of source code: for instance, they could be used in conjunction with third-party libraries to verify proper library usage. When the source code is available, for instance to the original developers during the development process, tracematches can be used to diagnose problems: if the binary includes line number information, then our tool can report the locations of tracematch property violations.

Our monitoring approach, for tracematches, inserts monitoring code into functions which match the tracematch specification supplied by the user. Using the specification, we construct a representation of the tracematch (in our case a finite state machine) and initialize our environment. The specification also allows the user to associate function calls with one or more objects. This allows sophisticated relationships between objects given a trace. Our monitoring code resolves any objects and creates a callback to our tool. Our tool then advances its current state, depending on the information from the monitor. Our tracematches have three operating modes: all, which ensures we take no unknown transition and the trace finishes in an ending state; only, which just ensures we take no unknown transition; and never, which ensures the trace does not reach the ending state. If any violation of the specification occurs, we report the violating trace along with debugging information (if present).

For unread memory, our monitoring tool inserts monitoring code into the standard allocation functions available in C/C++. These include `new` and the `malloc` family of functions. We only record memory allocations emanating from binaries/libraries specified by the user. The intuition behind this is that a developer only has control over memory they allocate. For C++ applications, we also monitor object constructors. This is so we can report the class of an object to aid in debugging. Next, we monitor all memory reads and writes. For all memory reads on currently allocated memory, we set the memory's block to be read. Also, if the memory location matches a write on the block, we set its state to read. For memory writes, as with allocations, we only record writes which happen in the developers binaries/libraries. For each write, we record the location and associate it with its memory block. We record locations so we can check that they are later read. If we overwrite a location which has an active write, we record that the previous write was not read. Finally, we monitor deallocation functions, so we can determine when a memory block was deallocated. When the program terminates, we report all memory which is either unread or contains unread writes. For each entry, we report the debugging information of the following: allocation location, deallocation location, class (if applicable) and any unread writes.

1.2 Results

To validate our results, we selected 4 open source projects to run using our tool. For each project, we ran it with dynamic tracematches and then unread memory detection. We also ran each project with only the dynamic binary translator present in order to determine the overhead solely from our tools.

First, we present a case study for one of the projects. The propose of the case study is to investigate real problems our tool can detect, and how our tool may help developers. We extract a usage constraint from the project along with an excerpt by a developer that violated the constraint. We recreated the bug and specified the constraint using our tool, to show we correctly the violation. For the unread memory detection, we used the same project. We discuss our findings, and how they can be used.

Next, we formulated tracematch specifications for all of the projects. The specifications came from the projects documentation, which showed common usage errors. We present our tracematch for each project, and run it using our tool. We report our overhead due to the dynamic binary translator and our tool.

We have a similar approach for unread memory. For each project, we run it with our unread memory detector and report our findings. We discuss possible causes of wasted memory and possible solutions. We also report our overhead due to the dynamic binary translator and our tool.

Finally, we present a summary of the overhead for our tool. We discuss the additional overhead on top of dynamic binary translation for both tracematches and unread memory detection.

1.3 Limitations

Our tool, presented in this thesis, has some limitations due to using dynamic binary translation. This section describes those limitations, and others present in our tool.

Dynamic binary translation itself imposes significant overhead, which may increase a program's runtime by two orders of magnitude. Also, since we may not have source code, we have to identify and follow conventions which may differ from compiler to compiler. During our testing, however, we found our tool can successfully resolve different calling conventions.

Another limitation is that, in order for the output of our tool to help developers identify the source of problems, we require the application to be compiled with debugging symbols. However, we can still use our tool to identify tracematch violations and unread memory without debugging information. In this case, we use the function name and memory offset in place of line numbers.

Finally, the unread memory detector may report a large number of false positives for C++ applications. This is because objects with virtual functions may have unread writes to its virtual table. We use debugging information in order to ignore these writes, which the developer cannot control. Without debugging information, we cannot determine which writes are to a virtual table.

1.4 Contributions

The contributions of this thesis include:

- **Unread memory detection.** We present a memory detector which ensures that all memory and memory writes are read at least once. The absence of reads for memory mean the memory is never used, hence wasteful; or may indicate a bug.
- **Runtime monitoring.** We present the concept for two types of runtime monitoring using dynamic binary translation. The first for finite-state properties using tracematches. The second for unread memory detection..
- **Implementation.** We present an implementation of tracematches and unread memory detection. Our implemenation is in C++, on top of the Pin dynamic binary translator.
- **Experimental results.** We applied our tools to 4 open source applications. We present experimental results which show the applicability of our approach. We present a case study which focuses on our tool's usability. Finally, we discuss our quantitative and qualitative results. We measure the overhead of each tool and compare the results to just having the dynamic binary translator present.

1.5 Organization

The remainder of the thesis is organized as follows.

In Chapter 2, we present a overview of our tool and approach. First, we motivate the problem with an example and present a high level overview of our tool. We then discuss our overall approach using a series of examples.

In Chapter 3, we present the details of our implemenation. We begin by discussing our tool's framework. We then discuss our monitor for tracematches. Our description includes details of the object resolution needed to handle C++ applications. We then explain our handling of multiple objects, as may occur in tracematch specifications. Next, we discuss our unread memory detector. We present the details our instrumentation and monitoring.

In Chapter 4, we present our experimental results. We present a case study that outlines the usage of our tool. We explain how both tracematches and unread memory detection reports help developers. Next, using 4 open source applications we benchmark our tool. We

present results by running the programs with the dynamic binary translator, tracematches and unread memory detection.

We present related work in Chapter 5. Finally, we present our conclusions and future work in Chapter 6.

Chapter 2

Tracerory

Tracerory implements dynamic analyses to verify tracematch specifications and detect unread memory. Dynamic tracematches allow the user to specify a series of function calls that should or should not happen. For instance, the user may specify usage constraints for a library. The unread memory detection allows users to check for unused memory in their binaries or libaries by specifying which images to watch. We detect any memory allocations from watched images which are never read by the program, and also any memory writes from watched images which are never read.

Figure 2.1 shows the workflow of our tool. We have two inputs: the user specifications and the executable to run. We use dynamic binary translation to set up a monitor based on the inputs. After the monitor is set up, we execute the program and continue monitoring it until the program exits. We then record any violations of the specifications.

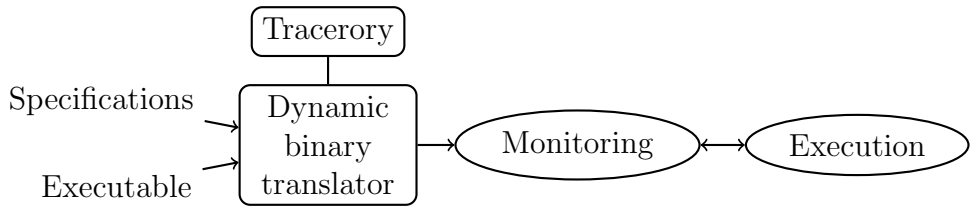

Figure 2.1: Tracerory workflow.

We provide a motivating example in the next section to demonstrate the application of our tool. We explain both analyses by working through examples and explaining our tool at a high level.

2.1 Motivating Example: Tracematches

A tracematch consists of a set of symbols, a regular expression over the alphabet formed by the set of symbols, and an operating mode which the regular expression should follow. One such operating mode is `all` which means the regular expression should never take a transition which does not exist, and the program should stop at an ending state. This mode works well to specify usage constraints that program executions should not violate. Traditional tracematches only execute code when the trace reaches an ending state.

Consider a program which converts RTF documents. Doucments may contain tables, and we may want to verify that the program closes any opened table and does not manipulate the cells after it has closed. Figure 2.2 presents a tracematch, applicable to the free AbiWord[1] word processor, which verifies proper table usage.

```
tracematch TableUsage (IE_Imp_RTF rtf)
{
  sym open before target(rtf): IE_Imp_RTF::OpenTable;
  sym close before target(rtf): IE_Imp_RTF::CloseTable;
  sym handle before target(rtf): IE_Imp_RTF::HandleCell;

  (open handle+ close)*
  { all }
}
```

Figure 2.2: AbiWord tracematch for table usage.

Note that while this particular tracematch definition contains only one variable, tracematches may, in general, bind multiple variables. This enables developers to state and verify sophisticated relationships between multiple objects.

Symbols. Our tracematch implementation supports symbols, or events, defined using the notion of a pointcut, drawn from aspect-oriented programming. We support a specific subset of more general pointcut languages; we chose this subset because it is sufficiently expressive for many useful tracematches. In this case, the `before` keyword indicates the symbol occurs at the entry of the function

[1] `http://abiword.org`

The alphabet of the example tracematch consists of three symbol declarations: `open`, `close` and `handle`. Consider the `open` symbol. The declaration states that `open` occurs before a call to the function `IE_Imp_RTF::OpenTable` and binds the tracematch variable `rtf` to the receiver object (target) of the call. Symbols may simultaneously bind more than one object.

Regular expressions. The set of declared symbols induce a so-called parametrized trace in the program's execution. We say that the trace is parametrized because each symbol comes with a set of variable bindings. After choosing specific values for the tracematch variables, it is possible to project a parametrized trace down to a set of bare symbols, discarding symbols which do not match the chosen values. A program's parametrized trace matches the regular expression whenever there is some set of variable assignments which causes the projected trace to match the regular expression.

For instance, the parametrized trace

$$\texttt{open}(\texttt{rtf} = o_1)\ \texttt{handle}(\texttt{rtf} = o_1)\ \texttt{close}(\texttt{rtf} = o_2)$$

does not match the example tracematch, because of the different variables, while the following trace

$$\texttt{open}(\texttt{rtf} = o_1)\ \texttt{handle}(\texttt{rtf} = o_1)\ \texttt{close}(\texttt{rtf} = o_1)$$

does. These bindings allow us to verify that the property holds for the RTF document.

With this tracematch specification we can run the program and our tool verifies the proper table usage property. If the property does not hold, we output the trace along with the debugging information of the symbol caller. Consider if we created an object and called open then handle and terminated the application. We would report the call to open and the caller's line number and the call to handle with its caller's line number. This allows developers to debug tracematch violations.

2.2 Specifications

In this section we discuss the specifications which the user may input for both unread memory detection and tracematches. The specifications are read from an input file given by the user. For our tool the input file must conform to the grammar in Figure 2.3.

```
input = [ unread memory ]
        { tracematch }

unread memory = "unread" "memory" "{"
                { image_name ";" }
                "}"

tracematch = "tracematch" name "(" [ variables ] ")" "{"
             symbol { symbol }
             regex
             "{" mode "}"
             "}"

variables = name name { "," name name }

symbol = "sym" name kind [ "target" "(" name ")" ]
         ":" function_name ";"

kind = "before" | "after" [ "returning" "(" name ")" ]

regex = regex_or { regex_or }

regex_or = regex_unary { "|" regex_unary }

regex_unary = regex_term ( "*" | "+" | "?" | "[" number "]" )

regex_term = name | "(" regex ")"

mode = "all" | "only" | "never"
```

Figure 2.3: Grammar for Tracerory specifications in EBNF.

The unread memory rule simply enables the unread memory detection portion of our

tool. It uses the specified image names as the binaries to monitor for unread memory detection.

Any occurrence of the tracematch rule enables the tracematch portion of our tool. A tracematch consists of variables, symbols and regular expression and an operating mode.

As discussed previously, a tracematch may have any number of variables. A variable, in the specification, must contain a class name, followed by a variable name.

A symbol is a name, followed by a kind (before or after), an optional target/returning variable and a function name. The kind controls whether the symbol matches on function entry or exit. The target controls whether or not to use the variable (which represents an object, determined by "this") as part of the match (binding) for the symbol. The returning preforms the same function, only we determine the object by the return value of the function.

The regular expression follows standard regular expression syntax. We support the following operations: `|`, `*`, `+`, `?`, `[]`, `()` and the implicit concatenation.

Finally, we allow the user to set the operating mode for the tracematch. As a reminder, the mode determines which violations of the regular expression to report. We discuss the modes in more detail in the next section.

2.3 Tracematches

We next present tracematches by means of a pair of examples. First, we present a tracematch which defines a safety property of C++ iterators (as implemented in the Standard Template Library). Our property enforces the constraint that whenever a container (e.g. vector) is modified, then all previously obtained iterators on that container become invalid. We therefore verify that programs never access invalid iterators.

```
tracematch UnsafeVectorIterator(vector v, __normal_iterator i)
{
  sym create_iter after returning(i) target(v): vector::begin;
  sym update_vec after target(v): vector::push_back;
  sym access_iter before target(i): __normal_iterator::operator*;

  create_iter access_iter* update_vec+ access_iter
  { never }
}
```

Figure 2.4: Tracematch for unsafe iterator usage.

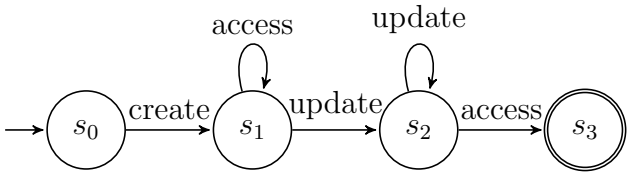

Figure 2.5: Finite state machine for tracematch from Figure 2.4.

Figure 2.4 shows our encoding of this property as a tracematch, and Figure 2.5 shows the corresponding finite state machine. Because we target programs compiled against the GNU standard C++ library, we look for instances of the `__normal_iterator` class. The tracematch declares two variables: the iterator, and its associated collection. The tracematch also declares three symbols. The first symbol, "`create_iter`", matches a vector's `begin` function, which returns a new iterator. Next, "`update_vec`" matches when the vector is modified; in this case, we are interested in the `push_back` function. Finally, the "`access_iter`" symbol matches the iterator's dereference operator—our definition of an access to an iterator is execution of the dereference operator.

The regular expression states that the trace will match if we create an iterator, access it zero or more times, update the vector and attempt to access it again. A match implies that the program has dereferenced an invalid iterator, which violates the iterator's contract. In the event of a complete match under the "`never`" mode, this tracematch prints the violating trace along with its debugging information.

To monitor this particular property, the runtime monitor must bind together the vector and iterator: a dereference of an iterator i is only invalid if i has been previously invalidated

by an update to its associated vector v. Updates to vectors $v' \neq v$ should not invalidate iterator i; conversely, however, iterators i' which are also associated with v should also be invalidated by writes to v.

2.3.1 Monitoring

In this section, we discuss Tracerory's monitoring for tracematches. Recall the high-level overview of our tool in Figure 2.1. Tracerory takes two inputs: a binary to be instrumented, and a set of tracematches. It first parses the tracematches and determines which functions should be monitored. It then intercepts these functions, using the dynamic binary translator, to insert monitoring code into the binary. This results in calls back to Tracerory; these calls update the state of the runtime monitor.

While preparing for execution, Tracerory first reads the tracematch specification, so that it knows what to monitor. It also creates data structures to track states of the various objects involved in partial matches. Finally, it prepares a set of callbacks to be inserted into the running program during the instrumentation phase, as alluded to above.

We represent the tracematch specification using a variable table, a class table, a symbol table, and a finite state machine (representing the regular expression). Unlike a traditional symbol table, our symbol table only contains entries for tracematch symbols. The variable table maps the tracematch's variables to their types (classes) and records a unique identifier for each variable. For instance, the iterator example might give a class table mapping `v` to $\langle$`vector, 1`$\rangle$ and `i` to $\langle$`__normal_iterator, 2`$\rangle$.

The symbol table similarly maps symbol names to information summarizing symbols. In particular, it includes the kind of the symbol (**before**, **after**, or **after returning**), the function names (potentially containing wildcards) associated with the symbol, and a pointcut signature, which includes information on the targets, arguments, and returning values to be matched, including their types, as well as a unique identifier for the symbol. The `update_vec` symbol would be associated with **after**, `vector::push_back` and the fact that it binds the target `v` (of the call to `push_back`), which has type `vector`, along with a symbol identifier from the tracematch symbol table.

Finally, Tracerory creates a finite state machine from the regular expression using standard techniques. The symbol identifiers are the labels on the edges of the finite state machine.

At runtime, Tracerory maintains two data structures to track the progression of objects through states. First, it updates a binding table, which stores all currently-bound variables

and their states as in the FSM. New bindings are set to the initial state of the finite state machine. Tracerory also maintains an object map, keyed by object addresses. This map contains each object's type.

We create callbacks to be invoked upon symbol execution. When the program executes a callback (indicating that a finite state machine transition ought to occur), it passes the symbol's identifier and all relevant objects back to the runtime monitor.

We discuss three different kinds of callbacks: ones which handle no objects, ones which handle one object, and ones which handle two or more objects.

A callback with zero objects corresponds to a symbol s_0 without bindings. The runtime monitor must advance all bindings in the binding table with the information that s_0 has occurred.

For a callback for symbol s_1 with one object o, we look up o in the bindings and advance all of o's bindings. In the absence of any such bindings, we create a new binding at the initial state, associate the new binding with o, and advance the binding appropriately for an s_1 execution.

We use a similar approach for a symbol s_2 with two or more objects $\mathcal{O}$. In such a case, we first advance all bindings which also bind $\mathcal{O}$. In the absence of complete matches, if we find any bindings which bind a strict subset of $\mathcal{O}$ (leaving one or more variables unbound), we copy the partial binding, add the previously unknown object, and advance the trace. Finally, if no bindings at all match, we create a new binding with the initial state and advance that.

2.3.2 Iterator Example

Figure 2.6 presents a simple program which exercises the iterator tracematch. This program first creates a vector `my_vec`, modifies the vector, and then creates an iterator `iter` on `my_vec` using `vector::begin`. It then modifies the vector using `vector::push_back` (invalidating the iterator), and tries to dereference the now-invalidated iterator.

```
1  #include <vector>
2
3  using namespace std;
4
5  int main()
6  {
7    vector<int> my_vec;
8    my_vec.push_back(7);
9    vector<int>::iterator iter = my_vec.begin();
10   my_vec.push_back(42);
11   int x = *iter;
12   return 0;
13 }
```

Figure 2.6: Simple iterator client.

At runtime, the first callback to the monitor occurs at line 8. This callback creates a new binding and attempts to move it out of the initial state, but fails, as there is no transition for "update_vec" out of that state. We remove the newly created binding since it did not advance (the monitor does not change states).

The first interesting callback occurs at line 9. This callback receives the `my_vec` and `iter` objects, as well as the symbol identifier for "create_iter". Since there are no bindings, the monitor creates a binding using these two objects with its state set to the initial state. Next, the binding successfully advances to state s_1 and the monitor records the symbol name ("create_iter") and debugging information (line 9).

The call to `vector::push_back` at line 10 triggers a callback to the monitor with the symbol identifier and the (address of the) vector `my_vec`. The monitor looks up this address in the bindings and finds the binding $\langle$`my_vec`, `iter`$\rangle$ it just created. It therefore advances the binding in the monitor to state s_2 (ignoring the lack of "access_iter", which is allowed to match zero times).

Finally, the iterator dereference `*iter` triggers another callback for "access_iter" and iterator `iter`. The monitor looks up bindings for `iter` and finds the same matching binding, now at s_2. It therefore advances the binding to the final state. This indicates a complete match, which, according to the specification, should not happen. Figure 2.7 shows the output of Tracerory for this example.

```
Tracematch "UnsafeVectorIterator" occurred, listing trace:
create_iter (iterator_client.cpp:9)
update_vec (iterator_client.cpp:10)
access_iter (iterator_client.cpp:11)
```

Figure 2.7: Tracerory output for iterator client.

We see the trace does match the finite state machine in Figure 2.5. With our monitor, we correctly report the symbols that occurred along with the corresponding caller line numbers.

2.3.3 File Example

Another use of tracematches is to verify that proper usage patterns do occur during program execution. An example of this is proper file usage in the C++ library. We demonstrate this by specifying a property that enforces the constraint that any file opened is closed. To express this, we use the `all` operating mode. Figure 2.8 shows our tracematch for this property.

```
tracematch FileUsage(basic_ofstream o) {
  sym open before target(o): basic_ofstream::open;
  sym close before target(o): basic_ofstream::close;
  open close
  { all }
}
```

Figure 2.8: Tracematch for file usage.

The `all` operating mode ensures that all bindings reach the ending state of the regular expression when the object(s) are destroyed or the program completes. The `all` operating mode also includes the functionality of the `only` mode. This mode ensures the binding(s) match throughout the program execution. For example, if the program starts by closing a file, that would not advance the trace and our tool would report that. Figure 2.9 presents a simple client which uses C++ files.

```
1  #include <fstream>
2
3  using namespace std;
4
5  int main(int argc, char *argv[])
6  {
7      ofstream out;
8      out.open("tmp/file.txt");
9      out << "Hello" << endl;
10     return 0;
11 }
```

Figure 2.9: Simple file client.

During runtime the only callback made by the monitor is on line 8, which opens the file `out`. Since there are no bindings, we create a new binding for `out` and attempt to advance its trace from the initial state. This succeeds and records the symbol "open" on line 8. The program then writes to the file and exits without closing the file. Tracerory scans the bindings on the program exit and finds `out`, which is not at the ending state. Figure 2.10 shows the output of our tool.

```
Tracematch "FileUsage" did not hold, listing trace:
open (file_client.cpp:8)
```

Figure 2.10: Tracerory output for file client.

Using the output of Tracerory, we can easily observe that the open which occurs on line 8 was never closed. The `all` and `only` operating modes allow users to express usage constraints that must occur. Combined with the `never` operating mode, which ensures a trace never occurs, users are able to specify safety properties which should hold during execution. These modes allow users to express both positive and negative properties about tracematches.

2.4 Unread Memory

This section provides a high level overview of our unread memory detection. We restrict our unread memory detection to only monitor binaries and libraries (called images) that the user specifies. This allows the developer to only view memory which they are responsible for and can remedy.

For the given images we monitor any memory they allocate and verify two properties. First, that the memory (or object) is read at least once from any image. Otherwise the allocation is never used and should be reported. Second, that any writes done to fields of the memory are read. If not, the writes are never read, which again is unnecessary and should be reported.

We explain our monitoring for unread memory at a high level in the next section. Next, we present an example program which contains unread writes. We use this to explain our tool and describe our output.

2.4.1 Monitoring

Our monitor must know which images it should watch for allocations and writes. Figure 2.11 shows an example input which monitors a single image. Note that our tool can monitor multiple images if required by the user.

```
unread memory {
    /home/jon/bin/variable_writes;
}
```

Figure 2.11: Tracerory specification for unread memory.

The monitor records any allocation calls by instrumenting standard C/C++ allocation functions. We also instrument deallocation functions to determine when memory is released. Along with these functions we instrument object constructors to provide the class name for memory which represents an object.

Our monitor only records memory allocations which are made from a watched image. Since the image is in the watched list, we know they are responsible for the object. However, they may not be responsible for deallocation of the memory. Therefore, we allow the memory to be deallocated in any image.

To keep track of memory reads and writes, we instrument all instructions which read or write memory. Our monitor records the destination of the memory operation for each instruction.

For memory writes, similar to allocations, they must come from a watched image, as any other memory writes are outside of the developer's control. We record the location of memory write and mark it as written.

We record memory reads from any image. This is because memory might only used by an unwatched image (possibility an external library) and should not be reported as unread. The first read to a memory location sets its read flag to true, since it was read at least once. We also look at the memory locations within an allocated black. If we find a match, we mark that memory location as read.

2.4.2 Variable Writes Example

Figure 2.12 shows an example program which we use to demonstrate our unread memory detection. This program allocates two objects, performs some writes on them and deletes them.

```
1  class X
2  {
3  public:
4      X() { }
5      int x;
6  };
7
8  int main(int argc, char *argv[])
9  {
10     X* x1 = new X();
11     X* x2 = new X();
12     x1->x = 1;
13     x2->x = 2;
14     x1->x = 3;
15     delete x1;
16     delete x2;
17     return 0;
18 }
```

Figure 2.12: Simple program with variable writes.

First, the program allocates two memory regions for objects `x1` and `x2` on lines 10 and 11 respectively. The monitor initially marks these objects as unread.

Next, the program performs three writes to the objects. It writes to `x1` on lines 12 and 14. It also writes to `x2` on line 13. The monitor records the first two writes to `x1` and `x2` and marks this memory location as initially unread. The second write to `x1` overwrites the first write. Therefore, the first write cannot be read and the monitor records the write on line 12 as unread.

Finally, the program deletes the objects on lines 15 and 16. After an object is deleted, no further reads can be made to it. Therefore, if the object is completely unread, we report it. We also report if the object has any unread writes. Figure 2.13 shows the output for the program using Tracerory.

```
Unread Memory
=============
Class: X
Created: variable_writes.cpp:10
Destroyed: variable_writes.cpp:15
Unread: true
Unread Writes: 2
variable_writes.cpp:12
variable_writes.cpp:14

Class: X
Created: variable_writes.cpp:11
Destroyed: variable_writes.cpp:16
Unread: true
Unread Writes: 1
variable_writes.cpp:13
```

Figure 2.13: Tracerory output for variable writes.

For each object which is unread or contains unread writes, we report the following: the caller location that created and destroyed it, whether or not it is unread, and the number of unread writes, along with their debugging locations. Note that an object may not have a destroyed location, which indicates it was never deleted. Therefore our tool can also detect memory leaks in a program.

The output for the example correctly identifies that both objects are of type `X`, and reports that they are unread along with their unread writes. Using this information, the developer can identify the sources of potentially wasted memory.

Chapter 3

Implementation

In Chapter 2 we described the high level operation of Tracerory. To validate our design, we implemented it on top of Pin, a dynamic recompilation toolkit [17]. In this chapter, we describe our Pin-based implementation on Linux systems using the x86-64 instruction set. Our technique applies to other instruction sets as well, since Pin's API provides a layer of abstraction.

The goal of the instrumentation is to insert functions into a binary implementing our tracematch and unread memory analyses. To accomplish our goal, we instrument binaries at two levels: routine level, which sees all functions as they are loaded by the application; and instruction level, which may execute instrumentation code before each instruction. At each of these levels, we can inspect the code to determine whether or not we need to add instrumentation. Pin recompiles sections of binary code just-in-time before running instrumented versions of the code.

We present the simplified timeline for a Pin execution in Figure 3.1. Routine inspection is done after the executable loads and before it is run. We are able to add instrumentation functions at routine entry and exit points. Instruction inspection occurs just before instructions execute (while the program runs). For instructions we add instrumentation functions before the instruction executes. The instrumentation functions added at a inspection point execute when they are encountered. When we give an overview of the tools, we explain their inspection and instrumentation functions.

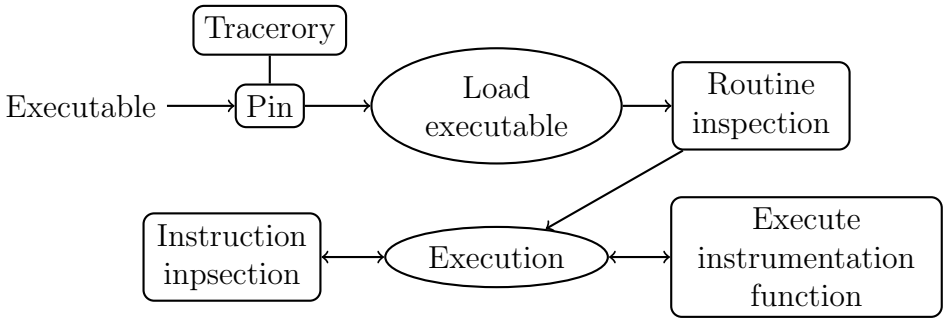

Figure 3.1: Timeline for Pin execution.

At the routine level we can instrument functions using the names included in a binary's symbol table. However, most C++ compilers, including the GNU compiler `g++`, "mangle" function names to ensure (for linking purposes) that similar function names do not clash. Our tool therefore unmangles these names, if necessary, to match them with the names from tracematch signatures. Furthermore, we can insert monitoring calls after entering or exiting a function using Pin. At the instruction level, we may also insert a monitoring call before or after an instruction (however, we never insert a call after). We use a combination of both levels for our implementation: routine level for function entry and exit points and instruction level for just-in-time instrumentation.

Section 3.1 describes the parser for our tool. Section 3.2 describes the common debugging information implementation for both the tracematches and unread writes tools. Finally, Section 3.3 and 3.4 describe the low level instrumentation and algorithms for both tools.

3.1 Parser

Figure 3.2 shows how the parser interacts with the rest of the program. The parser receives the input filename, containing specifications from the main program, and begins parsing. The main program tracks flags for unread memory and tracematches. We set these flags if we parse an unread memory rule or a tracematch rule.

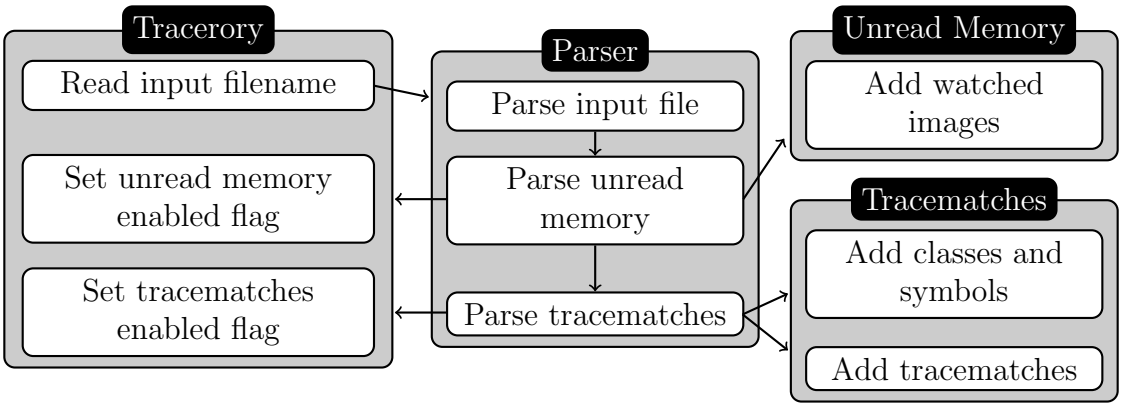

Figure 3.2: Parser overview.

For every image name in the unread memory rule, we add it to the watched image names in the unread memory tool. The unread memory tool does not require any additional input.

Tracematches consist of class names, symbols, a regular expression and a mode. When we encounter a tracematch, we first add all of the classes mentioned in the specification to the tracematch tool while parsing the variable rules. Next, while reading the symbol rule, we add all of the symbols in the tracematch to the tool. Both of these steps are required for proper instrumentation. For the regex rule, we collect tokens representing the regular expression in postfix notion. Finally, we read the mode (`all`, `only` or `never`) the user specified for the tracematch. Using tokens and the mode, we construct a tracematch which is later used by our tool.

3.2 Debugging Information

Both tools require accurate debugging information to help developers understand the source of errors (either unread memory or tracematch specification violations). In this section, we explain how we collect debug information to present it to the user. We use debugging information provided by DWARF [12], which can be read natively by Pin. This is the common debugging information format which GNU compilers include when the user specifies the `-g` flag.

For each monitored routine `x()`, we would like to know the debugging location and image name which invoked `x`. We therefore record this information at every call instruction. Figure 3.3 shows an overview. Note that this information is only valid at the entry point of a monitored function: because we store only one debugging location at a time, any subsequent calls in a routine will destroy the debugging information for that routine.

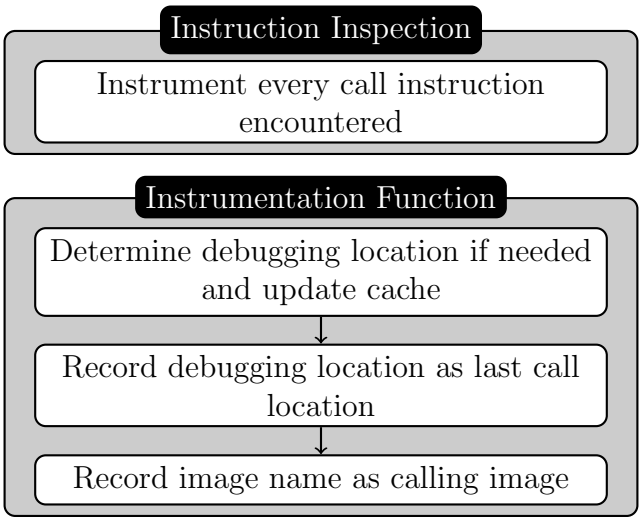

Figure 3.3: Debugging overview.

To determine the location of function calls within the binary, we first instrument the code at the instruction level, looking for `call` instructions. We add instrumentation which records the instruction pointer at each call instruction. Next, using the recorded instruction pointer in the instrumentation function, we determine and record the file and line number of each executed call instruction. If we do not have debugging information, we instead record the procedure name and its memory offset. We record this information in a hash map to

avoid the need to recalculate debugging information for the same instruction pointer. Note that all of the debugging information is computed on-demand.

Now, we are able to determine the caller information by reading the stored debugging location on routine entry. We must record every call instruction since it is not possible to determine which functions are called before execution. To see why, consider calls to dynamically linked libraries. In this case, the program will execute a call instruction to its procedure lookup table. The first time a function is called, the lookup table contains invalid information. This is because the procedure lookup table uses lazy binding, and only resolves external procedures as they are needed. Control flows to the dynamic loader, which resolves the location of the external procedure.

In addition to having to instrument every call instruction, we are faced with another challenge. Since control is passed to the dynamic loader, the operating system may use calls to other procedures to resolve the function address. These procedure calls would overwrite the last debugging location, causing it to be invalid. In order to prevent this,, we ignore all call instructions to the underlying operating system. For 64-bit Linux, this library is `/lib64/ld-linux-x86-64.so.2`.

Ignoring all system procedures and instructions does not negatively affect our tool. Most applications use the standard C/C++ API. This API then interacts with the operating system. We need not instrument the operating system, because we're still instrumenting the API. The operating system procedures and instructions mainly introduce noise for user level applications, which we are not interested in. Also, since we ignore the operating system, this allows our tool to be more easily used in other operating systems—we do not depend on any functionality specific to Linux.

3.3 Tracematches

Recall that tracematches consist of class names, symbols, a regular expression and a mode. Therefore, we must represent symbols, individual states in the regular expression with a debugging trace (history), and tracematch variable values (bindings). Our tracematch implementation consists of the following phases: instrumentation, building an NFA for the regular expression, advancing history and resolving bindings. Figure 3.4 shows an overview of the execution. First, we instrument all relevant routines. These instrumentation functions either determine active objects or resolve a symbol (and possibly its target and returning addresses). For symbols, we may create new bindings and then advance all matching histories using the regular expression NFA.

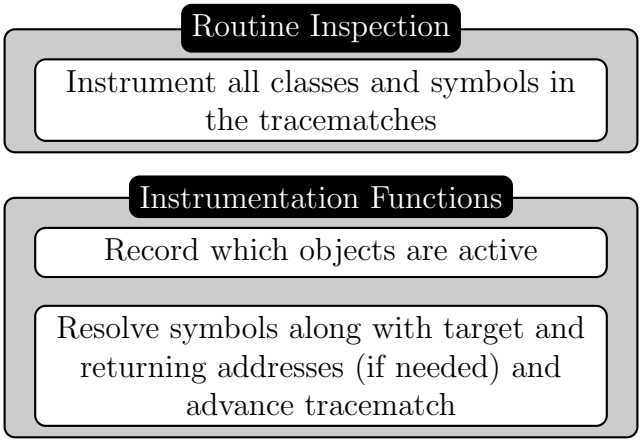

Figure 3.4: Tracematches overview.

We use two global data structures while parsing the input file. First, we maintain a bidirectional class mapping between a class name and its unique identifier. Also, we keep a list of tracematch symbols, as defined in the input file. Symbols contain the following: symbol name, tracematch identifier, symbol identifier, placement (before or after) and class and variable identifiers of the target and returning objects. All identifiers are represented as a unique number. These are required for the instrumentation phase outlined in Section 3.3.1.

During parsing, we maintain two mappings while parsing each tracematch. First, we have a variable mapping for a string to class and variable identifier. Next, we have a symbol mapping for a string to symbol identifier. Using these mappings we fill in the symbol entry, outlined above, while parsing the `symbol` rule in the grammar, shown in Figure 2.3.

While parsing regular expressions, we create a list of tokens representing each regular expression. The tokens are as follows: `symbol` (with identifier), `then`, `or`, `constant` (with number of repeats), `question`, `plus` and `star`. The `symbol` token represents a state transition for a particular symbol. We use `then` to represent the concatenation of two regular expressions. As a short-hand, `constant` is a concatenation of a regular expression for a specified number of repeats. The `or` token represents an alternation between two regular expressions. Finally, the remaining tokens represent the number of times to match a regular expression; `question` (zero or one), `plus` (one or more) and `star` (zero or more). These tokens are emitted in postfix notation by the parser. We outline their use in Section 3.3.2.

3.3.1 Instrumentation

The goal of the instrumentation is to call the monitor each time the program executes a symbol. In the tracematch specification, the points before and after correspond to the function entry and exit point respectively. The instrumentation is also responsible for resolving the target and returning addresses for a symbol, if they are present. We use these addresses to create bindings later in the program. This information is all we require for the rest of our tool to handle tracematches.

Since we have all of the symbol information from the input file, we can find the corresponding routines to instrument. Therefore, we iterate through all of the routine names, searching for a matching symbol name. When a match is found, we can instrument the entry and exit points of the function. This is done based on the location of the symbol, which is either `before` or `after`.

Before symbols. For `before` symbols we insert a call to an instrumentation function, which receives all of the identifiers for the tracematch/symbol/class/variable along with the first two arguments of the function. We use the arguments to determine the address of the target (if applicable), along with debugging information for the function, and call the appropriate tracematch to advance its state.

For x86-64 programs, the calling convention is to use registers `%rdi` and `%rsi` as the first and second argument, respectively. We found the first argument to constructors and destructors is always "this". However, this is not always true for other member functions; we found that functions for single variable objects on the stack used a different parameter passing convention depending on the presence of a destructor. Consider a call to the function `Foo::Bar`. Figure 3.5 shows the convention when class `Foo` has no destructor. In that case, the "this" object is located at the address `-0x10(%rbp)`, and we see that the

first argument (`%rdi`) is "this". The function is called as expected. The final line can be ignored for now.

On the other hand, Figure 3.6 shows the layout when there is a destructor. The object which is being called is located at address `-0x20(%rbp)`, which is actually the second argument (`%rsi`). Therefore, we must keep track of all active objects in order to determine which argument is "this". We are uncertain as to the specific reason why "this" may be the second argument.

```
lea     -0x10(%rbp),%rax
mov     %rax,%rdi
callq   <Foo::Bar>
mov     %eax,-0x20(%rbp)
```

Figure 3.5: Assembly code for single variable objects without destructor.

```
lea     -0x30(%rbp),%rax
lea     -0x20(%rbp),%rcx
mov     %rcx,%rsi
mov     %rax,%rdi
callq   <Foo::Bar>
```

Figure 3.6: Assembly code for single variable objects with destructor.

We maintain a mapping between addresses and class types for currently active objects. To keep track of active objects, we instrument calls to the constructor and destructor for every class defined in the tracematch. For these methods, the first argument is always the object being created or destroyed. We insert calls that pass the class identifier and address to be added or removed, respectively, in the active object pool of our tool. Every constructor will have a corresponding call to the destructor unless a destructor is not automatically generated by the compiler and the object is a stack variable. Stack variables can then be destroyed by checking the stack pointer and removing any objects which are below the current address (assuming the stack grows downwards, as is the case for x86-64 code generated by `gcc`).

Now, since we have both arguments to a member function, we check if the first argument corresponds to an active object for this class. If there is an entry we set this to the target. Otherwise we are in a case where the class has a destructor; the second argument is the target address.

After symbols. The `after` symbols always instrument the function exit point, and may instrument the entry point, if the symbol has a target. Recall that a target address is equivalent to "this" within the member function. The "this" object is passed as an argument to the method. However, the execution of the method may overwrite the value of the "this" object, so our tool captures the value of "this" upon entry.

At each symbol function entry, we insert a call to an instrumentation function, which passes it the class identifier of the target and the first two arguments of the function. Here, we resolve the target, record the debugging information for the location of the call, and push both onto our stack. We use the stack for temporary storage, so we can pop the values at the exit point. At the exit point of the call, we note all of the identifiers for the tracematch/symbol/variable, the class of the returning object and the return value.

Symbols containing `returning` in the tracematch signature present a difficulty. Functions do not always conform to a standard calling convention when returning an object. For most function calls returning an object, the return value is the address of the object. However, when the object contains a single variable, the function may instead return the value that is associated with the object. This is problematic, as we need the address of the returning object. The examples used above for the member functions demonstrate this point.

For instance, function `Foo::Bar` returns a `Foo` object. The expected case is shown in Figure 3.6, where `-0x30(%rbp)` is the address of the returned object. The return value of this function call is the address of `-0x30(%rbp)`. Figure 3.5 instead shows the case where the function returns the value of the single variable associated with the object. In this situation `-0x20(%rbp)` is the address of the returned object, which is not passed to the function. After the function call succeeds, the return value (`%eax`) is then written to the address of the returned object. This address is the actual address of the returned object, which is what we are looking for.

Therefore, in the instrumented exit function, we check if the return value corresponds to an active object. If it is active, we pop any target information (if needed) and call the appropriate tracematch to advance its state. Otherwise, the returned value is the value of the object. In that case, we save the current identifiers and target (if applicable), then set a flag to instrument the next memory write instruction. The address written to corresponds to the location of the object. We add this address to the set of active objects and then use all of the saved information to continue the tracematch.

In Table 3.1 we summarize all of the instrumentation needed for the `before` and `after` symbols. The instrumentation points are at a function's entry and exit points and the next memory write instruction on exit (which we outlined in the previous paragraph).

Point	Before	After
Entry	Record debugging information and, if needed, resolve target address from first two arguments (for before, also advance tracematch).	
Exit	N/A	If needed, attempt to resolve the returning address. If successful, advance tracematch.
Next memory write after exit	N/A	If resolving the returning address failed, the next write address is the returning address. Record it and advance the tracematch.

Table 3.1: Summary of symbol before and after instrumentation.

A tracematch allows users to specify which classes to track and use for resolving target and returning addresses. In addition to handling classes, we allow the user to use `void*` as the type of the variable. This skips any checks that a certain class exists. We use the first argument of a function entry as the target address and return value at function exit as the returning address. Hence, we do not need to instrument any class constructors/destructors for this type of variable.

The `void*` type also allows us to apply our tool to C code. This is because C libraries typically use first argument of a function as an equivalent to "this"; this is often a pointer to some data structure.

Overall, we have 5 instrumentation functions for routines: class constructors and destructors, `before` entry, `after` entry and exit. Also, we have 1 instrumentation function for memory write instructions, controlled by a flag. These allow us to handle all types of symbols possible from the tracematch specification. Together, these functions can ensure that the target and returning addresses used by the tracematch are valid.

3.3.2 Regular Expressions

Our goal is to create a directed graph representing the regular expression NFA. We create an NFA because it can be created and navigated quickly, using the standard linear time algorithm outlined in Section 3.3.3. Along with the graph, we record the start and end vertices for the regular expression. While iterating through the tokens, we create a stack of partial regular expressions, which are the start and end vertices of the current partial

regular expressions. We can use this stack to push and pop regular expressions to handle operations.

The symbol token represents the basic transition. This creates two vertices and an edge between them, labeled with the symbol identifier. We then push this regular expression onto the stack. Figure 3.7 also presents all other operations except the constant operation. The start vertices are prefixed by s and the end vertices by e. The prime ($'$) indicates that those start and end vertices are pushed onto the stack. Vertices without a prime indicates they are arguments and popped off the stack. For operations which take two regular expressions, the first argument is 0 and the other 1.

We implemented the constant operation by making a copy of the regular expression on the top of the stack and performing the `then` operation between the copies as many times as necessary. Since all of the regular expressions in the graph are disjoint, we perform a depth first search at the starting vertex and make a copy of all vertices and edges. We also maintain a mapping of vertices, so we can determine the start and end vertices for the copy.

For the rest of the operations, it is a matter of popping from the stack, adding vertices and edges and pushing the new start and end vertices onto the stack. The unlabeled transitions in Figure 3.7 are the current transitions in the regular expression, which remain unmodified. Since we only add vertices and edges and the rest of the term is unmodified, this allows a clean implementation. The transitions we add are epsilon (ε) transitions. As a reminder, these transitions can be taken for any symbol while transitioning between states.

After iterating through all of the tokens, we ensure that our partial regular expression stack contains a single entry. This single entry represents our complete regular expression. We pop off the final start and end vertices which represent the entire regular expression. Our directed graph fully represents the regular expression NFA.

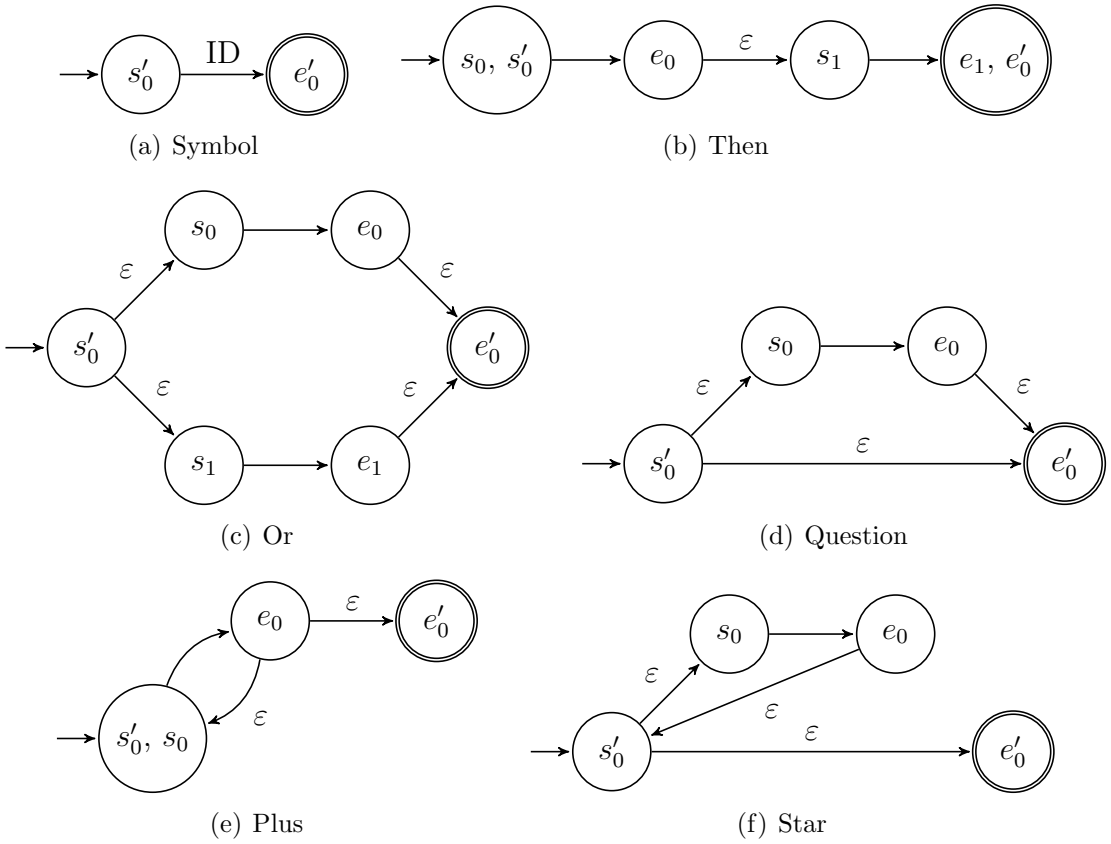

Figure 3.7: Regular expression transformations.

3.3.3 Advancing Bindings

For an individual binding we record a list of current states and a trace, which is a list of symbol names and debugging information. The symbol names and debugging information shows how a binding arrived at its current state(s). Using this, we can output a human readable trace if it matches, as we explain later in this section.

Initially, we start the current state at the starting state in the NFA, with an empty trace. Each time we advance a binding, the monitor gets the symbol identifier and its associated debugging information. Note that we also have a target and returning address (if applicable), which will be used in Section 3.3.4.

First, we must be able to advance a binding from its current state(s) to its next state(s), using the symbol identifier given. The algorithm for moving a step is given in Figure 3.8.

This algorithm is linear time in the number of states in the tracematch, based on Thompson's algorithm [25].

```
list<Vertex> current states /* current states of the history */
list<Vertex> next states /* initially empty */

while (current states is not empty) {
    pop off current state
    if (unvisited state) {
        for (every outward edge from the state) {
            if (epsilon transition) {
                add target of edge to current states
            }
            else if (transition matches symbol) {
                add target of edge to next states
            }
        }
        add state to visited state
    }
}

set current states to next states
```

Figure 3.8: Pseudocode for advancing current state(s).

However, this is not the complete implementation, since we have to take into account the modes, as well as debugging information. Recall that the modes control the output and are either `all`, `only` or `never`. We have a helper function called `current_states_is_end`, which returns whether the ending state is reachable from any of the current states. We use this to see if the trace matches in `never` mode and to make sure the trace makes it to the ending state in `all` mode.

If we advance and new current states list is not empty, we know the symbol matched and advanced the trace. We append the current symbol name and debugging information to the trace. When the mode is `never`, we output all the debugging information if the trace does indeed match. We check this by using `current_states_is_end`. Otherwise, if the new currents states list is empty, we know the symbol did not match. In this case, when the mode is `all` or `only`, we output all the debugging information.

The final check we do is for tracematches using the `all` mode. When the program ends, we check all of the current traces and check whether they can reach the ending state. For any that do not match, we again output all of the debugging information which violates the tracematch.

3.3.4 Bindings

A binding is a vector which keeps track of the address each tracematch variable is bound to. Recall that each binding has its own unique current state(s) and trace. The address at index n refers to the binding of the nth variable. Variables are allowed to be unbound, in which case the address in the binding is equal to 0. Our implementation keeps a list of all currently active bindings.

From our instrumentation we may call the monitor to advance a tracematch, passing the following: symbol identifier, variable of the target, address of the target, variable of the returning and address of the returning. Based on the presence of these parameters, we may have to keep track of multiple bindings between variables.

We maintain a list of bindings with the following information: the binding itself, the current states, the trace, and whether or not it is active. The active flag determines whether or not to silence any output from the binding. For every advance call, we must compare our current bindings to the inputs and determine which bindings we advance traces for, or if we need to create new bindings. We define 3 properties on bindings: matching, bindable and new signature.

For a binding to match, all variables which are not unbound must match exactly. Unbound variables can match any address. This allows us to advance existing bindings. For a binding to be bindable, at least one of the input variables must be unbound, while the others match or are unbound themselves. In this case, we want to bind the unbound variables and advance the history. We must also create new bindings if necessary. A new binding signature has all variables unbound except for the input variables, which are bound to their associated addresses.

With these properties in mind, we can outline how we advance the bindings. We create a list of all current bindings, which is initially empty, and populate it during execution. We present the pseudocode for advancing the bindings in Figure 3.9.

```
create new binding = true
tried to advanced = false
for (entry in bindings)
    if (entry binding is match)
        tried to advanced = true
        advance the history of the entry
        if (advance successful)
            if (entry binding is new signture)
                create new binding = false
        else
            remove entry from bindings
    else if (entry binding is bindable)
        tried to advanced = true
        copy the current entry
        set current entry to be inactive
        modify its binding
        set it to be active
        advance its history
        if (not successful)
            remove copied entry from bindings
        else
            if (copy binding is new signture)
                create new binding = false

if (create new binding)
    create new entry
    set its bindings
    set its state to the starting state
    if (tried to advanced)
        set it to be inactive
    else
        set it to be active
    advance its history
    if (successful)
        set it to be active
    else
        remove new entry
```

Figure 3.9: Pseudocode for advancing bindings.

We keep two flags while advancing a binding: whether or not we need to create a new binding and whether or not we tried to advance a binding. The first flag ensures that we don't have two bindings with the same values. For instance, assume we have a symbol 1. A tracematch's regular expression may start with one or more matches for a symbol (1+), and during execution may advance on this symbol twice (1 1). In this case we do not want two histories, one being 1 and the other 1 1. After the first symbol, we have a single history 1 with it's binding. When the monitor receives the second symbol, we attempt to advance the first history since it matches. It succeeds and we set the create new binding flag to false since its binding matches the new binding signature. After, we just want the single history 1 1, as expected.

The other variable controls whether or not the new binding is initially active or not. If the input is a match or bindable to any of the current bindings, it tries to advance a history. If we try to advance one history, it may be the case this symbol cannot advance from the starting state. Therefore, we create a new binding and silence the output by setting it invalid. If this new binding advances, we should set it to being active.

The final nuance is when we copy a binding. We only do this when the inputs are bindable to a binding. In this case we copy the binding and set it to inactive. We do not remove it since it may be used for another new binding with different arguments. Since it is inactive, it will not output anything if it fails to advance. Otherwise, if it does advance, it should be set to active since this is a valid history.

For example, consider a tracematch with the regular expression 1 2. Symbol 1 has a target and symbol 2 has a different variable as a target. Our program first executes symbol 1, which creates a new binding and advances its history successfully. The program then executes symbol 2. Since the current binding is bindable we copy the binding and set the current binding to inactive. In the copied binding, we bind this target advance the history. However, this is not a new binding signature, so we should create a new binding. Figure 3.10 shows an illustration of this example. If we did this, we would create a new binding and it would fail to advance. This would produce incorrect input in `all` or `only` modes, since it would report this binding failed. Therefore, in the case that the current symbol attempted to advance, we should silence the output. If it does advance, we should set it to active since at that point we know we were correct.

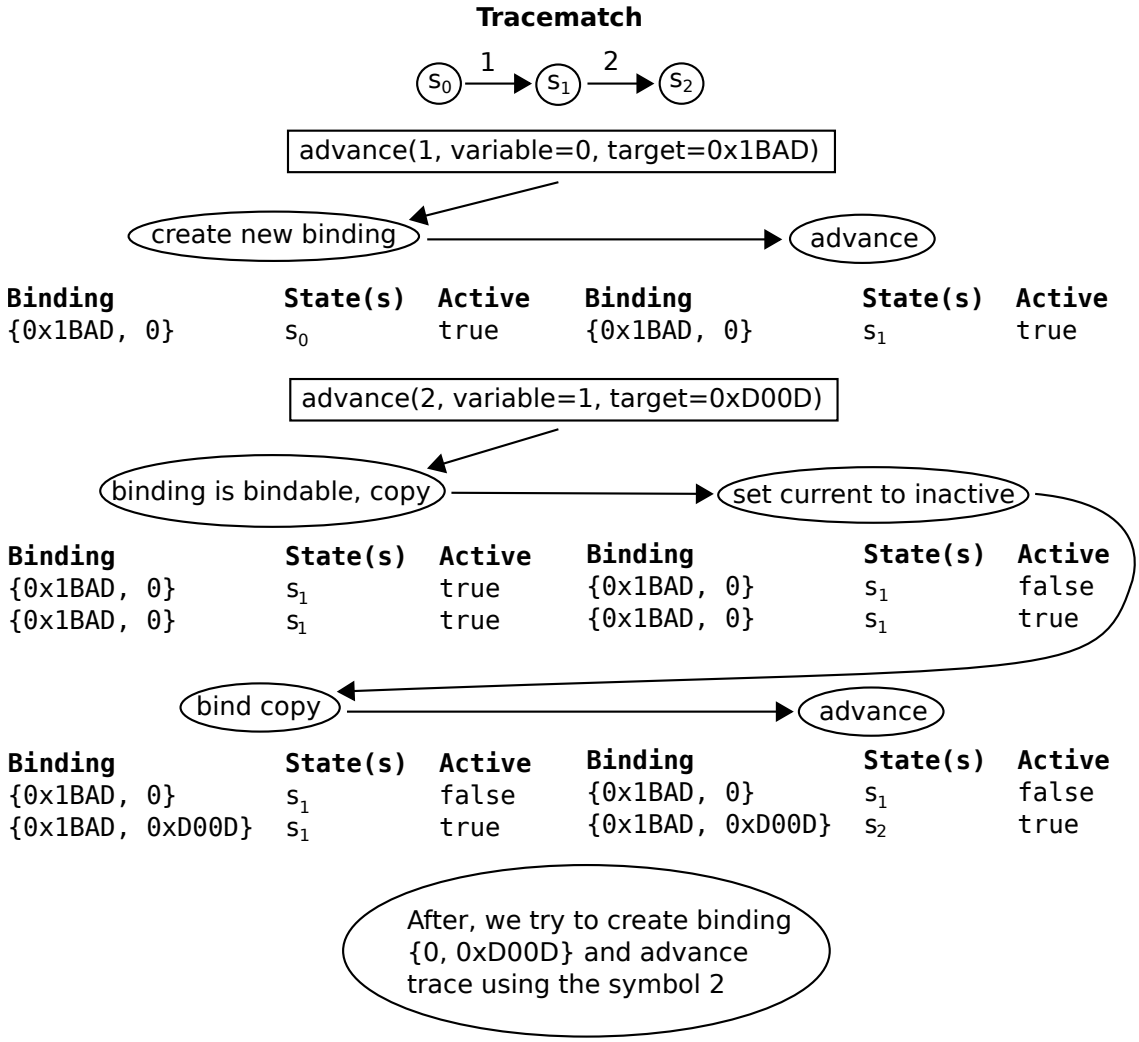

Figure 3.10: An example which shows when a new binding is inactive.

In summary, our instrumentation finds symbols and inserts code to determine target/returning addresses. When any symbols occur in the program execution, we pass the symbol identifier and address to the monitor, which maintains an NFA. Then, in the monitor we iterate through the bindings and determine which should be advanced, or created then advanced. For each of these we advance the current state(s) and trace, and output debugging information if the trace matches, depending on the mode.

3.4 Unread Memory

In this section, we discuss the implementation of our unread memory detection analysis. Our goal is to check whether memory allocated on the heap is ever read by the program, and whether any writes done to memory are eventually read. We do this only for a particular images (or binaries) specified by the user, since it is likely the developer only has control over a subset of images.

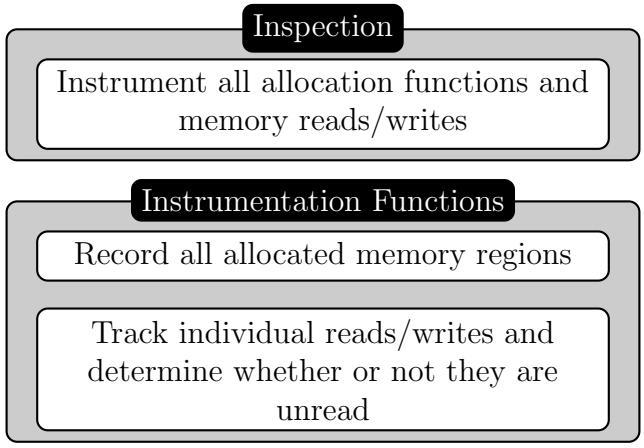

Figure 3.11: Unread Memory overview.

To utilize information from the instrumentation for use in the detection phase, we record the following data: memory address, size, class (if applicable), created debug location, destroyed debug location (if applicable), whether the memory is unread, any active writes to this memory, and any unread writes. We maintain two structures which record this information: one for currently allocated memory and another for previously unread memory.

Figure 3.11 shows an overview the implementation of this tool. The allocated memory is determined by the instrumentation of the allocation functions. We then instrument every memory read/write and determine if any memory locations are unread.

3.4.1 Instrumentation

To determine allocated memory we instrument function calls to the `malloc` functions (for C) and `new` functions (for C++). Both types of calls have the same format: the caller requests an amount of memory, and the call returns a pointer to the allocated memory. We instrument these memory-requesting functions at their entry and exit points. At the entry point, we record the size requested and store this value, to be used when the function exits. We also record the debugging information of the function call at this point. When the function exits, we read these values, along with the pointer returned, and add it to the allocated memory. Note: for calls to `new`, we ignore any malloc calls made internally, since the size and return values will be identical and we only want a single source for the allocated memory.

We instrument the free function to determine when memory is unallocated. We look for the function named `free` in the standard C library. We only instrument the function entry point and capture the first argument, which is a pointer to the memory to be unallocated. We simply record this pointer, along with the debugging information to be used in the detection phase.

To record class information we scan every symbol loaded by the binary looking for constructors and destructors for objects. We use a regular expression to determine if the function name matches the object, with an optional $\sim$ prefix. If the regular expression matches with the prefix the function is a destructor, otherwise the function is a constructor. We extract the class name and add it to the class mapping. We use the this information to report the class of an object to aid debugging.

We found that many programs had a large number of unread writes to objects with virtual functions. We investigated further by looking at the assembly code, and found it impossible to determine whether or not a memory write is to the virtual table, or a member of the object. To ignore virtual table writes, we record the debugging location of every function for an object. If we have debugging information, the line of the virtual table write corresponds to the definition of the function. Therefore, if the debugging location matches any of the debugging locations in the ignored list, this is likely a memory write to the virtual table and should be ignored. We add this debugging location to our ignore list.

Finally, we instrument memory instructions which read or write memory. However, since we determine unread memory on an image level, we only instrument instructions which are in the watched image(s).

3.4.2 Detection

For every allocation we initially record the allocated address, size and debugging information for where the memory block was created. This comes directly from the instrumentation. After the initial allocation, if it was for an object, a constructor call happens using the address as an argument. We look up the class of the first constructor call and set the allocated memory to be an object of that class. We only use the first constructor call since, if the class extends some other class, the base constructors are later called. These calls would overwrite the most specific class of the object.

For every read we look up the address in our allocated memory structure and determine if it falls in the range of [address, address + size) for any addresses. If it does, we mark the memory as read, and clear any writes to this specific location. For writes, we do the same lookup. If the current debug location is on the ignore list, we do nothing. Otherwise, we check if this location is already part of the active write. If it is, the previous write never got read, so we add it to the list of unread writes. Then, we add this write to the currently active writes. Recall that we only track allocation and read/writes from watched images specified by the user.

For a destructor or free, we look in the list of allocated memory and remove the address the function refers to from the allocated memory set. We do not need to instrument the delete function, since the destructor call happens first, and we can look it up in the allocated memory. While removing the address, we declare all active writes to be unread, since they can no longer be read. If the object is unread, or has unread writes, we move it to the unread memory list to output on the program exit.

When the program exits, we output all of the unread memory. We also output any allocated memory which is unread when the program terminates. For each entry, we output the location of creation (which will be in one of the watched images), location of destruction, whether or not it was unread and its unread write locations (which will also be in one of the watched images).

Chapter 4

Experimental Results

We describe our experience using the Tracerory tool to discover and verify properties of four open-source applications: AbiWord, a word processing application; ImageMagick, an image manipulation application; ffmpeg, an audio/video codec; and Xalan, a XSLT processor.

For tracematches and unread memory we record three runtimes: the runtime for a normal execution of the program; the runtime for an execution of the program with Pin present and no instrumentation; and the runtime of the application using our tool. Each runtime is averaged over 10 runs. At the end of the chapter, we include the standard deviation of the runs.

First, we begin by discussing our results for tracematches and unread memory in-depth using AbiWord in Section 4.1. The propose of this section is to discuss and evaluate how our tool can be used by developers.

Next, in Section 4.2, we describe our results for tracematches for each application. Section 4.3, similarly discusses our results for unread memory. For both tools re dicuss both qualivtative and quantitative results. Finally, in Section 4.4 we present a summary of the overall overhead for our tool.

4.1 Case Study: AbiWord word processor

We investigated usage patterns for AbiWord and found the following comment on their mailing list:

> However, I am pretty sure the initialisation at line 1122 needs to be to 0, because 1 results in a call to OpenTable even if there is no table in the document.

We inspected the AbiWord source code to determine the code the developer referred to and extracted a finite-state property for table usage. The complete AbiWord source code consists of 500,000 lines of code, however, we focused on the 15,000-line RTF import filter code.

We presented the property which specified constrains on table importing in Figure 2.2. To import a table, the import filter must open the table, handle the cells in the table, and then close the table. They may be multiple tables, so this pattern can repeat. We therefore specified an alphabet with symbols `open`, `close`, and `handle`. Our test program read an RTF file with two tables and converted it to a PDF file. We verified the property `(open handle+ close)+`; proper operation of the object requires it to (re-)open the table before handling a cell.

This property holds in the latest version of AbiWord, as the original bug was fixed. We were unable to obtain a version of AbiWord with the bug present. Therefore, to verify our tool, we inserted a call to `OpenTable` in the constructor to emulate the original bug as described on the list. Figure 4.1 shows the output of our tool for the modified version of the program.

```
Tracematch "TableUsage" did not hold, listing trace:
open (wp/impexp/xp/ie_imp_RTF.cpp:1487)
open (wp/impexp/xp/ie_imp_RTF.cpp:5219)
```

Figure 4.1: Modified AbiWord tracematch output.

From the output, we can easily identify the two calls to `OpenTable`. Inspecting both call locations, we observe that the call on line 1487, which is in the constructor, should not occur and needs to be removed. This demonstrates the use of our tool during development. The developer can specify constraints using our tool, and quickly verify that they hold.

Next, we ran the unread memory detector with the same test inputs. We found 177 unique locations which created memory which was either unread or contained unread writes, mostly involving strings.

Many unread writes arise from their `grow` function in the string implementation. This indicates that there are unnecessary calls which grow a string and are never used. There

are also many strings which are never read. This results in a wasted object creation, when a null pointer would suffice. The output of our tool for these objects are shown in Figure 4.2. The writes on line 187 are the object's initialization list.

```
Class: UT_StringImpl
Created: af/util/xp/ut_string_class.cpp:123
Destroyed: af/util/xp/ut_string_class.cpp:145
Unread: true
Unread Writes: 4
af/util/xp/ut_stringbuf.h:187
af/util/xp/ut_stringbuf.h:187
af/util/xp/ut_stringbuf.h:187
af/util/xp/ut_stringbuf.h:187
```

Figure 4.2: AbiWord unread memory output.

There were also many unread writes to memory involving graphics. This may indicate unnecessary allocation to structures which are only used for the GUI. Since we are only converting a document, we do not need any GUI elements. This code probably ought to be disabled for document conversions.

4.2 Tracematches

For tracematches, we focused on stating and verifying domain-specific properties of the 4 selected applications, rather than verifying generic safety properties which apply to all clients of a library, as is the case with most of the tracematches in [3]. We were particularly interested in domain-specific properties because they can serve as additional, verified program documentation through the course of the program's lifecycle.

All of the applications described in this section were compiled without debugging information included. Debugging information is only useful when a property does not hold. Our tracematch tool can still run without debugging infromation. This allows us to analyze an application without the need for recompilation. Another reason we do not use debugging information is so we can compare the results to runs that do use debugging information to see its effect.

AbiWord. As discussed in the case study, we ran AbiWord in its document-conversion mode using the table usage tracematch in Figure 2.2.

We found our input to AbiWord runs in 0.037 seconds without Pin, averaged over 10 runs. With Pin present our average runtime increases to 8.231 seconds, an overhead of 221.8x. When we enable the tracematch, we observe an average runtime of 10.043 seconds. This corresponds to an overhead of 270.7x. However, without activating our monitoring, we observe a slowdown exceeding 200x; we conclude that most of the monitoring overhead is due to Pin.

ImageMagick. ImageMagick is an application for manipulating images, which consists of over 400,000 lines of code. This application uses many external libraries to open and process images, such as `libjpeg`. We read the documentation in the header files for this library and found a function which referred the programmer to the documentation for its usage. The document states:

> You can write special markers immediately following the datastream header by calling `jpeg_write_marker()` after `jpeg_start_compress()` and before the first call to `jpeg_write_scanlines()`.

We converted the documentation to its corresponding finite state machine. Figure 4.3 presents the tracematch representing this property.

```
tracematch SpecialMarkers (void* j) {
  sym start_compress before target(j): jpeg_start_compress;
  sym write_marker before target(j): jpeg_write_marker;
  sym write_scanlines before target(j): jpeg_write_scanlines;

  start_compress write_marker* write_scanlines+
  { only }
}
```

Figure 4.3: libjpeg tracematch for markers.

As a reminder, the `only` operating mode just ensures the trace attempts to take an unknown transition. We use `only` for this tracematch since the application may not use `write_marker` or `write_scanlines` and therefore may not finish in an ending state.

We resized a jpeg image which contained markers using ImageMagick and verified that the property held. We found the program runs in 0.114 seconds without Pin. With Pin present our average runtime increases to 2.028 seconds, an overhead of 14.11x. When we enable the tracematch, we observe an average runtime of 2.315 seconds. This corresponds to an overhead of 16.11x. We found the property did hold, and the overhead introduced on top of Pin for our tool is minimal.

ffmpeg. The ffmpeg application, which encodes and decodes video/audio files, contains over 500,000 lines of code. Interally the program uses a generic framework which codecs use to perform decoding actions. Their framework contains complex usage patterns which each codec must follow. One such pattern from the documentation is the following:

> If the codec defines `update_thread_context()`, call this when they are ready for the next thread to start decoding the next frame.
>
> After calling it, do not change any variables read by the `update_thread_context()` method, or call `ff_thread_get_buffer()`.

Since this is a C library, and ffmpeg reuses memory, we at least looked at all the calls which initialized a codec frame. We made sure that one of these symbols occurred before getting the buffer, or that the thread updated its context. Next, we state that any calls to `ff_thread_get_buffer()` must happen before moving to the next buffer. The tracematch is shown in Figure 4.4.

```
tracematch InvalidBuffer (void* c) {
  sym init before target(c): ff_thread_init;
  sym init_decode before target(c): ff_h263_decode_init;
  sym update before target(c): ff_mpeg_update_thread_context;
  sym get before target(c): ff_thread_get_buffer;
  sym next before target(c): ff_thread_finish_setup;

  ((init|init_decode|update) (get* next)?)+
  { only }
}
```

Figure 4.4: ffmpeg tracematch for buffers.

We selected a video and used ffmpeg to reduce the framerate. This requires ffmpeg to decode the video and re-encode it using the new settings. We found the program runs

in 1.494 seconds without Pin. With Pin present our average runtime increases to 3.910 seconds, an overhead of 2.617x. When we enable the tracematch, we observe an average runtime of 9.973 seconds. This corresponds to an overhead of 6.676x. The property held over the program execution. The overhead for ffmpeg is the lowest we found, with less than a 7x slowdown for both cases.

Xalan. The final application we tested was Xalan. Xalan is a XSLT processor written in C++ containing approximately 250,000 lines of code. Xalan provides documentation and examples. Looking at the examples, the developers noted a common error is leaving a `DOMDocument` unreleased. Part of the documentation is the following:

> Users must call the `release()` function when finished using any objects that were created by the `DOMImplementation::createXXXX` (e.g. `DOMLSParser`, `DOMLSSerializer`, `DOMLSInput`, `DOMLSOutput`, `DOMDocument`, `DOMDocumentType`).

We expressed this property using the tracematch in Figure 4.5. This property verifies that each created `DOMDocument` (which is internally a `DOMDocumentImpl`) is released before the program terminates.

```
tracematch UnreleasedDocument (DOMDocumentImpl d) {
  sym create after target(d): DOMDocumentImpl::DOMDocumentImpl;
  sym release before target(d): DOMDocumentImpl::release;
  create release
  { all }
}
```

Figure 4.5: Xalan tracematch for releasing.

We converted a document using Xalan. During the conversation the program uses the `DOMDocument` structure.

We found the program runs in 0.008 seconds without Pin. With Pin present our average runtime increases to 3.504 seconds, an overhead of 445.5x. When we enable the tracematch, we observe an average runtime of 3.586 seconds. This corresponds to an overhead of 455.9x. The property held over the program's execution. We found the overhead for just using Pin resulted in a slowdown exceeding 440x. The additional overhead for tracematches was minimal.

4.3 Unread Memory

We test our unread memory detection by running the 4 applications and monitoring their binaries and associated libraries in the package. The unread memory detection requires debugging information in order to accurately assess unread writes for C++ programs (due to virtual functions). Also, we expect that these applications contain unread writes, and we need debugging information to interpret the results.

To test the applications, we used the same inputs as in the previous section. We do this so we can accurately compare the overhead between tracematches and unread memory detection. We also rerun our tracematches on the program containing debugging information as well in Section 4.4.

AbiWord. Previously, we discussed the output of our unread memory detection for AbiWord in Section 4.1. Now, we present the run times for the application. We found the average runtime (with debugging information) to be 0.041 seconds. With Pin present the runtime increases to 9.463 seconds, an overhead of 231.2x. With our tool the runtime is 18.9 seconds. This gives an overhead of 461.7x. In this case, the overhead of unread memory detection is approximately twice as much as with Pin alone.

ImageMagick. ImageMagick uses its own memory management, which masks the creation of memory allocations. So, for unread objects without writes, it is difficult to find the source. However, with unread writes, we can easily identify the allocations.

There are several instances of unread memory in ImageMagick. There are 131 allocations involving semaphores, even when ImageMagick is used in a single thread. The majority of the other allocations involve strings and xml-tree elments, which are unread. The largest source of unread writes came from `coders/jpeg.c:585`, with 950 unread writes. We present the code snippet in Figure 4.6.

```
583 p=GetStringInfoDatum(profile);
584 for (i=(ssize_t) GetStringInfoLength(profile)-1; i >= 0; i--)
585   *p++=(unsigned char) GetCharacter(jpeg_info);
```

Figure 4.6: ImageMagick code showing an unread write.

This shows that the jpeg info, which contains 950 bytes, is written to a pointer and

is never read. We do not have much knowledge of ImageMagick, but we believe this may indicate a bug in the program since the jpeg information string is never read.

We found the average runtime to be 0.206 seconds. With Pin present the runtime increases to 2.561 seconds, an overhead of 12.43x. With our tool the runtime is 43.854 seconds. This gives an overhead of 212.9x. In this case, the overhead of unread memory detection is an order of magntitude over just Pin. We believe this is mostly due to the large number of memory allocations.

ffmpeg. The ffmpeg application only showed 3 memory allocations which contained unread writes. The first involved the video's metadata and contained two unread writes, which may or may not be relevant. The conversion likely copies the data and does not need to read it to perform the re-encoding. The other two allocations contained thousands of unread writes each. The source of these unread writes came from the `ff_add_index_entry` which sets properties for an `index_entry`. This indicates the application is adding many index entries which are not used during execution.

We found the average runtime to be 1.502 seconds. With Pin present the runtime increases to 3.834 seconds, an overhead of 2.553x. With our tool the runtime is 155.464 seconds. This gives an overhead of 103.5x. In this case, the overhead of unread memory detection is approximately a 50x incease over Pin.

Xalan. Xalan uses its own memory management, like ImageMagick. We found no unread writes for any memory allocations in the program. However, there are over 3000 memory allocations which are never read and contain no writes. The default memory allocations themselves consume 2 bytes. This indicates the developers may want to use a representation of null, instead of a default allocation.

We found the average runtime to be 0.014 seconds. With Pin present the runtime increases to 7.535 seconds, an overhead of 537.8x. With our tool the runtime is 11.117 seconds. This gives an overhead of 793.5x, which is less than a 50% increase over just Pin.

4.4 Summary of Overhead

In this section we summarize the overhead of both tracematches and unread memory detection for the 4 open source projects. For each application, both with and without debugging information, we report the average runtime for a normal run, a run with just Pin, and a run with tracematches enabled and the associated overheads. Also, for each application with debugging information enabled, we report the average runtime and overhead for our unread memory detection. Note that these overheads do not make the program unusable.

Our results for AbiWord are shown in Table 4.1. We see that the overhead with and without debugging information is approximately the same for Pin and Tracematches. However, for unread memory detection the overhead doubles as compared to tracematches.

		Time (s)	Standard Deviation (s)	Overhead
No Debugging Information	Normal	0.037	0.005	
	Pin alone	8.231	1.093	221.8
	Tracematches	10.043	0.977	270.7
Debugging Information	Normal	0.041	0.000	
	Pin alone	9.463	1.232	231.2
	Tracematches	11.307	0.156	276.2
	Unread Memory	18.900	0.082	461.7

Table 4.1: Overhead for AbiWord.

The overhead for ImageMagick is shown in Table 4.2. The overhead for Pin and tracematches is less then 20x, with and without debugging information. The unread memory detection introduces an order of magnitude more overhead than for tracematches.

		Time (s)	Standard Deviation (s)	Overhead
No Debugging Information	Normal	0.144	0.007	
	Pin alone	2.028	0.282	14.11
	Tracematches	2.315	0.016	16.11
Debugging Information	Normal	0.206	0.020	
	Pin alone	2.561	0.195	12.43
	Tracematches	3.657	0.027	17.75
	Unread Memory	43.853	0.229	212.9

Table 4.2: Overhead for ImageMagick.

Table 4.3 presents the overhead for ffmpeg. For this application, we found a 3x increase in the overhead (both with and without debugging information) for tracematches as compared to just Pin. Similar to ImageMagick, the unread memory detection introduces an order of magnitude more overhead than for tracematches.

		Time (s)	Standard Deviation (s)	Overhead
No Debugging Information	Normal	1.494	0.176	
	Pin alone	3.910	0.611	2.617
	Tracematches	9.973	0.025	6.676
Debugging Information	Normal	1.502	0.188	
	Pin alone	3.834	0.604	2.553
	Tracematches	10.203	0.027	6.793
	Unread Memory	155.464	0.136	103.5

Table 4.3: Overhead for ffmpeg.

We present the overhead for Xalan in Table 4.4. The overhead for Pin and tracematches increases by 20-25% going from no debugging information present to having debugging information. The overhead of Pin for this application is quite substantial, and we are not sure what the cause of this is. The additional overhead for unread memory detection is approximately 40% more than tracematches.

		Time (s)	Standard Deviation (s)	Overhead
No Debugging Information	Normal	0.008	0.001	
	Pin alone	3.504	0.270	445.5
	Tracematches	3.586	0.038	455.9
Debugging Information	Normal	0.014	0.000	
	Pin alone	7.535	0.881	537.8
	Tracematches	7.971	0.055	569.0
	Unread Memory	11.117	0.071	793.5

Table 4.4: Overhead for Xalan.

In summary, we have found Pin adds slowdown which can range from 2-500x. For tracematches, the additional overhead beyond Pin is less than 50%, except for a single case were it was 300%. The unread memory detection overhead varies depending on the application. It can be as low as 40% or may introduce an order of magnitude higher overhead.

Chapter 5

Related Work

We discuss related work in the areas of dynamic binary translators and other binary analysis frameworks, as well as various approaches to verifying finite-state properties either at runtime, ahead of time (statically), or through hybrid monitoring approaches.

5.1 Dynamic Binary Translators

We have chosen to build our tracematch engine on top of the Pin engine [17]. Because we are (for the moment) not performing sophisticated program analyses but instead just monitoring program executions, any of the other dynamic binary translation engines would have worked equally well. For instance, DynamoRIO [7] and Valgrind [19] also expose the necessary information to permit the monitoring of tracematches. Valgrind's higher-overhead instrumentation enables it to detect more sophisticated program properties; it is usually used to detect memory errors, which can be thought of as typestate properties on memory blocks ("must not access memory after it has been freed").

Valgrind's memcheck tool checks the following memory errors: accesses to unallocated memory, uninitalized memory, memory leaks, double frees and overlapping memory. Our tool involving memory, however, analyzes unread writes which are not detected by Valgrind. While not as serious as memory errors (they don't cause crashes), unread writes may lead to bugs or at least wasted resources—they should qualify as a novel "code smell" [14].

Another possible approach is to dynamically rewrite the program source by inserting monitoring calls rather than building a dynamic binary translator. Such an approach could potentially be equally effective for our current class of runtime monitors; however,

it appears that some of our program transformations would be difficult to carry out in a binary rewriting system due to non-standard compilers. Also, because it is more difficult to analyze, modify, and agressively instrument the program, we believe that our current dynamic binary translation scheme will be more extensible in the future. Two rewriting frameworks are Vulcan [24] and Dyninst [8].

BitBlaze [23] applies emulation techniques to carry out dynamic analysis of binaries, primarily targetting security properties. BitBlaze illustrates a range of possible applications of dynamic analysis tools; it can protect systems against security vulnerabilities and enables the generation of exploits from security patches. A major difference between the translators mentioned above and BitBlaze is that BitBlaze emulates at the whole-system level, rather than at a per-binary level. Our tracematch analysis could also be implemented on top of BitBlaze, but, for the types of properties that we seek to verify, Pin's smaller overhead is more appropriate—we are not verifying, for instance, security properties, which are only valid if the entire system respects them.

5.2 Runtime Monitoring for Finite-state Properties

Our work implements the tracematch formalism for runtime monitoring. Other formalisms include the ones available as JavaMOP [9] plugins, as well as query languages over programs, e.g. the Program Query Language, PQL [18], or program traces in the Program Trace Query Language [15]. These formalisms enable developers to verify different classes of program properties; the differences in expressive power affect the monitoring code and any related static analyses. However, the dynamic approach in this paper should support a range of formalisms.

5.3 Static and Hybrid Approaches

Code rewriting approaches also enable static verification and analysis of the rewritten code. Compilers can, ahead of time, verify that a program never violates a stated property; but, if that is not possible, they can insert monitoring code to ensure that all executions of the program report violations of the property (or recover from the effects of the violation.) The primary difficulty in verifying finite-state properties statically is in disambiguating references to memory; analyses must use sophisticated pointer analyses to determine which potential transitions apply to which heap objects at runtime.

We list a number of examples of static and hybrid approaches. Bodden has implemented [5] a "nop-shadows analysis" which can optimize runtime monitors and statically verify that programs never violate tracematch properties by identifying instrumentation points which never contribute to violations. Dwyer and Purandare [13] also statically optimize runtime monitors; their approach groups together collections of transitions and executes these transitions all at once, to save time. Other static approaches include those of Bierhoff and Aldrich [4] and DeLine and Fähndrich's [11], who statically check typestate properties by leveraging type annotations which constrain aliasing.

All of the above approaches work on Java or .NET programs; it is easier to analyze bytecode or source code than C++ code. We chose to use dynamic binary translation in part to avoid the difficulties involved in building, parsing and analyzing C++ programs.

Xie et al. introduced the concept of using redundancies to find errors in programs [26]. They wrote a static checker in `xgcc` which looks for repeated operations. Similar to our tool, a repeated memory write to the same location is reported as an indication of a bug. However, their tool cannot detect bugs from a single write while ours can.

Seyster et al. have recently implemented an InterAspect extension to `gcc` [22], which enables developers to write aspects for their C and C++ programs. It would be possible to implement tracematches or other runtime monitors on top of InterAspect. Such an approach would support static analysis and optimization more easily, but would also require that it is possible to build the target programs with the augmented `gcc` compiler.

5.4 Leak Detection

Leak detection aims to find sources of memory inefficiency. Our tool reports any memory which is leaked and not deallocated, similar to modern tools. However, we focus on unread writes, which are beyond the scope of leak detection tools.

The most common approach is a *staleness*-based approach [6, 16]. For every allocation, staleness-based tools record timestamps with every memory access. Any memory unaccessed for a period of time can be considered as a bloat or leak. Visualization techniques using object ownership [21] allow developers to accurately pin-point more complex memory leaks.

Cherem et al. perform leak detection using static analysis [10]. They wrote a checker which records the flow of values from allocation to deallocation. They reduce the problem to a reachability problem and check that there are no memory leaks.

Novark et al. present Hound, which is a runtime system to track down memory leaks and bloat [20]. They replace the default allocators with ones that perform *data sampling*. They separate allocations based on age and take a fixed number of samples. Next, they used a staleness-based approach. Their approach is practical with less than a 1.5× slowdown. However, like other staleness approaches, they do not track individual memory reads or writes.

Chapter 6

Conclusions and Future Work

In this thesis, we have presented the concept of using runtime monitoring for finite-state properties and unread memory detection using dynamic binary translation. Our unread memory detection ensures that all memory and memory writes are read by a program.

We have implemented Tracerory, which monitors finite-state properties given in the form of tracematches and unread memory. Finite-state properties of objects constrain permissible actions on objects depending on the past history of those objects. Tracematches are particularly suitable for specifying and verifying safety and usage properties which rely on collaborations between associated objects. Our unread memory detection analyses identifies any memory blocks which are unread or contain unread writes.

Our system uses dynamic binary translation to instrument executables compiled from C++ source code. For tracematches, we have demonstrated that our system is able to monitor a suite of application-specific properties on a number of benchmark programs with acceptable run-time overhead. For unread memory detection, we have demonstrated that our system is able to correctly monitor unread memory blocks. We showed that these blocks are either wasteful, or may be a source of a bug. The overhead for this tool can be up to an order of magnitude worse than tracematches, but is acceptable.

6.1 Future Work

There are several improvements that can be made to our tool. Some libraries include usage different usage contraints based on values passed to, or returned by functions. To cover more librares, we could include bindings based on values with optional paths.

Our unread memory detection imposes a large overhead. We believe this is mainly due to the fact we must check the address of every memory operation. To reduce the runtime overhead, we could apply static analysis which indicate instructions which we can safely ignore during execution.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*, pages 345–364, San Diego, California, USA, October 2005. ACM Press.

[2] The AspectJ home page. `http://eclipse.org/aspectj/`, 2011.

[3] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitors feasible. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007)*, pages 589–608, Montreal, Quebec, Canada, October 2007. ACM Press.

[4] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007)*, pages 301–320, Montreal, Quebec, Canada, October 2007. ACM Press.

[5] Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, pages 5–14, Cape Town, South Africa, May 2010. ACM Press.

[6] Michael D. Bond and Kathryn S. McKinley. Bell: bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*, pages 61–72, San Jose, CA, USA, October 2006.

[7] Derek Bruening, Timothy Garnett, and Saman P. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 1st IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2003)*, pages 265–275, San Francisco, California, USA, March 2003.

[8] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, November 2000.

[9] Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Proceedings of the 3rd Workshop on Runtime Verification (RV 2003)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 108–127, July 2003.

[10] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 480–491, San Diego, California, USA, June 2007. ACM Press.

[11] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, volume 3086 of *Lecture Notes in Computer Science (LNCS)*, pages 465–490, Oslo, Norway, June 2004. Springer.

[12] The DWARF debugging standard. `http://dwarfstd.org/`, 2011.

[13] Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 124–133, Atlanta, Georgia, USA, May 2007. ACM Press.

[14] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[15] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*, pages 385–402, San Diego, California, USA, October 2005. ACM Press.

[16] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004)*, pages 156–164, Boston, MA, USA, October 2004.

[17] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 190–200, Chicago, IL, USA, June 2005.

[18] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*, pages 365–383, San Diego, California, USA, October 2005. ACM Press.

[19] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 89–100, San Diego, California, USA, June 2007. ACM Press.

[20] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI 2009)*, pages 397–407, Dublin, Ireland, June 2009. ACM Press.

[21] Derek Rayside and Lucy Mendel. Object ownership profiling: a technique for finding and fixing memory leaks. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 194–203, Atlanta, Georgia, USA, May 2007. ACM Press.

[22] Justin Seyster, Ketan Dixit, Xiaowan Huang, Radu Grosu, Klaus Havelund, Scott A Smolka, Scott D Stoller, and Erez Zadok. Aspect-oriented instrumentation with GCC. In *Proceedings of the 1st International Workshop on Runtime Verification (RV 2010)*, pages 405–420, November 2010.

[23] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th*

International Conference on Information Systems Security (ICISS 2008), Hyderabad, India, December 2008. Keynote invited paper.

[24] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, April 2001.

[25] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[26] Y. Xie and D. Engler. Using redundancies to find errors. *IEEE Transactions on Software Engineering*, 29(10):915–928, October 2003.