Master’s Thesis Nr. 153
Systems Group, Department of Computer Science, ETH Zurich

Program Trace Capture and Analysis for ARM

by
Andrei Pârvu

Supervised by
Prof. Timothy Roscoe
Dr. David Cock

March 2016 - September 2016
Abstract

Writing correct software for rack-scale machines has become virtually impossible without deeply understanding its behavior. In this thesis we build a proof-of-concept system that, using the DSTREAM tracing unit, non-intrusively collects program trace data from an ARM chip and uses it to verify properties. The properties are described using past time linear temporal logic formulae, which we convert into automata which are run on the obtained trace. This provides the users with a very general tool of verifying properties, which does not imply any code writing or modification of the executed program. The main advantage of our method is the decoupling of the actual automata verification from the execution of the program: it is run offline, thus decreasing the slowdown caused by adding the verification.
## Contents

1 Introduction ................................................. 5  
1.1 Properties to check ................................. 5  
1.2 Thesis layout ........................................... 6  

2 Background ................................................ 7  
2.1 DS-5 Framework & DSTREAM ......................... 7  
2.1.1 FVP simulation ..................................... 8  
2.1.2 Pandaboard OMAP 4460 .............................. 8  
2.1.3 Using scripts ....................................... 8  
2.2 PTM trace encoding format ............................ 8  
2.3 ELFIO library ........................................... 9  
2.4 Protocol buffers ....................................... 9  
2.5 LLVM .................................................... 10  
2.6 Linear Temporal Logic ................................. 11  
2.6.1 Operators ........................................... 11  
2.6.2 Past Time LTL ....................................... 12  

3 Related Work ............................................... 14  
3.1 LTLFO2MON ............................................. 14  
3.2 TESLA .................................................... 14  

4 Design ....................................................... 16  
4.1 Traced events and properties to check .............. 16  
4.1.1 malloc/free properties ......................... 16  
4.2 LTL formulae for our properties ................. 17  
4.3 Past time LTL formulae for our properties ....... 17  
4.4 Inserting data into the trace, event types & encoding ... 18  

5 Implementation ............................................. 21  
5.1 Trace decoding pipeline .............................. 21  
5.2 Trace modifying functions ............................ 25  
5.3 Symbol wrapper for malloc and free ............... 25  
5.4 LLVM instrumentation ................................. 26  
5.5 Past Time LTL parser & checker ................. 29  
5.5.1 Operators, usage and grammar ................. 29  
5.5.2 Automaton generation ....................... 31  
5.5.3 Running the automaton on a trace .......... 34  
5.5.4 Adding multiple variables .................... 37  
5.5.5 Addressing call sites .......................... 42  
5.5.6 Adding mathematical expressions ............ 42  
5.5.7 Variable quantification ....................... 45  
5.5.8 Expressions using unbound variables ........... 45
## Experiments & Results

### 6.1 Bare Metal

### 6.2 Linux

### 6.3 LTLFO2MON

### 6.4 Past Time LTL parser & checker
   - 6.4.1 Comparison for all properties
   - 6.4.2 expression checker without hash table
   - 6.4.3 Symbol wrapper
   - 6.4.4 Multiple variables

### 6.5 Trace decoding pipeline
   - 6.5.1 Symbol wrapper

### 6.6 Summary

## Conclusions & Further Work

### 7.1 Further work

## Appendices

### A DS-5 Utilization
   - A.1 FVP simulation
   - A.2 Pandaboard OMAP 4460

### B PTM trace packets
   - B.0.1 A-sync packet
   - B.0.2 I-sync packet
   - B.0.3 Atom packet
   - B.0.4 Branch packet
   - B.0.5 Waypoint
   - B.0.6 Context ID

### C DS-5 debugger launch script

### D LTLFO2MON syntax

### E Tracing Library

### F Custom Checkers
Acknowledgments

I would like to thank Prof. Timothy Roscoe for giving me the change to work in the Systems Group. I am also grateful to Dr. David Cock for his continual advice, feedback and ideas he gave me during the six months I have spent working on this thesis. Finally, I would like to thank my parents, who have supported me throughout my studies.
Chapter 1

Introduction

In the last couple of years, we have seen an increase in the usage and popularity of rack-scale machines: machines with thousands of CPU cores and terabytes of RAM, occupying a whole equipment rack. These systems are built from a large number of connected components, CPUs such as the ARMv8 X-Gene from APM, connected with a fast network, such as 40Gb Ethernet or Infiniband. While multicore CPUs are complicated enough, with their sophisticated internal interconnects, programming a rack-scale machine means treating a large distributed system as a single unit: cores in other nodes are just as much a part of the system as local cores. Writing correct software for such machines is virtually impossible, without deeply understanding its behavior. This thesis forms part of a larger research effort, to establish the feasibility of instrumenting an entire rack-scale system, to allow exploratory analysis of its behavior, and many other potential applications, such as the verification of low-level properties, and performance debugging.

The first step in this direction is to build a proof of concept: non-intrusively collect *program trace data* from an ARM chip, process it and demonstrate that we can verify some useful properties. Our basic setup consists of the DS-5 framework and DSTREAM tracing unit provided by ARM and an OMAP 4460 Pandaboard (described in section 2.1).

The first step in our work was to figure out what kind of properties we would like to check; a small introduction can be found in section 1.1 and a detailed analysis in sections 4.1, 4.2 and 4.3. Secondly, we need to build a tool which receives the program trace collected by the DSTREAM and extracts the information needed for verifying the properties mentioned earlier. This tool will effectively be a pipeline and we present its components and running mechanism in section 5.1. Lastly, the user needs a tool to specify the properties to check and verify them against the data obtained by the trace. We use a variation of linear temporal logic (section 2.6) called past time linear temporal logic (section 2.6.2) for this task and we present it in section 5.5.

The main advantage of our overall implementation is that it separates the actual decoding of the trace and the verification of the properties from the board on which the program is running, which gives a huge performance benefit. The trace can be stored and checked offline, multiple times, with different properties, if the user so desires.

1.1 Properties to check

There are a lot of interesting and important properties that one might like to check when verifying the security and correctness of a program. Probably the most basic (but, at the same time, very important) properties are related to memory management, especially in OS kernels written in C. A simple example is checking that each call to `free` with address `p` is preceded by a call to `malloc` which returns `p` and there is no other `free(p)` call between the two. If we had a trace of all the `mallocs` and `frees`, we could then check this property by keeping a set of all the addresses `p` which have been returned by `malloc` and not yet freed. Obviously, `mallocs` add addresses to the set
and frees remove them. If, at any point in time, a free tries to remove an address p which is not present in the set, then we have a violation of our property. At the same time, checking if the set is empty at the end of the trace allows us to determine if the allocated addresses are subsequently freed (there aren't any memory leaks). Our goal is to create a more general tool that allows the checking of this property (and behaves in a similar manner with the previous algorithm), but is not restricted to it, giving the users as much freedom as possible in expressing their properties.

Another useful feature is the ability to verify, at a certain point in one’s program, that a function has been previously called and returned a certain value. This is needed, for example, to make sure that initialization code has executed and hasn’t returned any errors. A description of all the properties we worked with can be found in section 4.1

Our framework allows users to either build their own custom checkers to implement properties such as the ones described above or write the properties using past time linear temporal logic (section 2.6.2) which will be subsequently checked by a generic checker (section 5.5).

### 1.2 Thesis layout

In order to build our proof-of-concept system we have taken the following steps:

- connect the DSTREAM debug unit to the OMAP 4460 Pandaboard and understand the format of the trace; this trace type, known as Program Trace Macrocell (PTM), is presented in sections 2.1 and 2.2.
- build a pipeline which takes the trace output from the DSTREAM, correlates it with the run program and filters out the information needed; we address this problem in section 5.1. The different stages of the pipeline communicate between themselves using protocol buffers, for which a brief summary is given in section 2.4
- figure out what data we need to include in the trace and how to we can do that (sections 4.4, 5.2, 5.3)
- allow the user to specify, in the run program, which function call and what data to trace; for this we use LLVM (briefly described in section 2.5) to modify the program and insert the necessary code that includes the desired information in the trace; we detail this in section 5.4
- build a way allowing users to specify properties to check; we do this by interpreting formulae written in past time linear temporal logic; a background presentation of linear temporal logic is done in section 2.6; we process these formulae and build an automaton which is used to validate a trace (section 5.5)
- implement the actual checker, which runs the previous automaton on a trace and decides if it satisfies the specified property (section 5.5)

Besides this, as related work, we briefly present two tools:

- TESLA, which allows users to specify temporal properties in programs, which are evaluated at runtime (section 3.2)
- LFLFO2MON, which allows users to specify linear temporal logic formulae, and verifies then against a given trace (section 3.1)

We will compare our system with these two tools and determine its advantages and disadvantages.
Chapter 2

Background

2.1 DS-5 Framework & DSTREAM

DS-5 Development Studio is a “suite of tools for embedded C/C++ software development on any ARM-based SoC, featuring editor, compilers, debugger and system profiler” [13]. It can be used to write, build, execute and debug programs for ARM SoCs. Together with the DSTREAM, which is a debug and trace unit [14], it can be used for connecting to an ARM board, doing step-by-step debug and obtaining the trace of the executed program.

The programs run on the chosen platform are compiled using the armcc compiler and linked with the armlink linker; the result of this process is an ELF file with the .axf extension which can be found in the “Debug” folder of the current DS-5 project.

The main feature of the DS-5 framework that we are interested in is the trace, so we will mainly focus on it. The DS-5 framework provides a list of commands with which the user can interact; this list is quite long and can be found here¹. The commands that we used are the following:

- **break** - allows setting a breakpoint
- **trace clear** - clears the buffer in which the trace is stored
- **run** - starts the debugged program and can pass it run-time arguments
- **trace start** - starts the trace
- **trace stop** - stops the trace
- **trace dump** - dumps the context of the trace buffer into a file; this is a compressed format (for Pandaboard, PTM, section 2.2)
- **trace report** - dumps the trace, but in human-readable format, i.e. printed assembly instructions
- **wait** - waits until a breakpoint is reached

As you can notice, there are two separate commands for dumping the trace: one for the compressed format and one for the human-readable format. Throughout this thesis we will prefer using the compressed one, because, although it needs to be decoded, it gives an advantage when dealing with a large trace sizes: we can determine what information we need to keep and what we can discard, so the program that does the checking does not need to read every executed assembly instruction.

DS-5 supports a number of platforms for debugging, out of which we will summarize two. The actual instructions to configure these platforms are omitted here, as they don’t have any importance to our overall project and are described in appendix A.

### 2.1.1 FVP simulation

This allows a user to run a program without having a physical board. However, tracing on an FVP is intrusive, as it affects the running time of the program. Also, the FVP simulation only supports the dumping of the trace in human-readable format; therefore, any tool which wants to analyze it must actually parse the file containing the printed assembly instructions.

### 2.1.2 Pandaboard OMAP 4460

Debugging the Pandaboard with DS-5 allows the user to dump the contents of the ETB buffer which contains the raw trace in the PTM format (section 2.2). Unfortunately, the ETB buffer is only 8KB on the Pandaboard, so the number of instructions which can be traced is limited. The `armlinker` linker must be given a base address for the ready-only code, which in this case is 0x80000000.

### 2.1.3 Using scripts

Of course, manually configuring options from the graphical interface of DS-5 is not viable when developing an automated tool. Luckily, the DS-5 framework also provides command line and configuration files which allow this. The `debugger` executable can be found in the installation folder of DS-5 (in our case, `/usr/local/DS-5_v5.24.0/bin/debugger`) and it can receive the following arguments:

- an entry which selects the board and method of debug
- an entry which selects the connection (DSTREAM in our case)
- an entry which receives a file with the Debug and Trace Service Layer options; this allows the user to specify ranges of addresses to be traced and other features (more in appendix A)
- the image which is to be loaded on the board
- a script which contains the debugger commands which are to be executed

The full script used for launching the debugger can be found in the Appendix, section C.

### 2.2 PTM trace encoding format

The trace output by the Pandaboard using the DSTREAM can be obtained in a binary format which can then be decoded in order to obtain all the assembly instructions executed. The trace consists of several packet types which are described in the PTM manual [1] and which we summarize. To minimize the size of the trace, it does not output every executed instruction, but only generates packets when a waypoint instruction is encountered: an instruction which changes the normal flow of the program.

Waypoint instructions are divided in two categories: direct branches and indirect branches. Direct branches are those instructions for which the target address can be determined from the instruction code and indirect branches are the ones for which the address can’t be determined from the code. Therefore, packets describing the results of direct branches are simpler, because they must only specify if the branch was taken or not. The similar packets for indirect branches are more complicated, because they have to include the destination address of the branch instruction.

Hence, in order to decode the trace we have to identify these branch packets and correlate them with the assembly instructions obtained from the ELF file.
The actual packet types and their format, although not important from a conceptual point of view, need to be understood in order to efficiently parse the binary trace and obtain the jump addresses. A more detailed description of these can be found in appendix B.

A small example is shown in Figure 2.1, containing assembly code and the corresponding PTM packets. The first branch instruction generates a PTM packet for direct branches: we only need to know if the branch was taken or not, because we can determine the destination address. Afterwards, no packet is generated for the arithmetic add instructions: the decoder must figure this out and find the next branch instruction. Finally, a blx instruction is encountered, where the destination address is taken from register r7, therefore PTM generates an indirect branch packet.

```
cmp r4, #2
beq 80000184 --------→ PTM direct branch packet
add r5, r5, #1
add r6, r6, #2 --------→ nothing generated
add r0, r5, r6
blx r7 --------→ PTM indirect branch packet
```

Figure 2.1: Correlation between PTM packets and assembly code

### 2.3 ELFIO library

As described in section 2.2, our trace decoding algorithm needs to inspect the ELF file and retrieve the instructions at certain addresses, e.g. waypoint instructions. For this purpose we use the ELFIO library [11] which provides an API for obtaining the needed information:

- elfio and section classes for iterating through the ELF’s sections
- text_section_accessor class and get_string method for instruction encoding
- symbol_section_accessor class get_symbol method for names of functions and their beginning address

### 2.4 Protocol buffers

Protocol buffers [10] are a platform-independent method of serializing structured data. The user defines one or more messages, each having a variable number of fields: either primitive data (int, string, etc.) or other messages. We chose this tool because, besides its ability of compacting data, it provides a clear way of specifying messages and their properties, which are enforced by the generated code. Message fields can be defined to be either required or optional and can repeat themselves; also, message fields can be stored under a union type (oneof), which suits our need of having multiple packets from the PTM trace. An example of a protocol buffer message can be seen in listing 2.1: it represents a message which contains the PTM packets decoded from the trace; the actual packets can be seen in appendix B and we will discuss their role in our tracing decoding pipeline in section 5.1; we show it here just as an example for the reader. The numbers on the right hand side of the fields are used for serialization and for possibly identifying older versions of messages.

The protoc compiler generates source code for reading, accessing and writing the messages a user defined. Messages can be serialized and parsed either from an input file (using the functions ParseFromIstream and SerializeToOstream) or to a buffer (ParseFromArray and SerializeToArray).
2.5 LLVM

LLVM is a set of tools and libraries which facilitate building compiler back-ends and front-ends. It provides an easy-to-use method of analyzing both high-level and low-level code of a given program and also supports adding, removing or modifying code.

A LLVM user can do this by implementing, in C++, either function passes (by inheriting FunctionPass) or module passes (by inheriting ModulePass). These are call-backs which get called each time the LLVM framework analyzes a function or a module. The user can iterate through the function’s basic blocks, inspecting each instruction one by one and determining their type and properties (if it’s an assignment to a variable or a call site, etc.). This inspection of a function’s instructions is usually done using iterators: each function provides a way to iterate through its basic blocks and each basic block provides an iterator for its instructions.

Call sites can be inspected using the CallInst class with which the user can get the name of the called function and the arguments used in the call site.

New code can be inserted using the IRBuilder class, which must receive an iterator which points to the position where the new code is to be inserted. The class has specific methods to create each instruction type (e.g. CreateCall, CreateIntCast). For replacing instructions the function ReplaceInstWithInst can be used, which has a similar usage with IRBuilder.

Module passes allow the calling of function passes and also support adding or removing functions, by using the getOrInsertFunction method of the Module class.

LLVM's ability to manipulate and insert code has proven to be very useful and is used both by our system and by TESLA, a similar property analyzer which we will present in section 3.2.
2.6 Linear Temporal Logic

Linear temporal logic (LTL) extends predicate logic (in which formulae contain predicates, e.g. $p(x)$) by adding special temporal operators that describe events in time. It is built on top of $\mathcal{L}$, a first order language: it contains variables which appear inside predicates and formulae which describe properties of these predicates; variables can be quantified, but the predicates cannot.

A formula in language $\mathcal{L}$ is usually called an assertion; given an assertion $p$ and a state $s$ which instantiates all the variables from $p$ we write “$s \models p$” to denote that $s$ satisfies $p$.

Temporal logic is interpreted over a model which is an infinite sequence of states, $\sigma = s_1, s_2, s_3, \ldots$. Given a temporal formula $p$ and a model $\sigma$, we say that $p$ holds at time $i, i \geq 0$, if $p$ satisfies the state of the model at $i$, $(\sigma, i) \models p \iff s_i \models p$.

Temporal logic is a very good tool for expressing properties to check in our programs, because it provides a formal and easy-to-understand way of allowing a user to interact with our tracing system.

We continue by listing and explaining the main temporal logic operators we use for describing our properties. After understanding these basic operators the reader can continue with section 4.2 and 4.3 for actual properties written using LTL. The notations and the basic concepts for these operators have been taken from [2].

2.6.1 Operators

Globally

$$(\sigma, j) \models \Box p \iff \forall k \geq j, (\sigma, k) \models p.$$ $p$ holds globally if it holds at the current state and in all states that will come in the future.

Eventually

$$(\sigma, j) \models \Diamond p \iff \exists k \geq j, (\sigma, k) \models p.$$ $p$ eventually holds if it holds at some state in the future.

Next

$$(\sigma, j) \models \Diamond p \iff (\sigma, j + 1) \models p.$$ $\Diamond p$ holds at the current state if $p$ holds at the next state.

Until

$$(\sigma, j) \models p U q \iff \exists k \geq j, (\sigma, k) \models q \text{ and } \forall i, j \leq i \leq k, (\sigma, i) \models p.$$ $p$ until $q$ holds at a state $j$ if $q$ holds at some state $k$ in the future and $p$ holds at every state between $j$ and $k$. 
Weakly Until

\( (\sigma,j) \models p W q \iff (\sigma,j) \models p U q \) or \((\sigma,j) \models \Box p \). \( p \) weakly until \( q \) holds if \( \Box p \) holds or \( p \) until \( q \) holds. The difference between weakly until and until is that the former becomes true even if we don’t reach a state in which \( q \) becomes true.

```
| 1 | \( p \) | \( p \) | \( p \) | \( \cdots \) | \( q \) | \( \cdots \) |
```

or

```
| 1 | \( p \) | \( p \) | \( \cdots \) | \( \cdots \) | \( p \) |
```

2.6.2 Past Time LTL

Building a tool to actually verify an LTL formula on a given trace is impossible in finite time. The LTL formula is transformed into a Büchi automaton [3] which must accept a trace infinitely often; of course, this is not possible when dealing with finite traces, like in our case.

Given the fact that the traces we obtain from our system are always finite we will use past time LTL (ptLTL) formulae for specifying our properties, because they have the advantage of being possible to check.

The ptLTL operators are very similar to the ones we saw in section 2.6.1 and we briefly list them here. A more thorough analysis of them and a description of the checking algorithm can be found in section 5.5.

Globally in the Past

\( (\sigma,j) \models \Box p \iff \forall k, 1 \leq k \leq j, (\sigma,k) \models p \). \( p \) holds globally in the past if \( p \) holds at any moment in time since 0 until the current moment.

```
| 1 | \( p \) | \( \cdots \) | \( p \) | \( j \) |
```

Eventually in the Past

\( (\sigma,j) \models \Diamond p \iff \exists k, 1 \leq k \leq j, (\sigma,k) \models p \). \( p \) holds eventually in the past if it holds at some point in the past.

```
| 1 | \( \cdots \) | \( p \) | \( k \) |
```

Previously

\( (\sigma,j) \models \Diamond p \iff (\sigma,j-1) \models p \). \( \Diamond p \) holds at the current state if \( p \) holds at the previous state.

```
| 1 | \( \cdots \) | \( p \) | \( j-1 \) |
```

Since

\( (\sigma,j) \models p S q \iff \exists k, 1 \leq k \leq j, (\sigma,k) \models q \) and \( \forall i, k \leq i \leq j, (\sigma,i) \models p \). \( p \) since \( q \) holds at state \( j \) if \( q \) holds at some state \( k \) in the past and \( p \) holds at every state between \( k \) and \( j \).
Weakly Since

\( (\sigma, j) \models p \, S_w \, q \iff (\sigma, j) \models p \, S \, q \) or \( (\sigma, j) \models \Box p \). \( p \) holds weakly since \( q \) if \( \Box p \) holds or \( p \) since \( q \) holds. As in the LTL case, weakly since holds even if \( q \) does not become true, as long as \( p \) is true at every point in time until the present state.
Chapter 3

Related Work

3.1 LTLFO2MON

LTLFO2MON\(^1\) is a tool developed by Andreas Bauer, Jan-Christoph Küster and Gil Vegliach which implements an automata-based method for verifying if a trace accepts an LTLFO formula \([6]\). It is based on LTL\(_3\) Tools \([8]\), \([9]\), which converts a LTL formula into a finite state machine. This tool has a specific syntax which we will not describe here, in order to not confuse the reader; it can be found in appendix D. We will continue to use the standard LTL syntax presented in section 2.6. The output of the monitor can be either \textbf{True} (\(\top\)), \textbf{False} (\(\bot\)) or \textbf{inconclusive} (\(?\)). Let’s consider a few examples. For example, the formula \(\Box \forall \text{pair} (x,y) \implies x \leq y\). If we verify the previous formula against the trace \(\{\text{pair}(2,3)\}, \{\text{pair}(4,5)\}, \{\text{pair}(10,5)\}\) then we will obtain \(\bot\), as 10 is not less or equal to 5. If, instead, we omit the last element of the trace (\(\{\text{pair}(10,5)\}\)) then the result becomes \(?\); this is because the trace is considered to be infinite, and the monitor cannot predict anything about future appearances of the \text{pair} element. If we change the previous formula to \(\exists \text{pair} (x,y) \implies x \leq y\) then the evaluation on the previous traces will yield \(\top\), because we eliminated the Globally statement and replaced “for all” with “exists”.

3.2 TESLA

Temporally Enhanced System Logic Assertions (TESLA) \([4]\) is a tool developed at the University of Cambridge Computer Laboratory by Jonathan Anderson et. al. which allows users to specify temporal properties that they expect their programs to have. These properties are inspired by LTL. TESLA allows users to insert assertions into their program which can test:

- if a function has been called / will be called
- if a function returns a certain value and is called with certain parameters
- field assignments to a given structure

The user can use the keywords \texttt{previously} and \texttt{eventually} to specify events relative to the moment of the assertion. Internally, TESLA constructs an automaton which transitions from one state to another based on the elements specified in the assertion (such as the ones we saw above). TESLA allows the user to specify temporal bounds, i.e. the moments when the automaton of an assertion is created / initialized and when it is finalized (and evaluated).

We present a few simple examples of TESLA assertions so the reader can get familiar with it. Below is a very simple assertion.

\(^{1}\text{github.com/jckuester/ltlfo2mon}\)
Listing 3.1: simple TESLA assertion

```
TESLA_WITHIN(example_syscall,
              previously(security_check(ANY(ptr)) == 0));
```

It checks that function `security_check` has previously been called, with any pointer value as a parameter, and has returned 0. The temporal bound is the function `example_syscall`, so only events that take place inside the function flow are taken into consideration. Below is a more complicated assertion, which specifies events previous to the moment of assertion (the call to `foo` with parameter 42 and return value 0) and future events (the call to `foo` with parameter 43 and return value 0).

Listing 3.2: another TESLA assertion

```
TESLA_WITHIN(bar,
              TSEQUENCE(
              foo(42) == 0,
              TESLA_NOW,
              foo(43) == 0
              )
              );
```

TESLA uses LLVM (section 2.5) hooks inserted in the intermediate code in order to detect events specified in assertions and evaluate their truth value. Unfortunately, this can prove to increase the execution time of the analyzed programs too much, a trait we want to improve using our tracing system. TESLA is very similar to our system with regard to the type of properties we check. A comparison of runtime between TESLA and our system can be found in section 6.2.
Chapter 4

Design

4.1 Traced events and properties to check

As discussed briefly in section 1.1, we monitor the following properties in our trace system:

- Checking that a function has been called and has returned a certain value — in this way we can compare our system with TESLA and also have a basic, easy to understand starting point.

- Monitoring calls to `free` and `malloc` — many properties can be checked when knowing this information, which are listed in the subsection 4.1.1. Memory checks are crucial for any serious application and are also easy to understand by most programmers. Moreover, they are a good exercise for LTL and past time LTL.

- Monitoring calls to a specific function (the parameters with which it is called and its return value, if any); this can be done either globally, or per call site; the users can then create their own property to check. This allows a user to address areas of the program and limit the bounds of the properties.

Our tool is not restricted to only these properties, as it is totally general. We are only using them as examples and for testing.

4.1.1 malloc/free properties

We check the following properties for malloc/free:

- **no double allocation**: two calls to malloc return the same value \( p \) only if between the two a call to `free(p)` exists and there is no other malloc returning \( p \).

- **no double frees**: a call `free(p)` is preceded by a call to malloc which returns \( p \), and there is no other `free(p)` call between the two.

- **no leaks**: all values returned by malloc are eventually freed

- be able to check the previous properties inside a function, not necessarily globally.

The first property verifies that the implementation of malloc is correct and does not return the same address without it being freed in the meantime. The other two properties verify that a program using malloc and free is implemented correctly: frees are called only for valid addresses and there are no memory leaks.

Another interesting aspect is that the three properties present different problems for a tool which uses LTL, as we will see in section 4.2.
4.2 LTL formulae for our properties

We tried to express the no double allocation, no double frees and no leaks properties using LTLFO2MON so that we could benefit from the monitor framework described in section 3.1. These properties can be categorized in safety properties, which verify that we never reach a bad state, and liveness properties, which verify that we eventually reach a good state. We presume that the trace is in the format of LTLFO2MON, e.g. malloc(address) and free(address).

- no double allocation: □ ∀ malloc(x) ⇒ ○(¬malloc(x) W free(x)); this formula specifies that each time we encounter a malloc with address x, starting from the next point in time we shouldn’t encounter another malloc with x, weakly until we encounter a free of x. As a reminder, weakly until signifies that the second property (free(x)) does not necessarily need to be encountered as long as the first one (¬malloc(x)) remains true. Because we verify that we never reach a state in which we have two allocated addresses, this property is a safety property.

- no double frees: (□ ∀ malloc(x) ⇒ add_in_set(x)) ∧ (□ ∀ free(y) ⇒ check_in_set(y)); this is the property we briefly talked about in section 1.1. As we have seen, we need to maintain a set of addresses already returned by malloc but not yet freed. However, LTLFO2MON does not have the ability to let the user store additional information that could be used in the future, so we had to implement the add_in_set and check_in_set functions ourselves. add_in_set just adds the value of x to the set of allocated addresses and always returns true whereas check_in_set checks if x is present in the set and removes it, or returns false otherwise. This is also a safety property, because we check that we never reach a state in which we have a double free. Of course, these additional functions are not part of the LTL syntax, and show the limitation of LTLFO2MON when dealing with our properties.

- We cannot check the no_leak property, because it is a liveness property: the output of the trace will always be undecided, so we cannot benefit from using it. This shows the limitation of dealing with a tool that operates with infinite trace.

To conclude, a tool that deals with infinite traces does not apply to the properties we want to check; using past time LTL would provide a better solution, as it can emulate a finite trace: time 0 is the entry point of our program and we only deal with events that happened between the current moment in time and the entry point.

4.3 Past time LTL formulae for our properties

As described in the section 4.2, past time LTL seems to be more suited to our needs, because it operates on a finite trace. Using the tool build by us in section 5.5, we can express no double allocation, no double frees and no leaks properties in the following way:

- no double allocation: □ x = malloc(\_) ⇒ ◯(¬x = malloc(\_) S w free(x)); this formula is the reverse of the one seen in section 4.2; the only interesting difference is the replacement of weakly until with its past time equivalent, weakly since.

\[
\begin{array}{ccccccc}
\text{free(x)} & \text{\_} & \text{\_} & \text{\_} & \text{\_} & \text{\_} & \text{\_} \\
1 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

or

\[
\begin{array}{ccccccc}
\text{\_} & \text{\_} & \text{\_} & \text{\_} & \text{\_} & \text{\_} & \text{\_} \\
1 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\]

- no double frees: □ free(x) ⇒ ◯(¬free(x) S x = malloc(\_)); this shows the improvement past time LTL brings to our formula for no double frees shown in section 4.2; it
does not require any side effect functions and it is similar with the formula for **no double allocation** with the exception of **weakly since** being replaced with **since**. This is due to the fact that if we encounter a **free** without a previous **malloc** (for the same address) we want to fail.

\[
\begin{array}{cccc}
\top & x=\text{malloc} & \neg\text{free}(x) & \neg\text{free}(x) \\
0 & \cdots & \cdots & \text{free}(x)
\end{array}
\]

- **no leaks**: \(\neg x = \text{malloc}(\_\_\_) \ S_w \ \text{free}(x)\): we are not allowed to encounter a **malloc**, which returns a given \(x\), since the last encounter a **free** of \(x\). Note that this property is lacking globally in the past (\(\square\)) because we are not interested in having no leaks at **any point** in the program, but only at the end.

\[
\begin{array}{cccccc}
\top & x=\text{malloc} & \neg x=\text{malloc} & \neg x=\text{malloc} & \neg x=\text{malloc} & \text{end of trace} \\
1 & \cdots & \cdots & \cdots & \cdots & \text{end of trace}
\end{array}
\]

or

\[
\begin{array}{cccccc}
\top & \neg x=\text{malloc} & \neg x=\text{malloc} & \neg x=\text{malloc} & \neg x=\text{malloc} & \text{end of trace} \\
1 & \cdots & \cdots & \cdots & \cdots & \text{end of trace}
\end{array}
\]

- **no leaks** inside a function: \(\square \text{ret}_{\text{foo}} \implies (\neg x = \text{malloc}(\_\_\_) \ \lor \ \neg \Diamond \text{call}_{\text{foo}}) \ S_w \ \text{free}(x)\): here we want to check **no leaks** only inside function **foo**. therefore we modify the previous formula by excluding any **mallocs** that happen before the call to **foo**. An actual example of a trace being verified with this formula can be seen in section 5.5.3.

\[
\begin{array}{cccccc}
\top & \text{call}_{\text{foo}} & \text{free}(x) & \neg x=\text{malloc} & \neg x=\text{malloc} & \text{ret}_{\text{foo}} \\
1 & \cdots & \cdots & \cdots & \cdots & \text{end of trace}
\end{array}
\]

### 4.4 Inserting data into the trace, event types & encoding

The PTM trace protocol traces only executed instructions - it does not support the possibility of adding data to the trace (such as return value of functions or function parameters).

This is unfortunate, because most (if not all) of the properties we want to check (such as those presented in section 4.1) need such data; the **no double frees** property discussed in section 1.1 clearly needs to know the addresses return by **malloc**, respectively the ones passed to **free**.

Because of this, we sought to improvise a method of adding arbitrary data to our trace: the PTM protocol can be configured to include the contents of the **CONTEXTIDR** register in the trace. The size of the register is 32 bits, but only the least significant 8 bits are used by the processor as an address space ID. Thus, each time we need to include a value in the trace, we can modify the remaining 24 bits.

For reading and modifying the value of **CONTEXTIDR** register the **mrc** and **mcr** instructions are used. Unfortunately these instructions are privileged, so, in order to use them from Linux, specific system calls have to be written.

Because of the limited space provided by the **CONTEXTID** register, we decided that each event we are tracing will be composed out of two context ID packets, for a total of 48 bits of available space.

These 48 bits are split in three: 8 bits which describe the type of the **event** (which is a number chosen by us), 8 bits of **information** about the ID of the function or the index of the parameter we are tracing and 32 bits storing the actual **data** generated by the event (the layout is shown in Figure 4.1).

We next describe the events that we include in our trace and the way the 48 bits of data are used.
CALL
A function call. The event type is \texttt{0xe0} and the information is a function ID which uniquely identifies the function being called. The 32 bits of data aren’t used.

ARG
The value of a parameter used in a function call. The event type \texttt{0xc0}, the information is the index of the parameter in the function’s list of parameters and the data is the parameter’s value.

RET
A return from a function call. The event type if \texttt{0x20}, the information is a function ID which uniquely identifies the returning function and the data is the value returned by the function.

MALLOC
A call to \texttt{malloc}. The event type if \texttt{0x40}, the information isn’t used (default value of \texttt{0xff}) and the data is the address allocated by \texttt{malloc}.

FREE
A call to \texttt{free}. The event type if \texttt{0x60}, the information isn’t used (default value \texttt{0xff}) and the data is the value returned by \texttt{free}.
Although both \texttt{MALLOC} and \texttt{FREE} events can be incorporated in the CALL and RET events, we wanted to have separate ones because of the predominance of properties addressing these two.

CHECK
An assertion for previous function call. As described in section 4.1 we wanted to allow the users the possibility of checking if a function has been called and if it returned a certain value. This event indicates such a check: the event type is \texttt{0x80}, the information is the function ID which uniquely identifies the checked function and the data is the value the function is expected to return.

RETURN_VAL
The complement event of CHECK, it represents the returned value of a checked function. The event type is \texttt{0xa0}, the information is the function ID which uniquely identifies the checked function and the data is the value the function returned.

Each value traced from a function (be it a parameter or a return value) is converted to 32 bits in order to fit into the data section (therefore we don’t support tracing of structs or 64-bit integers). These 32 bit values are always split in half and copied into the upper 16 bits of the two context id packets.
The function IDs found in the information section are generated by the LLVM pass which handles the actual insertion of these events in the trace; more details can be found in section 5.4. One special case is the tracing of parameters: the argument packets always follow a function call packet,
hence there is no need to record the ID of the argument’s function. Instead, the information bits are used to store the index of the parameter which is logged (thus only emitting packets for arguments we are interested in).
Chapter 5

Implementation

5.1 Trace decoding pipeline

Processing the trace is divided into 4 different stages: PTM decoding, assembly parser, trace generator and property checker, as seen in Figure 5.1. The output of each stage is stored as a series of protocol buffer messages (section 2.4) and passed as input to the next stage. We chose protocol buffers because they allow us to specify an exact format of the messages and they can be encoded as binary files, which occupy less space than text files.

We adopted the model of a pipeline, because each stage does different computations which can start before the previous stage finishes all its work (these results can be seen in section 6.5).

![Figure 5.1: Decoding Pipeline]

PTM decoding

The binary produced by the DSTREAM is read, decoded and used to create a protocol buffer message which contains the packets described in section 2.2 and appendix B. Once again, the understanding of these packets is important for the actual implementation and less for the high-level view of our system. The output is composed out of the following messages (their exact definitions being shown in listing 5.1 and 5.2):

- **ASync** - this is just a synchronization packet, and has no information
- **ISync** - stores the **address** for which the packet was generated and the **reason** the packet was generated. The **cycle_count** and **context_id** might appear, as described in section 2.2. This type of packet is quite important because, when specifying tracing ranges, the ISync is used to inform of the address of the region we are currently in
- **Atom** - stores the atom types, as described in section 2.2; might contain from 1 up to 5 atoms.
- **Branch, Waypoint** - they store the compressed address format. As described in section 2.2, the addresses might be compressed, showing only the bits that modified since the last Branch or ISync packet. In order to represent this, we have a 32 bitmask which symbolizes which bits of the **address** field are used: presuming the last address is **last_addr**, the new address is formed by keeping the bits of **last_addr** indicated by **mask** and OR-ing them with **address**.
- **a 32-bit context_id packet** (which can be seen in the Instruction message); it is important because we use it to store additional information, as described in section 4.4.
• Instruction - a wrapper for any of the above packets; the type of the packet is stored, which allows the user to then access the corresponding field

• Flow - a sequence of Instruction messages

Listing 5.1: Decoding PTM packets

```protobuf
message ASync {
}

message ISync {
  enum Reason {
    PERIODIC = 0;
    TRACE_ON = 1;
    OVERFLOW = 2;
    DEBUG = 3;
  }

  required Reason reason = 1;
  required uint32 address = 2;
  optional uint32 cycle_count = 3;
  optional uint32 context_id = 4;
}

message Atom {
  enum Type {
    E = 1;
    N = 2;
  }

  repeated Type atoms = 1;
  optional uint32 cycle_count = 2;
}

message Branch {
  required uint32 address = 1;
  optional int32 exception = 2;
  required uint32 mask = 3;
  optional uint32 cycle_count = 4;
}

message Waypoint {
  required uint32 address = 1;
  required uint32 mask = 2;
}

message Instruction {
  enum Type {
    ASYNC = 1;
    ISYNC = 2;
    ATOM = 3;
    BRANCH = 4;
    WAYPOINT = 5;
  }
```

Listing 5.2: Decoding instruction packets
ASM parser

The PTM packets stored in the Instruction messages from the previous phase are correlated with the ELF executable from which the trace has come. This allows us to obtain the actual instructions that are executed and store them for later use.

As described in section 2.2, the trace contains packets only for waypoint instructions, where the program flow changes. This implies that, for every new Atom, Branch or Waypoint message this stage of the pipeline must find the next waypoint instruction, determine its destination address (either from its encoding or from the Branch/Waypoint packet) and mark all the instructions in between as executed. In order to do this we use the elfio library shown in section 2.3, which allows parsing the ELF file, retrieving the encoding of the instructions and verifying if they are waypoint instructions.

Because storing all the executed instructions would be a waste of space, our current format only outputs sequences of basic blocks: groups of instructions that contain only one entry and one exit point and don’t have any jump instructions except, maybe, the last one.

The BasicBlock message contains the start and end addresses of the basic block, an optional function name and if the basic block is at the beginning of a function.

Besides the basic blocks, we also keep have a DebugInfo message which is created from the the values logged by the Context ID packets. As we have described in section 4.4 we log various events in our trace using the CONTEXTID register, and store the event type, the data generated by the event and other information.

Listing 5.3: Basic block packets

```plaintext
message DebugInfo {
  required uint32 data = 1;
  optional uint32 event = 2;
  optional uint32 info = 3;
}

message BasicBlock {
  required uint32 start_address = 1;
  optional uint32 end_address = 2;
  optional string name = 3;
}
```
trace generator

Identifies the events stored in the DebugInfo messages and creates specific messages for each of them, to be passed to the checker. This is useful, because it creates a protobuf file in which the events can be easily read by users and correlated with the program.

The output of this stage can be seen in listing 5.4: the event messages from here are the ones described in more detail in section 4.4. The names of the functions cannot be deduced directly from the trace, but instead are determined from the function IDs: the LLVM pass generates a manifest which contains the correlations between these two; this is done in order to allow the user to specify ptLTL formulae which contain function names, not IDs, which would have been cumbersome.

Listing 5.4: Trace packets

```protobuf
message Param {
  required bool is_defined = 1;
  optional uint32 param_val = 2;
}

message Call {
  required string name = 1;
  repeated Param params = 2;
}

message Ret {
  required string name = 1;
  required uint32 val = 2;
}

message Malloc {
  required uint32 addr = 1;
}

message Free {
  required uint32 addr = 1;
}

message Event {
  enum Type {
    CALL = 1;
    RET = 2;
    MALLOC = 3;
    FREE = 4;
  }
  required Type type = 1;
}
```
oneof event {
    Call call = 2;
    Ret ret = 3;
    Malloc malloc = 4;
    Free free = 5;
}

message Trace {
    repeated Event events = 1;
}

property checker
Takes as input a sequence of events and a past time LTL formula (section 2.6.2) and checks the truth value of the formula on the events. Instead of using a past time LTL formula a custom checker can be written to verify a certain property.
The actual past time LTL formula is transformed into an automaton, which enables easier checking: more details about the syntax of the formulae and the generated automata can be found in section 5.5.

5.2 Trace modifying functions
As described in section 4.4 we need to explicitly insert code on the target machine which modifies the CONTEXTID register in order to trace data.
The functions used for doing this can be found in appendix E. These are:

- `trace_value(p, event, info)`: main trace function; generates two context ID packets, as described in section 4.4, `p` being the 32-bit value, `event` being the 8-bit event and `info` being the 8-bit information, either a function id or an argument index.
- `send_return_value` for RETURN_VAL
- `emit_value` for CHECK
- `emit_call` for CALL
- `emit_ret` for RET
- `send_arg` for ARG

The trace ranges which can be configured from DTSL (section 2.1) prove to be valuable: we don’t need to trace the whole program, only the calls to `trace_value`: this reduces the size and the complexity of the trace output file.

5.3 Symbol wrapper for `malloc` and `free`
In order to have a lightweight, simple method for generating events for `malloc` and `free` (which, as shown in section 4.1 constitute a big part of the properties we want to check) we built a symbol wrapper which, given an address range of an ARM ELF executable, replaces all the calls to `malloc` and `free` with calls to the wrappers shown in listing 5.5.
Listing 5.5: malloc and free wrappers

```c
void *my_malloc(size_t size) {
    void *ret = malloc(size);
    trace_value((unsigned int)ret, 0x40 /* MALLOC */, 0xff);
    return ret;
}

void free(void *p) {
    free(p);
    trace_value((unsigned int)p, 0x60 /* FREE */, 0xff);
}
```

The wrappers use the `trace_value` function shown in section 5.2 and the MALLOC and FREE event ids from section 4.4 to add the allocated and freed addresses to the trace.

We built this symbol wrapper as a simple and quick method of generating events. Of course, it is limited only to malloc and free symbols, so we decided to build a more complex system, using LLVM, shown in section 5.4.

### 5.4 LLVM instrumentation

To be able to easily specify what events should be traced we added to our tracing library (appendix E) a number of helper functions described below. A LLVM pass goes through the module, identifies calls to these helper functions and takes the appropriate actions, by generating events using the functions from section 5.2.

As described in section 5.1, the pass generates unique IDs for the functions it has to work with. The association between function names and IDs is written in the `manifest.trace` file, which is later used by the trace generator phase of the trace decoding pipeline.

**check_ret**

`check_ret(function_name, return_value)` checks whether `function_name` has been previously called and returned `return_value`. The pass goes through all the functions in the current module, finds all call sites and then generates a RETURN_VAL packet for each. Afterwards it replaces the call to `check_ret` with a CHECK packet. An example can be seen in listing 5.6: one can observe that `foo` has been given ID 1, every call to it is now succeeded by a call to `send_return_value` and `check_ret` has been replaced with `emit_value` which also references the function based on its ID.
Listing 5.6: check_ret before pass

```c
int foo(...) {
    ...
    return bar;
}

void func() {
    ...
    foo();
}

void main() {
    x1 = foo(...);
    ...
    x2 = foo(...);
    check_ret("foo", 0);
}
```

Listing 5.7: check_ret after pass

```c
int foo(...) {
    ...
    return bar;
}

void func() {
    ...
    ret = foo();
    send_return_value(ret, 1);
}

void main() {
    x1 = foo(...);
    send_return_value(x1, 1);
    ...
    x2 = foo(...);
    send_return_value(x2, 1);
    emit_value(0, 1);
}
```

**instrument_call_site**

`instrument_call_site(int, ...)` is used when the user wants to trace a certain call site of a function. `instrument_call_site(int, ...)` must be called directly before the call site and it has to contain the same number of parameters as the traced function has, ended by −1. Each parameter must be either 0 or 1, signifying whether the corresponding parameter from the call site will be traced.

The pass first identifies the instrumented functions and generates a wrapper for each of them. The wrapper, besides calling the original function, issues a CALL packet, multiple ARG packets, depending on the value of the parameters of `instrument_call_site` and a RET packet. An example can be seen in listing 5.8; one can observe that only the first call to `foo` was instrumented.

**instrument**

`instrument(function_name, ...)` is similar to `instrument_call_site`, but will trace all calls to `function_name`. Listing 5.10 shows that this time both calls to `foo` have been instrumented.
Listing 5.8: instrument_call_site before pass

```c
int foo(int a, int b, int c) {
    ...
    return bar;
}

void main() {
    instrument_call_site(0, 1, 1, -1);
    foo(2, 4, 6);
    ...
    foo(4, 5, 6);
}
```

Listing 5.9: instrument_call_site after pass

```c
int foo(int a, int b, int c) {
    ...
    return bar;
}

int foo_wrap_(int a, int b, int c) {
    emit_call(1); // 1 is the id generated // for 'foo'
    send_arg(b, 2);
    send_arg(c, 3);
    int ret = foo(a, b, c);
    emit_ret(ret, 1);
    return ret;
}

void main() {
    foo_wrap_(2, 4, 6);
    ...
    foo(4, 5, 6);
}
```
Listing 5.10: instrument before pass

```c
int foo(int a, int b, int c) {
    ...
    return bar;
}
void main() {
    instrument("foo", 0, 1, 1, -1);
    ...
    foo(2, 4, 6);
    ...
    foo(4, 5, 6);
}
```

Listing 5.11: instrument after pass

```c
int foo(int a, int b, int c) {
    ...
    return bar;
}
int foo_wrap_(int a, int b, int c) {
    emit_call(1);
    // 1 is the id generated
    // for 'foo'
    send_arg(b, 2);
    send_arg(c, 3);
    int ret = foo(a, b, c);
    emit_ret(ret, 1);
    return ret;
}
void main() {
    foo_wrap_(2, 4, 6);
    ...
    foo_wrap_(4, 5, 6);
}
```

5.5 Past Time LTL parser & checker

In order to be able to specify properties seamlessly, without writing a specific checker, we developed a parser which accepts as input a ptLTL formula and builds an automaton from it. The automaton can then be used by a generic checker to validate a given trace. The parser extracts the events from the input formula, and then uses the recursive formulas described in section 2.6.2 to build an automaton. The states of the automaton encode different values of the set of subformulas and the edges are labeled with the different events that might take place.

5.5.1 Operators, usage and grammar

Our tool provides the following past time LTL operators (next to them are the equivalent operators presented in section 2.6.2):

- $G_p$ - globally in the past ($\square$)
- $E_p$ - eventually in the past ($\diamond$)
- $P$ - previously ($\rho$)
- $S$ - since ($S$)
- $Sw$ - weakly since ($S_w$)
We also have the basic logic operators: \( \Rightarrow \) (implication), \( \land \) (conjunction), \( \lor \) (disjunction) and \( ! \) (negation).

In order to address the events described in section 4.4 we provide the following semantics:

- **call\_$func_name** - the event of \$func_name being called; note that \'$\$' is part of the syntax
- **ret\_$func_name** - the event of \$func_name returning
- **malloc, free** - call to malloc or free in the trace; these events are the ones generated by the symbol wrapper form section 5.3

### Adding one variable

In order to be able to refer to the values returned by functions or to parameters of functions, we added the possibility of having one variable which can be bound to one of these values. The variable is universally quantified at top level (so the property written must hold for all instances of it) and the first occurrence of it binds it to the respective value. The next occurrences of the variable become assertions which must hold in order for the formula to remain true. The following semantics are used to address the variable (which must be named \( x \)):

- **x\=$func_name**, \( x \) is bound to the return value of \$func_name; note that \'$\$' is part of the syntax
- **\$func_name(x)**, \( x \) is bound to the value of the parameter of \$func_name. If \$func_name has more than one parameter, the ones the user is not interested in are marked with \( _ \) (e.g. \$foo(_, _, x, _))

Let’s consider a simple example,

\[
Gp \ (x = \$foo \rightarrow (Ep \ $bar(x)) \land (Ep \ x = \$my\_func));
\]

for which the standard ptLTL syntax is

\[
\Box \ (x = \$foo \rightarrow (\Diamond \ $bar(x)) \land (\Diamond \ x = \$my\_func));
\]

This formula tests whether each time we encounter a call to \$foo which returns \( x \) we must have encountered previously a call to \$bar with parameter \( x \) and a call to \$my\_func which returns \( x \).

Let’s see how the variable \( x \) is bound during an example of an accepting trace, like the one in Table 5.1. The first time we encounter \$bar(x) the variable \( x \) is bound to whatever value appears in the function, in our case 1. Next, further occurrences of \$bar or \$func with the same value of \( x \), i.e 1 will correlate with the previous events involving \( x \). This can be observed when we encounter \$bar(2): \( x \) is bound to a new value, 2, which is completely independent of previous bindings. Finally, after a call \( x = \$foo \) a check is performed to see if the current binding of \( x \) has previously encountered calls to \$bar and \$func. In our example, for both bindings (1 and 2), it does, so the trace satisfies the property. However, if the trace would have ended with the event \( 3 = \$foo \), then that would have created a new binding for \( x \), which would have failed the property.
In section 5.5.4 we will discuss about adding more than one variable to our formulae, but, for the moment, what we have is enough to write the properties we have shown in section 4.1.

The overall grammar that describes a past time LTL formula is:

```
root : expression ;

expression:
    Gp expression |
    expression -> expression |
    Ep expression |
    P expression |
    expression S expression |
    expression Sw expression |
    expression □\ expression |
    expression V expression |
    ! expression |
    "x" = $name |
    $name(func_param) |
    (expression) |
    call |
    ret

func_param:
    param |
    param, func_param

param:
    "x" | "_"

call:
    "call_" $name

ret:
    "ret_" $name
```

The precedence of the operators is:

1. =
2. !
3. /\, V
4. Sw, S
5. Ep, P
6. ->
7. Gp

5.5.2 Automaton generation

We evaluate the operators described in the previous section using the recursive formulas presented in [5]. We'll use the same notation as in section 2.6: \((\sigma, j)\), meaning the state of our trace at time \(j\). These recursive formulas allow us to implement a dynamic programming algorithm which
evaluates the truth values of the formulae at the current state by using the values from the previous state. Hence,

- \((\sigma,j) \models \Box F\) iff \((\sigma,j) \models F\) and \((j > 1 \implies (\sigma,j-1) \models F)\); globally in the past at time \(j\) is true if \(F\) holds at time \(j\) and globally in the past is true at time \(j-1\)

- \((\sigma,j) \models \Diamond F\) iff \((\sigma,j) \models F\) or \((j > 1 \land (\sigma,j-1) \models F)\); eventually in the past at time \(j\) is true if \(F\) holds at time \(j\) or eventually in the past is true at time \(j-1\)

- \((\sigma,j) \models \# F\) iff \((j > 1 \land (\sigma,j-1) \models F)\); previously is true at time \(j\) if \(F\) holds at \(j-1\)

- \((\sigma,j) \models F_1 S F_2\) iff \([(\sigma,j) \models F_2 \lor (j > 1 \land (\sigma,j) \models F_1) \land (\sigma,j-1) \models F_1 S F_2]\); \(F_1\) weakly since \(F_2\) is true at time \(j\) if \(F_2\) holds at time \(j\) or \(F_1\) holds at time \(j\) and \(j\) is greater than 1 implies that \(F_1\) weakly since \(F_2\) is true at time \(j-1\). The difference between this formula and the previous one is that a trace formed entirely out of \(F_1\) satisfies the weakly since property: this is due to the implication, which is true if \(j\) equals 1

In order to apply the previous algorithm we recursively split the input formula into a tree of smaller formulae, where the leaves are function calls or returns and inner nodes temporal or...
logical operators. We continue using the example of the check for no double frees which has the following formula:

\[ Gp \, \$\text{free}(x) \rightarrow P \, !\$\text{free}(x) \, S \, x = \$\text{malloc}; \]

for which the standard ptLTL syntax is:

\[ \square \, \$\text{free}(x) \implies \neg \$\text{free}(x) \, S \, x = \$\text{malloc} \]

The tree derived from this formula can be seen in Figure 5.2. The leaves of the formula tree consist the events that we might encounter in our trace, so they will be used to compute the transition function of the automaton. The nodes are numbered using a post-order traversal of the tree.

The states of the generated automaton represent the truth values of the formulae from the computed formula tree. Therefore, each state will be encoded as a truth vector, in which the \(i\)-th element represents the truth value of the subformula from node number \(i\) of the formula tree; this value can be true (1), false (0), or undecided (2). Undecided is used for certain formulae in the initial state, in which there is no current event and the formula cannot be evaluated either true or false.

The initial state will be the one corresponding to an empty trace, in which we have not encountered any events so far. Let’s determine the truth vector of the initial state from the formula tree shown in Figure 5.2.

1. \(\$\text{free}(x)\); is 0, as there is no current event
2. \(\$\text{free}(x)\); is 0, as there is no current event
3. \(!\$\text{free}(x)\); is 1, because \(2.\text{free}(x)\) is 0
4. \(x = \$\text{malloc}\); is 0, as there is no current event
5. \(!\$\text{free}(x) \, S \, x = \$\text{malloc}\); is 0, because \(x = \$\text{malloc}\) is 0 and the trace is empty
6. \(P \, !\$\text{free}(x) \, S \, x = \$\text{malloc}\); it’s 2 (undecided) because nothing has happened in the past
7. \(\$\text{free}(x) \rightarrow P \, !\$\text{free}(x) \, Sw \, x = \$\text{malloc}\); is 1, because \(\$\text{free}(x)\) is 0
8. \(Gp \, \$\text{free}(x) \rightarrow P \, !\$\text{free}(x) \, Sw \, x = \$\text{malloc}\); is 1, because on an empty trace we must consider every globally in the past formula as being true

Hence, the initial state will be encoded as 00100211 and is written in italic in Figure 5.3.

The other states will be computed with a Breadth-First-Search algorithm: we maintain a queue of states for which we haven’t yet computed the transitions. At each step, we remove a state from the queue and we proceed to add one of the events from the leaves of our formula tree as an outgoing edge. We compute the truth vector of the new state. If the new state hasn’t been previously encountered, we add it to the queue, otherwise we do nothing. Let’s see how we compute the new truth vector when we add the event \texttt{malloc} to the initial state:

1. \(\$\text{free}(x)\); is 0, as \texttt{malloc} is the current event
2. \(\$\text{free}(x)\); is 0, as \texttt{malloc} is the current event
3. \(!\$\text{free}(x)\); is 1, because \(2.\text{free}(x)\) is 0
4. \(x = \$\text{malloc}\); is 1, as \texttt{malloc} is the current event
5. \(!\$\text{free}(x) \, S \, x = \$\text{malloc} \); is 1, because \(x = \$\text{malloc}\) is 1
6. \(P \, !\$\text{free}(x) \, S \, x = \$\text{malloc}\); is 0 because in the initial state \(!\$\text{free}(x) \, Sw \, x = \$\text{malloc}\); was 0
7. $free(x) \rightarrow P \neg free(x) Sw x = \$malloc;$ is 1, because $free(x)$ is 0

8. $Gp \neg free(x) \rightarrow P \neg free(x) Sw x = \$malloc;$ is 1, because 7 is 1 and because formula 8 from the previous state was 1

Hence, the new state will be encoded as 00111011; this corresponds with the generated automaton, which can be seen in Figure 5.3. The accepting states of the automaton are the ones which have 1 at the end of the label, signifying that the root of the formula tree is true. Also, the accepting states are enclosed in double circles, for easier observation.

Because the output automaton can become exponential, even for a relatively simple formula (as the one seen above), an finite state machine open source library [12] is used to minimize it so that it can be efficiently used by the checker. The minimized version of the previous automaton can be seen in Figure 5.4. If thinking about our no double frees property the automaton makes sense: if consecutive frees are encountered, then we should reject; otherwise, we keep moving between two accepting states. Also, we must encounter at least one malloc before a free, because we don’t want to free addresses which haven’t been allocated. Note that no transition for a certain event in a state signifies that the automaton rejects if that event is encountered in that state.

### 5.5.3 Running the automaton on a trace

The final stage of checking a formula is to take the trace output by the trace generator stage of our pipeline and run it on the automaton.

The main problem of this step is how to handle bound variables because, up until now, our automaton hasn’t any knowledge of them. The solution we decided to implement is to keep a separate instance of our automaton for each binding, because they are independent. If thinking of our previous example, with the no double frees property, we are only interested in frees and mallocs which refer to the same address.

Every time we encounter an event in a trace which contains a variable $x$, we look for the automaton instance associated with $x$ and move on the edge labeled with the event. If no instance corresponding to $x$ exists, then we create a new automaton, starting from state 0.

To better demonstrate the algorithm, we shall run the automaton from Figure 5.4, corresponding with the no double frees property, on the following trace:

$$\text{malloc}(10), \text{malloc}(16), \text{free}(10), \text{malloc}(10), \text{free}(16), \text{malloc}(20)$$

The events correlated with the automaton instances can be seen in Table 5.2. For each instance we store the address of malloc or free as a key and the state of the automaton as a value.

<table>
<thead>
<tr>
<th>event</th>
<th>automaton instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>{}</td>
</tr>
<tr>
<td>malloc(10)</td>
<td>{10: 1}</td>
</tr>
<tr>
<td>malloc(16)</td>
<td>{10: 1, 16: 1}</td>
</tr>
<tr>
<td>free(10)</td>
<td>{10: 0, 16: 1}</td>
</tr>
<tr>
<td>malloc(10)</td>
<td>{10: 1, 16: 1}</td>
</tr>
<tr>
<td>free(16)</td>
<td>{10: 1, 16: 0}</td>
</tr>
<tr>
<td>malloc(20)</td>
<td>{10: 1, 16: 0, 20: 1}</td>
</tr>
</tbody>
</table>

Table 5.2: Automaton instances

As you can see, until now, the no double frees property is not violated; every instance of our automaton is in an accepting state. However, if we add to the trace either:

- free(16): we have no free transition for the instance (16: 1), so the automaton will reject. This makes sense, because we have double freed the address 16

- free(100): we have no free transition starting from state 0 of the automaton, so, again, we will reject, because the address 100 has not been allocated.

34
Figure 5.3: Generated automaton for no double frees
This set of instances is very similar to the set of addresses we discussed in section 1.1 when checking the **no double frees** property. This demonstrates that our automaton generation and execution is just a generalization of that custom checker, a generalization that permits a user to verify properties without writing an explicit checker. Implementation-wise we need to keep a hash which links the values of our variable to the state in which the corresponding automaton is.

**Events without variables**

Let’s consider a more complicated example, the **no leaks** property within a function (described in section 4.3); this can be expressed as

$$Gp \text{ ret}_\$foo \rightarrow (x = \$\text{malloc} V (Ep \text{ call}_\$foo)) \ Sw \$\text{free}(x);$$

for which the standard ptLTL syntax is:

$$\Box \text{ ret}_\$foo \implies (\neg \$\text{malloc}(x) \lor (\Diamond \text{ call}_\$foo) \ Sw \$\text{free}(x);$$

We’ll presume that we are talking about a single call site of foo, so, for clarity, we need to add the following formulas to be one above:

- $Gp \text{ ret}_\$foo \rightarrow P (Gp \text{ ret}_\$foo) - we encounter the return of the function only once
- $Gp \text{ call}_\$foo \rightarrow P (Gp \text{ call}_\$foo) - we encounter the call to the function only once
- $Gp \text{ ret}_\$foo \rightarrow Ep \text{ call}_\$foo - call precedes return

The resulting automaton, from AND-ing the previous 4 formulae can be seen in Figure 5.5. The ret_$foo and call_$foo events don’t have any associated variables, so when we encounter them we need to update every instance of our automaton. Let’s take the examples of the programs in listing 5.12 and listing 5.13: in the first one, 10 addresses are allocated, but only 9 freed inside function foo; in the second one all the addresses are freed inside foo. Until we encounter call_$foo we remain in state 0, because we don’t care about any operation happening outside the function. Afterwards, we switch between states 1 and 2 for each pairing of malloc and free with the same address: state 2 signifies that an allocated address has not (yet) been freed and state 1 signifies the opposite. When the end of the function is reached, denoted by ret_$foo, we transition to state 3 where, again, mallocs and frees have no effect. All instances of the automaton must be in state 3 (because state 2 does not have an outgoing edge labeled with ret_foo), otherwise the no leaks property doesn’t hold.

Iterating through all the instances when encountering an event without variables decreases the performance of our trace system, because we now no longer have to only compute a hash of the variable’s value. However, if these events are infrequent then their impact shouldn’t be very big.
### 5.5.4 Adding multiple variables

The next step in building our verification system was allowing users to specify multiple variables in our ptLTL formulae. Although, in the beginning, this seems as an easy generalization (just concatenate the values of each variable to identify the instance of the automaton), we soon observe the following problem: what happens if a variable has not yet been bound? We must maintain a wildcard for it and match it later when the variable becomes bound.

What we have now is an instance of the first-order unification problem in which we have to match previous bindings of variables (which might contain wildcards) with the current binding (which also might contain wildcards, not necessarily for the same variables).

The key to understanding the unification algorithm is being able to correlate variable bindings with the set of automaton instances which they represent. For example, the binding \((10, 37)\)
signifies all the automaton instances in which the second variable is bound to 10 and the first and third variables have any/all value. Theoretically, we should keep an automaton instance for every possible instantiation of variables and then, when encountering an event, update all the instances that correspond to that event. Of course, this is practically impossible, because each variable has an infinite set of possible values; therefore, we use a more compact representation of it.

Throughout the algorithm we maintain a unification tree which has the following properties:

- each node has an associated binding.
- in the root all variables are unbound (\(\_\) for all variables).
- each node of the tree might have 0 or more children.
- each child represents a subset of automaton instances included in the set of instances which the parent represents.
- the order of the children matters: a child represents the set of all instances which its binding covers, but which are not included in the children to its left.

Before describing our algorithm we must define some properties between bindings. These properties are the similar with set operations because we are, in fact, dealing with operations on sets of automaton instances.

- equality: a binding \(b_1\) equals another binding \(b_2\) if all unbound variables in \(b_1\) are also unbound in \(b_2\) (and vice-versa) and any variable bound in \(b_1\) to a certain value is bound in \(b_2\) to the same value.
- matching: a binding \(b_1\) matches binding \(b_2\) if for each bound variable in \(b_1\) it is unbound in \(b_2\) or it is bound in \(b_2\) to the same value as in \(b_1\). Examples: \((\_ 10 20)\) matches \((\_ 10 \_); (10 20 \_)\) matches \((\_ 10 \_); (10 \_ 20)\) does not match \((\_ 10 30)\).
- inclusion \((b_2 \subseteq b_1)\): a binding \(b_2\) is included in \(b_1\) if \(b_1\) matches \(b_2\) and any unbound variable in \(b_2\) is also unbound in \(b_1\). Examples: \((\_ 10 20)\) is included in \((\_ \_ 20)\), but isn’t included in \((10 \_ 10)\)
- intersection \((b_1 \cap b_2)\): if two bindings \(b_1\) and \(b_2\) match then we can compute their intersection: for each variable, if it has a value in at least one of the two bindings then that value goes into the intersection. Example: \((10 \_ 30)\) intersected with \((\_ 1 30)\) is \((10 1 30)\). Hence, the matching property can be rewritten as \(b_1 \cap b_2 \neq \emptyset\).

The algorithm for inserting a node with a binding is recursive and we describe it for binding \(b\) and a node \(n\), whose binding we will write as \(n:b\). The main idea is that if we find a child in which our binding is included, then we can recurse in it. If the binding is not included, but matches, then the set of instances \(b\) represents is effectively "split" in two: the instances that are included in the child and the rest. We must therefore both recurse in the child and continue our search in the rest of the children. If, after checking all children, we still have left-over instances from \(b\), we must add a new child which contains them.

The pseudocode for our function is listed below:

```plaintext
if b equals n:b then STOP
else
    added = add_new_child = False
    foreach child c of n do
        if b matches c_b then
            new_b = b intersect c_b
```

38
Let’s presume we have the following formula

\[ G_p \, my\_func(x,y) \implies G_p \, (\bar{z},x) \implies E_p \, foo(y,z) \];

It checks that when we encounter a call to \( my\_func(x,y) \) preceded by a call to \( \bar{z},x \) then we must have already encountered a call to \( foo(y,z) \). This formula is not particularly useful, but it serves the purpose of illustrating our algorithm.

The automaton generated for it can be seen in Figure 5.6. It can be observed that if we encounter \( \bar{z},x \) before \( foo(y,z) \) then there isn’t any edge for \( my\_func(x,y) \) because the automaton will fail.

Let’s consider the trace from table 6.2. Obviously, the trace doesn’t satisfy the formula shown above, because the call \( \bar{2,20} \) isn’t preceded by a call \( foo(20,2) \).

Let’s see how our unification tree looks like throughout the processing of the trace. Initially, it
contains only the root (Figure 5.7). All the instances represented by the root are in state 0. The circle to the left shows the order of the variables and the italic number on the right of each node is the current state of the node’s automaton instances.

When encountering $foo(20,1)$ we have to insert binding $\langle 20 \ 1 \rangle$ in the tree: because the root has no children, we add one; this is due to the fact that some automaton instances must step and others must not. All the automaton instances represented by this new node will be in state 2, because we took a $foo$ transition from state 0 (Figure 5.8). Next, we encounter $bar(2,10)$ with binding $\langle 10 \ 2 \rangle$. This binding does not match $\langle 20 \ 1 \rangle$, so we have to add a new node. The automaton instances of this node will be in state 1, because we took a $bar$ transition from state 0 (Figure 5.9).

The algorithm continues with $bar(1,10)$ and binding $\langle 10 \ 1 \rangle$. This binding matches $\langle 20 \ 1 \rangle$, but is not included in it, so we compute the intersection and recurse into the first child. This new node will remain in state 2. Because $\langle 10 \ 1 \rangle$ was not included in $\langle 20 \ 1 \rangle$ and does not match $\langle 10 \ 2 \rangle$ we have to add a new child to the root. This child will represent the instances for all the bindings of the form $\langle 10 \ y \ 1 \rangle$ in which the second binding ('y') is not equal to 20 (otherwise, it would be a subset of the first child) (Figure 5.10).

Finally, we encounter $my\_func(10,20)$ with binding $\langle 10 \ 20 \ 1 \rangle$. It will match the node with $\langle 10 \ 20 \ 1 \rangle$, which will remain in state 2, but it will also match $\langle 10 \ 2 \ 1 \rangle$. This implies that the new node will have to be in the state that results from transitioning along the edge $my\_func$ from state 1, but no such edge exists, therefore the checker outputs that the trace fails the property. Notice that although $\langle 10 \ 20 \ 1 \rangle$ matches $\langle 10 \ 1 \rangle$ their intersection $\langle 10 \ 20 \ 1 \rangle$ is included in the first child, so it’s not added to the tree (Figure 5.11).

### Complexity

Of course, this algorithm is more computationally intensive than the hash-based one we used for one variable, and its complexity, in the worst case, can be $O(N^3)$, where $N$ is the total number of events. This is due to the fact that at each event, we might need to traverse all the nodes of the unification tree and check all the previous children for intersection.

However, if after reading the formula we have only one variable, we can adapt our algorithm to use the same hash-based method we described in section 5.5.3.
Figure 5.9: Unification tree after 2 events

Figure 5.10: Unification tree after 3 events

Figure 5.11: Unification tree after 4 events
5.5.5 Addressing call sites

Let’s analyze again the formula for no leaks inside a function presented in section 5.5.3. As explained earlier, the equivalent automaton (Figure 5.5) works for only one call site: after encountering ret_foo the automaton transitions to state 3, where further mallocs and frees don’t have any effect.

Now having the possibility of using multiple variables in formulae, we sought to improve the previous formula to handle multiple call sites. To achieve this, we make use of the call_id and ret_id fields computed during the trace generator phase in our pipeline (section 5.1). We added the possibility of addressing these IDs in our formulae by specifying them at the end of call or return events.

Hence, our previous formula from section 5.5.3 now becomes:

\[ Gp \text{ ret } \$\text{foo}\{y\} \rightarrow (|x = \$\text{malloc} V !(Ep \text{ call } \$\text{foo}\{y\})) Sw \$\text{free}(x) \]

We are referring to the pairs of ret_foo and call_foo events which have the same ID (denoted by the variable \( y \)), so we are analyzing the calls to malloc and free which specifically reside within the same call-site of foo. Also, as we’ve done before, we need to specify that we are dealing with call sites, so we add in the formula:

- \( Gp \text{ ret } \$\text{foo}\{y\} \rightarrow P (Gp !\text{ret } \$\text{foo}\{y\}) \) - we encounter a given return instance only once
- \( Gp \text{ call } \$\text{foo}\{y\} \rightarrow P (Gp !\text{call } \$\text{foo}\{y\}) \) - we encounter a given call instance only once
- \( Gp \text{ ret } \$\text{foo}\{y\} \rightarrow Ep \text{ call } \$\text{foo}\{y\} \) - a given call instance precedes its return instance

The resulting automaton can be seen in Figure 5.12.

5.5.6 Adding mathematical expressions

So far, we can only refer to a variable by using it as a parameter of a function call or as the return value of a function call, but we are not able to write expressions involving these variables (equalities or inequalities). This would help us verify, for example, that certain return values of functions are strictly positive (which in the case of malloc implies that it actually returns a valid address).

Therefore, we improved our grammar by adding operators for equals, less than, less than or equal,
greater than, greater than or equal and operators for ‘AND’ and ‘OR’. We decided that the symbols for these operators should be different from the ‘AND’ and ‘OR’ symbols used to connect temporal expressions, in order to provide an easier grammar parsing and more clarity to the users.

Our new grammar can be seen in listing 5.5.6.

| root: temp_expression ; |
| telmp_expression: |
|     Gp predicate | |
|     predicate -> predicate | |
|     Ep predicate | |
|     P predicate | |
|     predicate S predicate | |
|     predicate Sw predicate | |
|     predicate V predicate | |
|     predicate /\ predicate | |
|     ! predicate | |
|     (temp_expression) |
| predicate: |
|     pred_op | |
|     temp_expression |
| pred_op: |
|     operand == operand | |
|     operand >= operand | |
|     operand > operand | |
|     operand <= operand | |
|     operand < operand | |
|     (pred_op) | |
|     pred_op && pred_op | |
|     pred_op || pred_op | |
|     operand |
| operand: |
|     call | |
|     label | |
|     assign | |
|     VAR | |
|     INT |
| assign: |
|     VAR = FUNC_NAME | |
|     VAR = FUNC_NAME { VAR } |
| call: |
|     FUNC_NAME (func_param) | |
|     FUNC_NAME (func_param) {VAR} |
| func_param: |
|     param | |
|     param , func_param |
| param: |
|     VAR | |
Unfortunately, this isn’t sufficient for our new trace checker to work. As we have seen in previous examples, so far the edges of our automata have only been function calls and returns. Now we will also have expression-edges, which need to be evaluated at run-time. Moreover, it is possible to have multiple events happening at the same time. Let’s take the following formula:

\[ Gp \ foo(x, y) \to (Ep \ bar(x)) / y == 0; \]

It checks that whenever we see a call to \(foo(x, y)\), \(y\) must be zero and we must have previously seen a call to \(bar(x)\). We have three different predicates in our formula: \(foo(x, y)\), \(bar(x)\) and \(y == 0\). When encountering \(foo(x, y)\) both the first predicate and the last might be true. Therefore, the edges of our new automaton must now consist not of single predicates, but of subsets of predicates. We can see this new automaton for the above formula in Figure 5.13. Encountering only the third predicate (001) keeps the automaton in state 1, because we haven’t yet seen a call to \(bar(x)\). Any event involving \(bar(x)\) moves the automaton to state 2 where we are ready to encounter a call to \(foo(x, y)\) which also satisfies the \(y == 0\) predicate (101).

The downside of adding expressions is that the automaton becomes harder to read.

**Generating the edges**

When generating the edges for our automaton, we need to take into account all the subsets of the set of predicates that appear in our formula. One easy optimization which we have implemented is not to generate edges where two predicates which represent calls or returns of different functions are set to true, as that can never happen: we only have one function event at a time. However, we don’t do any checks for expressions, so we presume that any expression can become true alongside any function call. That’s why in the previous example, we have edges such as 011 or 001 which we won’t ever encounter.
5.5.7 Variable quantification

As stated earlier, every variable in our ptLTL formulae is universally quantified at top level; this fact can sometimes confuse the users, depending on the formula they want to write. Let’s consider the following formula:

\[ Gp \text{ call} f \implies Ep \text{ call} g \land y = 0; \]

As a first sight, it may seem that it checks that whenever we encounter a call to \( f \) we must have previously encountered a call to \( g \) with parameter \( 0 \). However, when running the checker on this formula, it evaluates every trace to false, because \( y \) is globally quantified. The formula checks in fact that for every possible \( y \) we encounter a call to \( g \) and every possible \( y \) is equal to \( 0 \). This is obviously false. There is a way to rewrite the above formula, such that is actually checks want we want:

\[ Gp \text{ call} f \implies Ep \text{ call} g / \ (g(y) > y == 0); \]

By adding the implication we have guarded the check \( y == 0 \) with an actual call to \( g \) which binds \( y \). Also, \( \text{call} g \) is necessary, because we are checking for an actual call site of \( g \) (otherwise, the implication becomes true when the current event isn’t a call to \( g \)).

5.5.8 Expressions using unbound variables

Another important aspect of our algorithm is the possible comparison between bound and unbound variables (we have described in section 5.5.1 the way variables from a formula are bound during a certain trace). Let’s consider the following formula:

\[ Gp f(x) - > (Gp g(y) - > x > y); \]

It tests that if we encounter a call to \( g \) before a call to \( f \), then \( f \)’s parameter must be greater than \( g \)’s parameter. The problem is that when we encounter a call to \( g \) and the rightmost implication is evaluated, the variable \( x \) is unbound. Hence, we cannot evaluate this expression and we would need to split the set of automaton instances that refer to the unbound \( x \) in two: one which satisfies the \( x > y \) inequality and one which does not.

Notice that although a comparison between a bound and an unbound variable can appear quite often, formulae can be written in such a way that the result of these comparisons doesn’t matter in the overall truth value. Let’s examine the formula:

\[ (Gp f(x) - > x == 2) / \ (Gp g(y) - > y == 2); \]

According with what we have seen in section 5.5.6, for each event we have to compute the truth values of all predicates of the formula, so we can determine the label of the edge in the automaton. Hence, when encountering a call to \( g \) which binds \( y \), we will have to evaluate \( x == 2 \), but its result is not relevant, due to the false value of the right hand side of the implication.

The first formula we’ve seen in this section can also be rewritten such that the comparison \( x > y \) is done at top level, after both variables are bound:

\[ (Gp f(x) \ V (Gp g(y)) \ V x > y; \]

This works, because when the value \( x > y \) actually matters, we know that we have already bound both \( x \) and \( y \).

However, our current checker cannot distinguish between these two cases, so it’s left up to the user to be sure there aren’t any such expressions present in the formula. The formula parser could possibly identify the usage of unbound variables and try to rewrite the formula, but this is left as further work.
5.6 Summary

In section 5.5, we have presented a total of three different checkers, as they evolved while we were adding more complex formulae to our system. To sum up, these checkers are:

- the **simple checker**, shown in section 5.5.3, which only supports having one variable and keeps a hash from this variable’s value to the state of the corresponding automaton instance

- the **multiple variable checker**, shown in section 5.5.4, which allows having an unlimited number of variables in our formulae. It automatically regresses back to the **simple checker** if it has to deal with only one variable

- the **expression checker**, shown in section 5.5.6, which allows mathematical expression regarding variables to be added (equality and inequalities)

- the **custom checker**, which is a particular checker that can be written for any desired property. However, it does not have the advantage of just specifying the formula which you want to be checked

To conclude, the overall steps taken by our tool when executing a program and verifying a property are:

1. the user, by calling the `instrument` and `instrument_call_site` functions from section 5.4, specify, in the program which is run, what data he desires to be included in the trace

2. the **symbol wrapper** might be executed in order to replace calls to `malloc` and `free` with wrappers which add the addresses to the trace

3. the program is executed and the trace decoding pipeline is run

4. the users enter the past time LTL formula they want to check; it is analyzed and converted to an automaton

5. the trace data output from the decoding pipeline is used to step the automaton and decide whether the program satisfies the formula
Chapter 6

Experiments & Results

Our experiments are divided in two categories: those run on the OMAP 4460 Pandaboard, testing the impact of tracing and the comparison with TESLA (sections 6.1, 6.2) and those run on a desktop machine (Intel Core i5 750 Processon, 2.66 GHz), testing the LTLFO2MON tool, our ptLTL checkers and the trace decoding pipeline (sections 6.3, 6.4, 6.5). Each of the experiments consists out of 10 runs, out of which the average (and in some cases the standard error) is shown.

6.1 Bare Metal

This subsection describes the experiments run on the bare-metal Pandaboard with the DSTREAM attached.

The first experiment we did was to verify that the DSTREAM’s program tracing is non-intrusive. Hence, the same simple program (shown in listing 6.1) was executed on the board, with and without tracing.

The second experiment tests the effect of checking the return value of a function call. This implies adding two extra calls per call to \texttt{foo}, one to \texttt{send\_return\_value}, for the actual return value, and another to \texttt{emit\_value}, for the value with which the check is made - these functions have been discussed in section 5.2.

We can observe the following results (number of runs is 10, no compiler optimization \texttt{-O0}):

- effect of trace on a simple calling function program (without assertions), Figure 6.1.
- effect of trace on a simple calling function program, with calls to our helper functions, Figure 6.2.

Listing 6.1: trace impact test program

```c
int id(int i) {
   return i;
}

void foo(int i) {
   id(i);
}

int main(int argc, char **argv) {
   int calls;
   sscanf(argv[1], "%d", &calls);
   long ticks = clock();
```
for (int i = 0; i < calls; i++) {
    foo(i);
}

printf("time: \%d\%d\%d\n", calls, clock() - ticks, CLOCKS_PER_SEC);
return 0;
}

Listing 6.2: assertion impact test program

int id(int i) {
    return i;
}

void foo(int) {
    int x = foo(i);
    send_return_value(x, 1);
    emit_value(i, 1);
}

int main(int argc, char **argv) {
    int calls;
    scanf(argv[1], "%d", &calls);
    long ticks = clock();
    for (int i = 0; i < calls; i++) {
        foo(i);
    }
    printf("time: \%d\%d\%d\n", calls, clock() - ticks, CLOCKS_PER_SEC);
    return 0;
}

We are able to see that in both cases the program with trace performs the same as the program without a trace, thus proving the non-intrusiveness of the tracing method. There are some small differences between the two running times, but they are covered by the standard error, which is also shown in the figures.

6.2 Linux

We compared our trace system with TESLA on Ubuntu 15.10 running on our Pandaboard OMAP 4460. Our goal was to see whether our instrumentation functions add a smaller overhead to the original program than TESLA’s. The analyzed program contains a number of calls to a simple, int-returning function, whose return value we want to check.

The running times of both tracing and TESLA can be seen in Figure 6.3. The lowest line, labeled simple, is the running time of the simple program shown in listing 6.1; as the time is very low compared with the others it is almost indistinguishable from the X axis. The line labeled with full represents adding the full tracing code to our program, including the system calls to actually change the value of the CONTEXTID register (this has been detailed in section 4.4). Clearly this
running time is much higher than the one represented by w/o syscall, in which the tracing function trace_value does not contain the system calls, but contains all other code necessary for tracing the data. We have analyzed both options because, although the use of system calls is essential to obtain the trace at this current moment, it should be possible to do without them, if we have a hardware which supports ITM [15], or any other protocol which has a native API of inserting data into the trace.

Even with the slowdown inflicted by the system calls, the running time of our tracing is still lower than that of the TESLA assertions (denoted by the w/ TESLA assertions) label. Moreover, we compared the slowdown created by the TESLA assertions with the one created by our tracing (full, with system calls) in which we can clearly see (Figure 6.4) that:

- our system performs better, creating less overhead
- the slowdown, after an initial stabilization period, becomes constant for both segments; the
initial spikes are probably caused by the influence of the initial loading of the address pages, which is more visible for smaller number of operations

Listing 6.3: TESLA test program

```c
int id(int i) {
    return i;
}

void foo(int) {
    int x = id(i);
    TESLA_WITHIN(foo, previously(id(ANY(int)) == 3));
}

int main(int argnr, char **argv) {
    int calls;
    sscanf(argv[1], "%d", &calls);

    long ticks = clock();

    for (int i = 0; i < calls; i++) {
        foo(3);
    }

    printf("time: %ld
", calls, clock() - ticks, CLOCKS_PER_SEC);
    return 0;
}
```

Figure 6.3: TESLA and tracing running times
6.3 LTLFO2MON

We tested the performance of the LTLFO2MON tool described in section 3.1 by running it against the trace produced by the program in listing 6.4 with the no double allocation and no double frees formulae presented in section 4.2. The program shown in listing 6.4 satisfies the two properties, so all the runs should validate the traces.

The results for no double frees \((\Box \forall \text{malloc}(x) \implies \text{add.in.set}(x)) \land (\Box \forall \text{free}(y) \implies \text{check.in.set}(y))\) are shown in Figure 6.5. The plot includes the different phases of the execution: formula & trace parsing and processing of the property. The results show that the time needed for formula parsing is far too small to contribute to the total time and that almost 75% of the time is spent doing the trace parsing.

Listing 6.4: malloc/free program

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv) {
    int calls;
    scanf(argv[1], "%d", &calls);

    long ticks = clock();

    for (int i = 0; i < calls / 2; i++) {
        addr[i] = malloc(4);
    }
    for (int i = 0; i < calls / 2; i++) {
        free(addr[i]);
    }

    printf("time:\%ld\n", calls, clock() - ticks);
}
```

Figure 6.4: Comparison between the slowdown of TESLA and assertions
The events measured on the X axis are mallocs and frees. The running time is not every impressive, moreover, for the no double allocation formula, on a trace of only 100 mallocs and frees, the running time is close to 8 seconds. Hence, this would not have been a suitable tool for further work.

6.4 Past Time LTL parser & checker

Continuing the experiments from the previous section, we took the same trace generated by the program in listing 6.4 and verified the no double frees property. As discussed in section 5.1 the property checker stage of the pipeline can be implemented with either a custom checker or using our more general ptLTL parser & checker (section 5.5). We expect that the custom checker will perform better than the ptLTL checker, but we still want to compare the two and to see if they are both a better option that the LTLFO2MON tool discussed previously.

We chose to test the no double frees property \((Gp \$free(x) -> P !\$free(x)) S x = $malloc;\), because the LTLFO2MON tool has the best performance on it. The automaton for no double frees has been shown in Figure 5.4. For testing, we have used the simple checker without any optimizations (-O0) in order to compare the slowest running time with the LTLFO2MON tool.

The compared custom checker, ptLTL checker and LTLFO2MON can be seen in table 6.1. As expected, the custom checker has a faster running time than the generic simple checker, but both of them are faster than the LTLFO2MON results from the previous section.

6.4.1 Comparison for all properties

We also ran the custom checker and the expression checker, compiled with optimizations (-O2) for all three malloc/free properties: non-interleaving mallocs, no double frees and no leaks. We ran these experiments in order to see the comparison between the two for all properties and to see the running times of the optimized executables, which are the ones we would like to use in the future. The results can be seen in Figure 6.6. The expression checker is configured to use a hash table if it contains only one variable. Again, as expected, the custom checker performs faster (roughly 40%) than the expression
Table 6.1: Comparison between LTLFO2MON, ptLTL checker and custom checker for \textit{no double frees}

checker: however, given the fact that the \textit{expression checker} is more general (users need only specify the ptLTL formula they want to check) we found this decrease in speed acceptable.

6.4.2 \textbf{expression checker without hash table}

As discussed in section 5.5.4 the complexity of the \textit{expression checker} is $O(N^3)$, where $N$ is the number of events in the trace. However, if the formula contains only one variable, we don’t need to use the \textit{unification tree} and we can fall back to the hash based method described in section 5.5.3, as we did in the previous section.

We also want to observe the performance of the \textit{unification tree} algorithm for the \texttt{malloc/free} properties, which can be seen in Figure 6.7. One can observe that all three properties have similar (almost indistinguishable) running times, which model the $O(N^3)$ complexity.

6.4.3 \textbf{Symbol wrapper}

We also want to see how using the \textit{symbol wrapper} described in section 5.3 influences our execution times. The advantage of the \textit{symbol wrapper} is that it generates smaller protobuf messages than when using normal instrumentation, because the \texttt{MALLOC} and \texttt{FREE} events and associated protobuf messages (section 5.1) are molded specifically for calls to \texttt{malloc} and \texttt{free} and don’t have any additional data.

The running times for the three properties are shown in Figure 6.8 and, indeed, are smaller than
the ones for the expression checker seen in Figure 6.6. Because of its lower running time, the symbol wrapper should be preferred over the LLVM pass when dealing with a large amount of function calls.

6.4.4 Multiple variables

We want to determine the impact of increasing the number of variables from the formulae. From analyzing the unification tree algorithm (section 5.5.4) we believe that the worst situation would be the one in which every event adds as many new nodes as possible in the unification tree. Hence, we ran experiments with the following traces:

1. $Gp \, $f(x) \rightarrow E \, $g(x);

2. $(Gp \, $f(x) \rightarrow E \, $g(x)) \setminus (Gp \, $a(y) \rightarrow E \, $b(y))$;
First, let’s consider a small example, whose trace can be seen in Table 6.2. Let’s analyze each unification tree for the above formulæ. In Figure 6.9 we have the tree for formula 1: two nodes for each instance of $x$; the other variables are ignored as they don’t appear in the formula. The unification tree for the second formula can be seen in Figure 6.10: the number of nodes has risen from 3 to 9: the $x$ variable still creates two nodes (the ones with binding (0) and (1)), but the $y$ variable, besides adding nodes (0) and (1), also descends into the nodes created by the $x$ variable and adds the nodes in which both variables are bound: (0 0), (0 1), (1 0) and (1 1). Figure 6.11 shows the unification tree for the third: it’s constructed in a similar manner as the previous one, but, because we now have three variables, its size grows to 27 nodes.

The experiments have been run with traces similar with the one in Table 6.2, but with a higher number of variable values. The running times (in seconds) can be seen in Table 6.3.

As expected, running time increases exponentially together with the number of variables involved. However, the important observation here is that not just the number of variables increases the running time, but the fact that each event only contains one of the three variables, thus creating a binding in which the other two variables are unbound. This causes the binding to match a lot of nodes in the unification tree, increasing its size.

If, on the other hand, we have the following formula: $4. Gp$ $f(x, y, z) \rightarrow Ep$ $g(x, y, z)$; then this will act similar to formula 1 from above, as seen in Table 6.4.

### 6.5 Trace decoding pipeline

In the first implementation of the decoding pipeline (described in section 5.1) each stage writes its output protobuf message at the end of its execution, message which is then read by the next phase of the pipeline. The running time of each stage is shown in Figure 6.12, including the
Figure 6.9: unification tree for formula 1

Figure 6.10: unification tree for formula 2
Table 6.4: Execution time multiple bound variables

<table>
<thead>
<tr>
<th>Number of values</th>
<th>formula 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.001</td>
</tr>
<tr>
<td>100</td>
<td>0.001</td>
</tr>
<tr>
<td>200</td>
<td>0.003</td>
</tr>
<tr>
<td>300</td>
<td>0.004</td>
</tr>
<tr>
<td>400</td>
<td>0.006</td>
</tr>
<tr>
<td>500</td>
<td>0.008</td>
</tr>
</tbody>
</table>

time it takes to write the data for the next stage. The last phase, that of checking the property, is not shown because it was discussed in section 6.4. We notice that the most time consuming stage is the Assembly Parser, due to the fact that it is to analyze every assembly instruction of the executable, which involves repeatedly inspecting the ELF file. Also, we notice a non-linear increase in the running time for the ASM w/ write after 1,500,000 events; this is probably due to the size increase in the output protocol buffer, which adds an overhead either in the generated code from protoc or because of the caches used for write not being large enough. The should see this increase disappear when using the symbol wrapper, because it outputs a smaller protobuf file.

In the second implementation of the pipeline we sought replace the writing of the whole result at the end of each stage with writing partial results during the execution, so that a later stage of the pipeline can begin working before a previous stage has finished. We needed to determine the optimal size of the data at which should be sent to the next stage. The analyzed sizes and associated running times for a trace containing 100,000 mallocs and frees can be seen in Figure 6.13, in which we can see both the added execution times of all the stages and the overall execution time of the pipeline (which is the one that interests us). As observed, the best performance is achieved with a size of 10MB.

Hence, we ran our pipeline with a size of 10MB on the same dataset as the experiment shown in Figure 6.12 and plotted the execution times in Figure 6.14. As before, we show the running time of each individual stage and the overall execution of the pipeline, which is less than the sum of the individual stages because now we have different stages working at the same time.

### 6.5.1 Symbol wrapper

We decided to trace the running times of the two different pipelines described previously when using the symbol wrapper discussed in section 5.3 which can be seen in Figures 6.15 and 6.16. Of course, in both cases the running times are smaller, because the generated trace and the intermediary protobuf messages are smaller. As expected, we observe the disappearance of the non-linear increase at 1,500,000 events, proving that it was caused by the size of the output file.

To conclude, the symbol wrapper performs better both in the case of the trace decoding pipeline and the ptLTL checkers. However, this situation is similar with the one between the custom checker and the ptLTL checker: it does perform better, but it loses the generality offered by the other tool.
Figure 6.11: unification tree for formula 3
Figure 6.12: Performance of pipeline stages

Figure 6.13: Execution time for different data sizes
Figure 6.14: Performance of pipeline stages with 10MB buffer

Figure 6.15: Performance of pipeline stages with symbol wrapper
6.6 Summary

In this section, we have proven, through experiments, that:

- the trace captured from the DSTREAM is non-intrusive
- the overhead from adding our tracing functions is less than the overhead caused by TESLA
- our past time LTL checker is faster than the LTLFO2MON tool
- the trace decoding pipeline works best for message sizes of 10MB
- the symbol wrapper performs better both when using the ptLTL checker and the trace decoding pipeline

Overall, we are satisfied with the performance of our tool and we believe that further work might improve it even more.
Chapter 7

Conclusions & Further Work

In this paper we have presented a proof of concept system that allows a user to specify properties to check using a past time linear temporal logic language and then verifies these properties using the captured program trace from the ARM DSTREAM debug unit. Our system has three interesting features that stand out. Firstly, the trace capturing is done non-intrusively, so the performance of the executed program is not affected (in opposition to other similar property checking tools such as TESLA). Secondly, the user can specify the properties we want to check using a well-defined syntax built upon past time linear temporal logic formulae. We developed an algorithm that transforms these ptLTL formulae into an automaton, which can then be executed, its transitions being events that are taken from the program’s trace. This is a very important part of our system, because it allows numerous formulae to be checked on a given trace, without modifying the program or running it again. Lastly, the effective checking of the formulae is done offline, which drastically decreases the slowdown of the executed program.

7.1 Further work

There are a number of things which we think should be improved with our tool, of various levels of difficulty.

Running it on other platforms

The Pandaboard we used is a very small and simple board which, although proved useful for our development, is not used in rack-scale computers. Also, the small size of the ETB buffer (8KB) used for storing the trace makes it impossible to debug more complex programs. Towards the end of our work, we obtained a Xilinx zc706 board, which allows the connection of a High Speed Serial Trace Probe (HSSTP [16]); it circumvents the usage of an ETB buffer and supports online streaming of the captured trace.

Debugging kernel modules

Running only bare-metal projects limits the complexity of the applications which can be traced and for which we can verify properties. The possibility of obtaining a trace from kernel modules or other parts of the Linux kernel would be tremendously useful and could potentially create ideas for new properties to check.

Adding existential operators to our past time LTL formulae

Currently, all variables that appear in our ptLTL formulae are globally quantified, which may lead to interpretations of the formulae which may not seem natural (as we have seen in section 5.5.7);
adding existential quantifiers and integrating them in the unification algorithm and the automaton checker should be a very big step towards having a more clear and expressive logic.

**ITM**

The Pandaboard we have worked with only supports a PTM trace format which, as we have seen in section 4.4 does not have a native way of adding data to our instruction trace; therefore we were required to insert data in the CONTEXTID register and use Context ID packets to retrieve it. Other tracing formats, such as ITM [15], allow embedding of data inside the trace, which would allow a cleaner implementation and reduce the overhead in the executed program created by our current `trace_value` function and its subsequent system calls.

**Improving the trace decoding pipeline**

Currently, the trace decoding pipeline was implemented in the simplest of ways, without spending time on thinking of optimizations. Certainly, its running time can be improved, especially for the assembly phase, in which instructions and branches for the ELF file could potentially be cached.

**Extending the temporal logic**

We currently support only using past time LTL operators in our formulae. Hence, users cannot specify properties that happen in the future (such as using the `eventually` operator, like in TESLA); adding such a feature could certainly improve our system.
Appendices
Appendix A

DS-5 Utilization

The way to do this is by selecting “Debug Configuration” from the “Run” menu of DS-5 and selecting your desired target. DS-5 supports debugging either a physical board or a Fixed Virtual Platform, which is a simulation of a processor. A user can choose between doing a bare-metal debug (running the program without an underlying OS), a Linux application debug or a Linux kernel debug. DS-5 comes with some pre-installed FVPs, with processors ranging from ARMv8 to Cortex A9.

In the “Debug Configurations” menu the “Connections” tab is used for selecting the above mentioned debug platform and mode and for specifying the “Debug and Trace Service Layer” (DTSL) configuration.

A.1 FVP simulation

From the “Connection” tab select “ARM FVP (Installed with DS-5)” - “VE_Cortex_A9x1” - “Bare Metal Debug” - “Debug Cortex A9”. The “DTSL Options” allow the user to select the size of the trace buffer (16, 32, 64 or 128 MB) and the behavior of the trace (if it starts on connect, etc.). The “Files” tab is used for specifying which .axf file is loaded on the FVP. The “Debugger” tab is used for specifying run control options: if the debugger should connect only, if it should debug from an entry point or if it should debug from a symbol. Debugger commands can also be specified in this tab.

A.2 Pandaboard OMAP 4460

From the “Connection” tab select “pandaboard.org” - “OMAP 4460 (Pandaboard ES)” - “Bare Metal Debug” - “Debug Cortex A9_0”. The DSTREAM must be connected both to the Pandaboard and to the host machine running DS-5. The “DSTREAM” option must be selected from the “Target Connection” menu and the DSTREAM connection must be entered in the “Connections” menu: this can be made either via USB or by TCP. The “DTSL Options” allow the user to select the ETB buffer as the trace capture method. The “Cortex-A9” tab of the same “DTSL Options” allows the user to enable the actual trace and to enable certain features, such as PTM context IDs, Cycle Accurate tracing and trace capture ranges. The “Files” and “Debugger” tabs are used in the same way as in the FVP simulation.
Appendix B

PTM trace packets

As we described in section 2.2, branch instructions are divided in two: direct and indirect branches. Direct branches:

- **B** - unconditional branch
- **B<cc>** - conditional branch
- **BL** - branch and link
- **BLX <imm>** - branch & link with exchange
- **ISB** - instruction synchronization barrier
- **DMB** - data memory barrier
- **DSB** - data synchronization barrier

Indirect branches:

- **RFE** - return from exception
- **BX** - branch and exchange
- **BLX <reg>** - branch with link and exchange
- **BXJ** - branch and exchange to Jazelle
- **LDR or LDRT to PC** - load word to PC
- **LDM including the PC** - load multiple to PC
- **ERET** - exception return

The trace can be configured to use cycle-accurate tracing. In this case, some packets will include the number of cycles since the last packet containing cycle information (more information in the PTM manual [1], section 4.4). We won’t provide more details about this, because we have not used this information in our work.

Besides accurate-cycle tracing, the PTM output may contain information regarding the context IDs of the debugged processes. This packet will prove to be important to our system, because, although the register’s size is 32 bits, only 8 bits are used as the Address Space ID, leaving the remaining 24 bits to be populated however the user desires (more details about what information we pass through this register can be found in section 4.4).
B.0.1 A-sync packet

The A-sync packet is the first packet of the trace and it might also be output periodically. Therefore, besides proving that the binary file we are dealing with is actually a PTM-encoded trace, it has no other use. The format can be seen in Figure B.1.

![Figure B.1: ASYNC packet format](image)

B.0.2 I-sync packet

The instruction synchronization packet is output periodically. It is also generated each time the trace is turned on, which might happen multiple times if the configuration specifies regions of the program (and not the whole) to trace. In that case, the trace will be turned on each time the execution enters one of the regions, and turned off each time is leaves.

Besides specifying the address, the I-sync packet (Figure B.2) can also output the context ID of the process currently run. This is very useful, as, later on, we’ll see that we can populate the CONTEXTID register with data that interests us.

![Figure B.2: ISYNC packet format](image)

B.0.3 Atom packet

An atom packet is generated each time the program encounters a direct branch. Because direct branches contain the destination address in their encoding, all the decoder needs to know is if the branch was taken or not: an E atom indicates the waypoint instruction changed the flow of the program and N indicates it did not. In Figure B.3 we can see the format of the atom packet. In case cycle-accurate tracing is not enabled, one packet can contain from 1 up to 5 atoms (to reduce the size of the trace); Figure B.4 shows all 5 variants and how they are decoded.
A Branch packet is generated when an indirect branch or an exception is encountered or when the processor security changes. The format of the branch packet is shown in Figure B.5. The target address has from 1 to 5 bytes, because it is compressed: it contains only the bytes that have changed compared to the address in the last Branch or I-Sync packet.

If the exception information is present, then the field Exception[3:0] describes the exception call. The values for this field can be found in the PTM manual [1], section 4.5.4.

---

**Figure B.5: Branch packet format**

---

<table>
<thead>
<tr>
<th>Format</th>
<th>Encoding</th>
<th>Meaninga</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format 0</td>
<td>10000000</td>
<td>Reserved</td>
</tr>
<tr>
<td>Format 1</td>
<td>10000100</td>
<td>One atom, F</td>
</tr>
<tr>
<td>Format 2</td>
<td>10001100</td>
<td>Two atoms, FF</td>
</tr>
<tr>
<td>Format 3</td>
<td>10011100</td>
<td>Three atoms, FFF</td>
</tr>
<tr>
<td>Format 4</td>
<td>10111100</td>
<td>Four atoms, FFFFF</td>
</tr>
<tr>
<td>Format 5</td>
<td>11111100</td>
<td>Five atoms, FFFFF</td>
</tr>
</tbody>
</table>

---

a. F is 0 for an E atom and 1 for an N atom. The least-significant F is the most recent waypoint.
Because of the compression of the address, the branch packet can have multiple formats, which can be seen in Figures B.6, B.7, B.8, B.9 and B.10. These formats are the ones in which there is no exception information, recognizable by the 6th bit of the last byte being set to 0. In case exception information is present, then there must be at least 2 address bytes and the 6th bit of the last address byte is set to 1 (Figures B.11, B.12, B.13 and B.14). Although these different packet formats are not hard to understand, not taking all of them into account might cause a failure when decoding the trace.

The trace generator might use a return stack, in order to reduce the number of Branch packets and replace them with Atom packets. When a branch with link instruction takes place, its address is put onto the return stack; this address can be later popped from the stack, if we encounter an Atom packet associated with an indirect branch. BL, BLX <reg> and BLX <immed> can be used for pushing an address onto the stack. For more details, see the PTM manual [1], section 4.13.
B.0.5 Waypoint

An exception which occurs in the program’s execution might cause a branch instruction which cannot be determined from statically analyzing the program. Therefore, a Waypoint packet will contain the address of the last non-waypoint instruction that has been executed, i.e. the point of the program reached before the exception. Similar to the Branch packet discussed previously, the address might be compressed, as seen in Figure B.15.

![Waypoint format](image)

Figure B.15: Waypoint format

B.0.6 Context ID

A Context ID packet is generated when the value of the CONTEXTID register changes. The value might contain from 1 to 4 bytes, depending on the configuration of the debugger. As mentioned earlier, this packet will prove to be important when we add data to our trace (section 4.4).

![Context ID packet format](image)

Figure B.16: Context ID packet format
Appendix C

DS-5 debugger launch script

```bash
#!/bin/bash

if ! [ $# -eq 4 ]; then
    echo "$0 start_address end_address out_dir args"
    exit
fi

ENTRY="pandaboard.org::OMAP_4460(Pandaboard_ES)::BareMetal_Debug::BareMetal_Debug::Debug_Cortex-A9_0::DSTREAM"
CONN="Connection=USB:005210"
IMAGE="~/DS-5-Workspace/Panda2/Debug/Panda2.axf"

echo """ break 0x$1
trace clear
run $4
wait 1s
trace start
continue
wait
trace stop
trace dump $3 PTM_0 2
trace report FILE=report.txt"" > script.txt

rm -rf $3

/usr/local/DS-5_v5.24.1/bin/debugger \
  --cdb-entry "$ENTRY" \
  --cdb-entry-param "$CONN" \
  --cdb-entry-param "$DTSL_OPTIONS" \
  -b $IMAGE \
  -s script.txt

rm script.txt
```
Appendix D

LTLFO2MON syntax

LTLFO2MON processes an LTLFO formula which can have the following syntax:

\[ f := ( f ) | \neg f | f /\ f | f \setminus f | f \rightarrow f | f \leftrightarrow f | G f | F f | X f | f U f | f W f | A (x_1, \ldots, x_n): p. f | E (x_1, \ldots, x_n): p. f | p(t_1, \ldots, t_n) | r(t_1, \ldots, t_n) | \text{true} | \text{false} \]

The operators have the following meaning:

- **G** - Globally
- **F** - Eventually
- **X** - Next
- **U** - Until
- **W** - Weakly Until
- **A** - for all
- **E** - exists
- **p(t_1, \ldots, t_n)** - U-Operators; they become true by their appearance in the trace
- **p(t_1, \ldots, t_n)** - I-Operators; they cannot appear in the trace and are evaluated by the monitor
Appendix E

Tracing Library

```c
#ifndef CHECK_FUNCTIONS_HPP
#define CHECK_FUNCTIONS_HPP

#define RET 0x20
#define CALL 0xe0
#define MALLOC 0x40
#define FREE 0x60
#define CHECK 0x80
#define RETURN_VAL 0xa0
#define ARG 0xc0

void my_isb();

void trace_value(unsigned int p, unsigned char event, unsigned char info) {
    unsigned short high_id = (unsigned short)event << 8,
    low_id = (unsigned short)info << 8;

    unsigned int val = 0;
    // read context id
    asm volatile("mrc p15,0,%0,c13,c0,1": "r"(val));
    val &= 0xff;

    unsigned int new_val = val | ((p & 0xffff) << 16) | high_id;
    asm volatile("mcr p15,0,%0,c13,c0,1": "r"(new_val));
    my_isb();

    new_val = val | (p & 0xffffffff) | low_id;
    asm volatile("mcr p15,0,%0,c13,c0,1": "r"(new_val));
    my_isb();

    asm volatile("mcr p15,0,%0,c13,c0,1": "r"(val));
}

void check_ret(char *s, unsigned int val) {
```
void instrument_call_site(int a, ...) {
}

void instrument(char *s, ...) {
}

void send_return_value(unsigned int val, unsigned short id) {
    trace_value(val, RETURN_VAL, id);
}

void emit_value(unsigned int val, unsigned short id) {
    trace_value(val, CHECK, id);
}

void emit_call(unsigned short id) {
    trace_value(0, CALL, id);
}

void emit_ret(unsigned int val, unsigned short id) {
    trace_value(val, RET, id);
}

void send_arg(unsigned int val, unsigned short pos) {
    trace_value(val, ARG, pos);
}

void* my_malloc(int size) {
    void *ret = malloc(size);
    trace_value((unsigned int)ret, MALLOC, 0xff);
    return ret;
}

void my_free(void *p) {
    free(p);
    trace_value((unsigned int)p, FREE, 0xff);
}

#ifndef ...
Appendix F

Custom Checkers

```cpp
#include <cstdio>
#include <fstream>
#include <vector>
#include <algorithm>
#include <unordered_map>
#include <ctime>

using namespace std;

#include "proto_code/trace.pb.h"

using namespace trace;

const int MAX_ID = 15000000;
const unsigned int BASE_ADDR = 0x80000000;

// unordered_map<int, uint32_t> calls;
uint32_t calls[MAX_ID];
bool id_present[MAX_ID];

#define OVERLAPPING_MALLOCS 0
#define CONSISTENT_FREES 1
#define NO_LEAKS 2

bool malloc_addr[10000000], free_addr[10000000];

vector<uint32_t> mallocs, frees;

set<uint32_t> current_addresses;
int type;

void do_malloc(uint32_t addr) {
    if (type == OVERLAPPING_MALLOCS) {
        if (current_addresses.find(addr - BASE_ADDR) != current_addresses.end()) {
            printf("FAILED\n");
            exit(0);
        }
    }
}
```
void do_free(uint32_t addr) {
    free_addr[addr - BASE_ADDR] = 1;
    if (type == CONSISTENT_FREES) {
        if (current_addresses.find(addr - BASE_ADDR) == current_addresses.end()) {
            printf("FAILED\n");
            exit(0);
        }
    }
    frees.push_back(addr - BASE_ADDR);
    current_addresses.erase(addr - BASE_ADDR);
}

int main(int argc, char **argv) {
    if (argc != 3) {
        printf(stderr, "usage: %s trace_file type:[0|1|2]\n", argv[0]);
        exit(0);
    }
    clock_t start = clock();
    scanf(argv[2], "%d", &type);
    fstream input(argv[1], ios::in | ios::binary);
    Trace trace;
    trace.ParseFromIstream(&input);
    for (int i = 0; i < trace.events_size(); i++) {
        int type = trace.events(i).type();
        uint32_t addr;
        switch (type) {
        case Event_Type_RET: {
            if (trace.events(i).ret().name() == "$malloc") {
                do_malloc(trace.events(i).ret().val());
            }
        } break;
        case Event_Type_MALLOC: {
            do_malloc(trace.events(i).malloc().addr());
        } break;
        case Event_Type_CALL: {
            if (trace.events(i).call().name() == "$free") {
                do_free(trace.events(i).call().params(0).param_val());
            }
        } break;
        case Event_Type_FREE: {
            do_free(trace.events(i).free().addr());
        } break;
    }
if (type != NO_LEAKS) {
    printf("PASSED\n");
    fprintf(stderr, "%f\n", (double)(clock() - start) / CLOCKS_PER_SEC);
    exit(0);
}

if (mallocs.size() != frees.size()) {
    printf("FAILED: \text{number of } \text{mallocs} \text{ don't match } \text{number of } \text{frees}: \%d vs \%d\n", mallocs.size(), frees.size());
}

bool bad = false;
for (auto i = mallocs.begin(); i != mallocs.end(); i++) {
    if (!free_addr[*i]) {
        printf("FAILED: \text{value at } 0x%x \text{ not freed}\n", *i);
        bad = true;
        break;
    } else {
        printf("0x%x checked\n", *i);
    }
}

for (auto i = frees.begin(); i != frees.end(); i++) {
    if (!malloc_addr[*i]) {
        printf("FAILED: \text{value at } 0x%x \text{ not malloc-ed}\n", *i);
        bad = true;
        break;
    } else {
        printf("0x%x checked\n", *i);
    }
}

if (!bad) {
    printf("PASSED: all \text{allocated values are being freed}\n");
}

fprintf(stderr, "%f\n", (double)(clock() - start) / CLOCKS_PER_SEC);
return 0;
}
Bibliography

[6] Andreas Bauer, Jan-Christoph Küster, Gil Vegliach The ins and outs of first-order runtime verification
[7] Andreas Bauer, Jan-Christoph Küster, Gil Vegliach From propositional to first-order monitoring
[8] Andreas Bauer, Martin Leucker, Christian Schallhart Runtime Verification for LTL and TLTL
Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

PROGRAM TRACE CAPTURE AND ANALYSIS FOR ARM

Authored by (in block letters):
For papers written by groups the names of all authors are required:

Name(s):  
Pârvu

First name(s):  
Andrei

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date  
Zurich, 15.09.2016

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.