Compact Traceable Logging

I.S.W.B. Prasetya, Ales Šturala, Arie Middelkoop, Jurriaan Hage, Alexander B. Elyasov
Dept. of Inf. and Comp. Sciences, Utrecht Univ.
Utrecht, the Netherlands
Email: ales.sturala@hotmail.com, {A.Elyasov,S.W.B.Prasetya,J.Hage}@uu.nl, amiddelk@gmail.com

***Abstract*—Logging is a commonly employed technique to gather information about the dynamic behaviour of a program. The resulting logs can be analysed to derive statistics, infer models, to diagnose failures, and used for testing. Balancing the cost of logging (in terms of I/O time and disk usage) and the benefits of increasing logging details is a challenging task. In this paper, we present a source code transformation scheme that converts the given program with ordinary logging to enhance it with tracing information, and at the same time significantly reduces the size of the generated logs by applying a form of binary encoding. Decoders are generated to interpret the logs and establish how the executions that produced them flowed through relevant decision branches in the program. This paper describes the transformation for sizeable subset of sequential Java, including its complicated control structures. As a proof of concept, we have implemented a prototype.**

Keywords-software logging; logging; software tracing; tracing.

I. Introduction

With the rise of complexity of modern applications, it becomes impossible to fully anticipate their behavior prior to deployment. Many applications employ logging to provide diagnostic information about their dynamic behavior. It provides means to at least diagnose erroneous and unexpected behavior after the fact [1], [2], [3], and may even allow us to reconstruct the original execution. Other kinds of information can also be learned from logs, e.g., usage statistics, usage patterns [4], code coverage, profiling information, potential security breaches [5], and even behavior models and specifications [6], [7], [8]; the latter also imply that test-cases can thus be learned from logs as well.

Clearly, the information that can be extracted from logs ultimately depends on what is exactly logged. Ideally, given a program *P*, we want to generate *executable* and *reproducible* logs. Such a log can be interpreted (executed) to drive an execution of *P*, and will at least reproduce the same log. Being able to re-execute allows common debugging tools to be employed to diagnose the log. Unfortunately, most logs are not executable, let alone reproducible. A *traceable* log allows us to infer how *P*'s execution flowed when it produced the log. It is a weaker property (than executability), but the inferred information can still be useful for error diagnosis. However, the amount of information that must be stored in the logs to make them traceable is substantial, which ultimately slows down application execution, and leads to very large files.

Saving can be gained by not logging pieces of text, but instead encoding them as a (short) binary/bitstring that refers to the location in the program that produced it. Essentially, these bitstrings can be considered as indices into an array of log comments. Our contribution is as follows. We present a new binary encoding scheme. The basic idea is to choose this bitstring, such that so that it also encodes a fragment of control flow, and thus providing tracing information without costing us (many) additional bits –note that whatever representation we use, I/O overhead and disk usage can still be further reduced by post-compressing the resulting logs. Our approach works by transforming the statements of the original program, so that they produce the bitstrings that encode their control flows. The transformation also produces the corresponding decoders to interpret the produced bitstrings and reconstruct normal logs, enhanced with tracing information. The produced bitstring logs are significantly more compact; in our experiments, they are 2.5 - 40 times smaller than normal logs, while enhancing the logs (after decoding) by a factor of 3 - 11. The transformation-based approach also implies that our logging scheme is *transparent* to the programmers: they can in principle write their logging code as they always do, after which our transformation will extend them for free.

The problem is however non-trivial. Modern programming languages support a whole range of control constructs, e.g., switches and breaks, which trigger a non-standard execution flow that is hard to encode faithfully. We also have to deal with exceptions and external call-backs; they dynamically disrupt the normal execution flow, and thus disrupt the encoding. In this paper, we will address some of these constructs. As a proof of concept, we have built a prototype implementing the approach in Java. It covers more constructs than those mentioned here.

This paper is structured as follows. Section II defines what kind of tracing information we want to add. Section III explains the basic idea of our transformation-based approach. Section IV explains the transformation of basic control structures, such as **while**, but also **switch** and **break**. Exceptions and external call-backs are handled separately in Section V. Section VI shows some experiment results of our prototype. Section VII discusses related work. Section VIII concludes and mentions future work.

II. Logging and Traceability

As an example, let us consider the program in Figure 1, consisting of just a single method, for sending some hello greetings. For logging, the primitive to use is $log(s, v)$; it writes the pair (s, v) to the log, where s is a descriptive string, and v is a value. The string s is static (its value does not depend on the program's state), whereas v is a dynamic value. The variant $log(s)$ can be used if we have no dynamic value to log.

```
void hello(int i) {
  while(i>0){
    if (decide(i)) {
      log("Sending...") ;
      send("hello world") ; }
    if(even(i)) i=i/2 ;
    else i-- ;
  }
  log("Done,i=",i) ;
}
```

Figure 1. The method `hello`.

An example of a log produced by the program is shown below:

```
Sending...Sending...Sending...Done,i=0
```

This log does not really tell us how the execution went through the program. Obviously, we can extend the logging to also trace the flow of control. But how can we do this without excessively increasing the overhead? Furthermore, we also notice that most characters in the above log actually belong to static strings. As suggested in Section I, they can be compacted, but can we combine compaction with tracing?

Let us first define what kind of tracing information we want to add. Let P be a single threaded target program. To help us defining our concepts, imagine G_P to be P's total control flow graph (CFG). If e is an edge in G, $m{\rightarrow}n$ denotes its pair of source and target.

We assume that every node in G_P has at most one call to $log(..)$ (else we split the node). Such a node is called a *logging node*. An edge leading to such a node is called a *logging edge*. G_P's initial nodes are assumed to be non-logging.

Given an execution of P, its *history* is an imaginary log obtained as follows: (1) when the execution passes a logging edge e that calls $log(s, v)$, we append the tuple $(line(e), s, v)$ to the history, where $line(e)$ is the line number of e in P; (2) when the execution passes a non-logging edge f we append $line(f)$. Histories represent logging enhanced with maximum tracing information. Because the overhead to actually log them is usually too large, we will only log *partial* histories; but which edges are useful to be included?

A *sibling* of an edge $e = m{\rightarrow}n$ is another edge $m{\rightarrow}n'$ in G_P with the same source, but a different target. A *branch* is an edge that has siblings. A *path* in G_P is a sequence σ of edges, such that σ_i and σ_{i+1} are two connecting edges in G_P, with the same direction. A *full path* is a finite path that starts from an initial node of G_P and ends in a terminal node (or a node marked as an end-node if P is intended to run forever). An edge e *can reach* another edge f if there is a full path with $[e, f]$ as a subsequence. It *can avoid* f if there is a full path that passes e, but $[e, f]$ is not a subsequence of this path. A branch e is an *attractor* of f if it can reach f, and it has a sibling d that can avoid f. Conversely, d is a *distractor* of f if it can avoid f and it has a sibling that can reach f. So, an attractor always has at least one distractor as a sibling, and vice versa.

Let us, at least for now, decide to log this kind of edges:

Definition 1: An edge is *log-relevant* if it is either: (1) a logging edge, (2) an attractor of a logging-edge, or (3) a distractor of a logging-edge. □

So, a log-relevant edge is either itself a logging edge, or a branch that has been decisive towards reaching or not reaching a logging edge.

Note that passed attractors and distractors cannot in general be inferred back from the normal log (the one without tracing information). For example:

```
if(g){
  if(h) return ;
}
log(s)
```

The two `else`-branches above are two attractors towards reaching $log(s)$. When we see s in the log, we cannot from s itself tell which attractor was taken.

We also want to note that in a language with exceptions the above definition of log-relevance may become impractical; we will return to this issue later.

III. Transformation, the Basic Idea

We will first transform P to its so-called *tagged* version $\ominus P$, and deploy the latter. Rather than producing a normal log, $\ominus P$ produces a *binary trace* (or simply *trace*) which is a bitstring interspersed by $\langle v \rangle$ values. The transformation also constructs decoders, which are later used to decode/interpret the produced trace to reconstruct the corresponding normal log, enhanced with human-readable tracing information. The latter log is called *enhanced log*.

To minimize the amount of logged data, in $\ominus P$ we suppress the logging of static strings. So, calls to $log(s, v)$ in P are replaced by $log(v)$ in $\ominus P$, and calls to $log(s)$ are removed. The decoders will later reconstruct them from the given trace.

Definition 2: A *decision node m* is a node with outgoing branches. Such a node is *log-decisive* if it has an outgoing branch e that is an attractor of some logging-edge f (which also implies that e has a distractor as a sibling). □

For example, the guard of the **while** and the first **if** in Figure 1 are log-decisive. The guard of the 2nd **if** is not.

In $\ominus P$ we 'tag' all log-decisive nodes m. Some additional code is injected in m: (1) to assign a unique bitstring to each outgoing edge of m, and (2) writes this bitstring to the trace when the edge is passed. So, if m only has two branches, then a single bit is sufficient to distinguish them. In Figure 2 we show the tagged version of the program in Figure 1.

```
void hello(int i) {
 while(tag(i>0)){
  if (tag(decide(i))) {
    send("hello world") ;
  if (even(i)) i=i/2 ; else i-- ;
 } ;
 log(i)
}
```

Figure 2. The tagged version of hello.

The function $tag(e)$ evaluates the Boolean expression e and returns the value; but as side effect it also writes the bit 1 to the log if e is true, and else 0. Effectively, this performs the tagging as meant above.

Notice that the static strings have been removed. Calls to $log(s, v)$ are converted to $log(v)$ that writes $\langle v\rangle$ to the trace. The execution that previously produced the log shown below Figure 1 now produces the following trace:

```
1111110⟨0⟩
```

The above string encodes which log-relevant edges were passed by the execution. The string is given to a decoder to produce the corresponding enhanced log. The decoder for hello is shown in Figure 3.

```
void dec_hello() {
  while (pop(3,9))
      if (pop(4,6))
          emit(5,"Sending...") ;
  emit(9,"Done,i=") }
```

Figure 3. *hello*'s decoder.

The original control flow of P is reflected by the decoder. We also include relevant line numbers information into the decoder. For each log-decisive node v in P (let's assume it only has two branches), the corresponding decision node v' in the decoder calls $pop(t, f)$. This pops the current bit b in the given trace. This bit tells us which branches of v that P took. The method $pop(t, f)$ simply returns the same bit, and this causes the decoder to follow the same branch at v'. The t and f are line numbers of v's branches in P; and they will be printed to the log accordingly by the decoder as tracing information.

Calls to $log(s, v)$ or $log(s)$ at line k in P are translated to $emit(k, s)$ in the decoder. When executed, it consumes the current $\langle v\rangle$ in the trace, if there is any; then writes either (s, v) or just s to the enhanced log, and decorating it by the line number k.

Other details of P are not carried over to the decoder.

Applying the decoder in Figure 3 to the above example trace produces the following enhanced log:

```
[Hello:2][Hello:3][Hello:4]@Sending...
[Hello:2][Hello:3][Hello:4]@Sending...
[Hello:2][Hello:3][Hello:4]@Sending...
[Hello:2][Hello:9]@Done,i=<0>
```

Once such a log is created, another tool could be written to actually step through the source code in an IDE, by following the tracing information.

Things can however get more complicated. The decoder above will actually be incorrect if $decide(i)$ also does logging, or if it throws an exception. Further adjustments are thus still needed. The next section describes the transformation in more details.

IV. Basic Transformations

More generally, P may have multiple methods. Only log-relevant methods need to be tagged, and for each such method a matching decoder should be generated. In the sequel, we will use the term 'statement' and 'expression' interchangeably. For a statement S, $\ominus S$ denotes its tagged version, and $\sharp S$ the corresponding decoder.

Recall that we want to tag log-decisive nodes. To decide which nodes are log-decisive, we do not actually want to construct G_P, since such a graph can be pretty large and is cumbersome to work with. We prefer to implement the transformation in a syntax directed way. We, therefore, decide the tagging based on the log-relevance of the corresponding statement, which is calculated syntactically:

Definition 3: A statement S is log-relevant, denoted by $S \in \mathbb{L}$, if it contains a call to $log(..)$ or a call to a log-relevant method. □.

Definition 4: A method m is log-relevant, denoted by $S \in \mathbb{L}$, if its body contains a call to $log(..)$ or a call to a another log-relevant method. □

The transformations are described by a set of transformation rules, which are denoted as the example in (1).

$$\textsc{val/var}\ \frac{v}{v \quad \epsilon} \quad v \text{ is a constant or variable} \tag{1}$$

The rule is named val/var, and specifies how constants and variables are transformed. The result of the transformation is a pair, as specified under the line; the left specifies the resulting tagged version, and the right one specifies the decoder. So, the rule above says that a constant or variable v is copied to the tagged program, but is removed (denoted by ϵ) from the decoder.

The rule to transform a whole method m is shown in (2). Only log-relevant m's need to be transformed. The decoder is another method named $dec\_m()$. It has no formal parameter, though implicitly it takes a trace as its input.

$$\textsc{mdef}\mathrm{L}\ \frac{\textbf{def}\ m(e_1...e_k)\ S}{\textbf{def}\ m(e_1...e_k)\ \ominus S \quad \textbf{def}\ dec\_m()\ \sharp S} \quad m \in \mathbb{L} \tag{2}$$

Some of the rules to transform the body S are shown in Figure 4. They only deal with standard constructs; more complex constructs are discussed later.

The rule log causes the logging of the static string s to be suppressed in the tagged version. The decoder on the other hand, does $emit(l, s)$, where l is the line number of the call $log(s, v)$ in the original program.

$$\textsc{log}\ \frac{log(s,v)}{log(v) \quad emit(linenr,s)}$$

$$\textsc{if}\mathrm{N}\ \frac{\textbf{if}\ (e)\ S}{\textbf{if}\ (\ominus e)\ S \quad \sharp e} \quad S\notin\mathbb{L}$$

$$\textsc{if}\mathrm{L}\ \frac{\textbf{if}\ (e)\ S}{\begin{array}{ll}\textbf{if}\ (tag(\ominus e))\ \ominus S & \sharp e\ ; \\ & \textbf{if}\ (pop(l_t,l_f))\ \sharp S\end{array}} \quad S\in\mathbb{L}$$

$$\textsc{assign}\ \frac{e_1 = e_2}{\ominus e_1 = \ominus e_2 \quad \sharp e_1 ; \sharp e_2} \qquad \textsc{seq}\ \frac{S\,;T}{\ominus S\,;\ominus T \quad \sharp S\,;\sharp T}$$

$$\textsc{conj}\mathrm{N}\ \frac{e_1\ \&\&\ e_2}{\ominus e_1\ \&\&\ e_2 \quad \sharp e_1} \quad e_2\notin\mathbb{L}$$

$$\textsc{conj}\mathrm{L}\ \frac{e_1\ \&\&\ e_2}{\begin{array}{ll}\ominus e_1\ \&\&\ \ominus e_2 & \sharp e_1\ ; \\ & \textbf{if}\ (pop(l_1,l_2))\ \sharp e_2\end{array}} \quad e_2\in\mathbb{L}$$

Figure 4. Transformation rules for standard constructs.

Rules ifL and ifN deal with the **if**(e) S structure (if-then). If S is log-relevant, then the then-branch is an attractor (of a logging node), so we have to tag the corresponding decision point (ifL). Else both branches of the if-then are neither an attractor nor distractor, and thus we should not tag the decision point (ifN). Note that the transformation is also recursively applied to e, since it may call a log-relevant method.

The rules for assignment (Assign) and sequential composition (seq) are straightforward. Unary expressions $op\ e$, and binary expressions $e_1\ rop\ e_2$ where rop is an arithmetic or relational operator do not induce implicit control branching. Therefore, their rules are similar to assignments. Logical operators are more involved. The semantics of $e_1\ \&\&\ e_2$ in Java specifies that e_2 is not evaluated if e_1 turns out to be false (*short-circuiting*). This matters for the control-flow, in particular when e_2 is log-relevant. The rules conjN and conjL deal with this. Other operators with implicit branching, such as `||`, and `?:` can be treated analogously.

A. Method call and polymorphism

If a statement S calls a log-relevant method m, its decoder $\sharp S$ should also call the decoder of m. The rules to handle method calls are shown in (3) and (4).

$$\textsc{call}\mathrm{L}\ \frac{\textbf{call}\ m(e_1 ... e_k)}{\begin{array}{ll}\textbf{call}\ m(\ominus e_1 ... \ominus e_k) & \sharp e_1 ; ... ; \sharp e_k ; \\ & \textbf{call}\ dec\_mD()\end{array}} \quad m\in\mathbb{L} \tag{3}$$

$$\textsc{call}\mathrm{N}\ \frac{\textbf{call}\ m(e_1 ... e_k)}{\textbf{call}\ m(\ominus e_1 ... \ominus e_k) \quad \sharp e_1 ; ... ; \sharp e_k} \quad m\notin\mathbb{L} \tag{4}$$

If the method has a receiver, we treat it as its first argument (so, we write $m(x,y)$ instead of $x.m(y)$).

However, to also handle polymorphism, these rules have to be extended. We will explain this with an example; consider the following program:

```
Person p = getPerson() ;
p.work() ;
```

Suppose *Person* has K number of subclasses that override *work*(). The decoder of the above statement will now have to figure out which variant of *dec_work* it has to call. The obvious case is when neither *Person*'s *work*() nor its variants are log-relevant. Then the call *p.work*() is *not* log-relevant, and we can use callN to handle it.

In other cases, the decision cannot in general be made statically. To handle this, we assign *Person*'s *work*() and its variants a unique variant-number *id* in the range $[0...K]$. The call *p.work*() is instrumented such that it checks at the run time which variant is called, and it writes *bits*(*id*) to the trace, where *id* the variant number and *bits*(*id*) returns its unsigned bits representation.

For example, suppose only *Student* overrides *Person*'s *work*(), and at least one of them is log-relevant. The decoder of the above statement first reads the encoded variant number and then, it interprets it to call the correct variant:

```
code = bits2num(pop(),pop()) ;
switch(code) {
  case 0 : Person.dec_work() ; break;
  case 1 : Student.dec_work(); break;
```

B. Jump-over

```
m() { if(a) x++ ;
      if(ok) return ;
      log("not ok") }
```

Figure 5. A method with a jump-over.

Consider the example in Figure 5. Since the then-parts of both conditionals are not log-relevant, the rule ifN will not tag the corresponding decision nodes, resulting in these tagged version and decoder:

```
m() { if(a) x++ ;
      if(ok) return ; }
```

```
dec_m() { emit(3,"not ok") ; }
```

This decoder is however incorrect, because it always produces "not ok", whereas the original m may not do so.

The problem arises from the rule for transforming sequential composition (SEQ): it assumes normal control flow. That is, any execution of $S;T$ always executes the T-part. Under this assumption, no edge in S can become an attractor or distractor of a logging-edge in T. Consequently, when transforming S we do not need to care about the log-relevance of T. However, statements like **return**, **break**, and **continue** (we call them *jump-over* statements) break this assumption. Note that not all jump-overs in S can actually be distractors of a logging-edge in T, and calculating which of them are, introduces some overhead. We will instead redefine $S \in \mathbb{L}$ (Definition 3):

Definition 5: A statement S is *directly log-relevant*, denoted by $S \in \mathbb{L}_0$, if it contains a call to *log*(..), or a call to a log-relevant method (as in the old definition of "log-relevant" statement). □

Definition 6: A statement S is *log-relevant*, denoted by $S \in \mathbb{L}$, if either (1) $S \in \mathbb{L}_0$, or (2) it is a part of a log-relevant method *and* one of these holds:

2a) S contains a **return**.
2b) S is a part of a directly log-relevant **switch** and S contains a **break**.
2c) S is a part of a directly log-relevant loop and S contains a **break** or a **continue**.

□

With this new definition, the previous example will give the following tagged version and correct decoder:

```
m() { if(a) x++ ;
      if(tag(ok)) return ; }

dec_m() { if(pop(2,3) return ;
          emit(3,"not ok") ; }
```

Consider $S;T$ in a log-relevant method m. The new definition of log-relevance presumes that if an edge e in S is an attractor to a **return** node in S, it will also be a distractor of a logging edge in T. This is only true if T has a logging edge. We have a similar situation in **while**$(g)\{S;T\}$ if S contains a **break** or **continue**. This means that if T actually has no logging edge, we will end up logging some edges that are actually not log-relevant. But at least we do not miss one (so, the change above is safe).

Some rules need to be added as well; jump-overs change the flow of control, so they need to be copied to the decoder. These are in (5); bc is either **break** or **continue**.

$$\text{BRC}\ \frac{bc}{bc \quad bc} \qquad \text{RET}\ \frac{\textbf{return}\ e}{\textbf{return}\ \ominus e \quad \sharp e;\ \textbf{return}} \tag{5}$$

C. Switch statement

An **if** statement has two branches. So, when tagging it, one bit suffices to encode the choice between them. A **switch** statement may have N branches, with $N \geq 2$. This can be dealt with as follows. We overload the function *tag*(e) to also take a bitstring as argument. It returns nothing, and writes the bitstring to the trace. We tag the k-th branch by *tag*(*bits*(k)) where *bits*(k) is the bitstring (of length $n = {}^2log(N)$) that encodes the number k.

The absence of **break** in a switch branch requires a special treatment though. The control flow would implicitly "fall through" to the next branch. Consider this example:

```
switch (day) {
  case 6:   log("sat")
  case 7:   log("weekend") ; break ;
  default:  log("work")    ; break ;  }
```

If we just treat **switch** analogously to **if**, we would get the tagged version and decoder shown in Figures 6 and 7.

```
switch (day) {
  case 6:   tag(FF);
  case 7:   tag(FT); break ;
  default:  tag(TF); break ;  }
```

Figure 6. An incorrectly tagged fall-through switch.

```
int code = bits2num(pop(),pop()) ;
switch (code) {
  case 0: emit(2,"sat")       ;
  case 1: emit(3,"weekend") ; break ;
  case 2: emit(4,"work")     ; break ;  }
```

Figure 7. The decoder for a fall-through switch.

The decoder correctly consumes two bits. However, if *day* = 6 the tagged version will fall through and incorrectly produces four bits. The issue can be solved in two ways. One solution involves remembering the number of times a **switch** execution falls through; e.g., x times. The decoder must then consume $n{+}x{*}n$ bits, and discard the last $x{*}n$ bits. However, this involves quite a bit of bookkeeping. A much simpler solution, the one that we follow, is to simply duplicate the **switch**, in which the first of the duplicates executes a single call to *tag* for each case, and the second implements the original logic (but without the tags). This results in the tagged version in Figure 8, to be used with the decoder in Figure 7.

```
switch (day) {
  case 6:   tag(FF); break ;
  case 7:   tag(FT); break ;
  default:  tag(TF); break ; }

switch (day) {
  case 6:   ;
  case 7:   break ;
  default:  break ; }
```

Figure 8. Correctly tagged fall-through switch.

D. Loops

Consider first the following straight forward proposal to transform a **while**-loop:

$$\frac{\textbf{while}\ (e)\ S}{\textbf{while}\ (tag(\ominus e))\ \ominus S \quad \sharp e\ ;\ \textbf{while}\ (pop(l_t, l_f))\ \{\ \sharp S;\ \sharp e\ \}} \qquad e \in \mathbb{L} \lor S \in \mathbb{L}$$

We can see it as the loop-version of the IFL rule. The reader may notice that unlike IFL now we also apply the transformation when only the guard is log-relevant. This is correct: the branch that goes into the loop's body would still be an attractor of the edge that goes from the body's end back to the guard. In contrast, in an **if**(e) S, the then-branch cannot be an attractor if S is not log-relevant.

The above rule is however incorrect if the loop contains a jump-over. Let us consider the example below; suppose $decide \in \mathbb{L}$.

```
while(decide(i)){ log("hi") ; continue ; }
```

The resulting tagged version and decoder:

```
while(tag(decide(i))) { continue ; }
```

```
dec_decide() ;
while(pop()){
  emit(1,"hi") ; continue ;
  dec_decide() ;
}
```

The decoder incorrectly produces the logs from *decide* exactly once, whereas the original loop can do this multiple times. The correct rule is shown in (6).

$$\textsc{loopL}\ \frac{\textbf{while}\ (e)\ S}{\begin{array}{ll}\textbf{while}\ (tag(\ominus e)) & \textbf{while}\ (true)\ \{\\ \quad \ominus S & \quad \sharp e\ ;\\ & \quad \textbf{if}\ (pop(l_t,l_f))\\ & \qquad \textbf{break}\ ;\\ & \quad \sharp S\ \}\end{array}}\quad \begin{array}{l} e\in\mathbb{L}\ \vee\\ S\in\mathbb{L}\end{array} \tag{6}$$

If the loop is not log-relevant, it is removed from the decoder, as shown in (7).

$$\textsc{loopN}\ \frac{\textbf{while}\ (e)\ S}{\textbf{while}\ (e)\ S \quad \epsilon}\quad e\notin\mathbb{L}\wedge S\notin\mathbb{L} \tag{7}$$

The transformations for do-while and for-loops are a bit more elaborate, although they follow the same general idea [9]. Note however, that simply treating **do** S **while**(e) as S; **while**(e) S does not work if S contains a jump-over.

E. Loop compaction

The above tagging scheme for loops is still problematical. Consider this example:

```
i=0 ;
while(i<n)
  if (i==999) log("special") ; i++ ;
}
```

Recall that we have required to also log distractors of log-relevant node (Def. 1). For the above loop, this means that every iteration always logs at least one bit, despite the fact that most, if not all, of its iterations will not actually produce any call to *log*(...). This can spam a very long bitstring, to eventually produce at most one static string in the enhanced log. To avoid this, we choose to discard the trace produced by an iteration if it does not actually pass any logging node. This is done by the following *loop compaction* algorithm.

There is a global stack *Lstack*. Elements of this stacks are pairs (i,z) where i is a so-called loop-ID, and z is a buffer where we can temporarily save a trace-fragment.

1) Every log-relevant loop H is instrumented such that when the loop is entered, a unique loop-ID *lid* is created. The id is created fresh every time the loop is entered, to distinguish between different invocations of the loop.
2) Whenever the guard g of H is evaluated, and it evaluates to true (so, a new iteration is about to start), a new buffer z is created, and the pair (lid,z) is pushed into *Lstack*. We maintain the following invariant:

   The traces buffered in Lstack never pass a logging node.

   So, when the execution of H cycles back to its guard, we can remove (lid,z) and all pairs above it from *Lstack*.
3) $tag(e)$ will write to the buffer at the top of *Lstack*, unless this stack is empty. Then it will write the bit directly into the trace file.
4) a call to $log(v)$ breaks the above invariant. So, we dump all buffers in *Lstack* (from bottom to top) to the trace file. And then clear the whole stack. From this point on *tag* and *log* will write directly to the trace file, until an iteration push a new (lid,z) into *Lstack*.
5) Because a loop may terminate through a jump-over, there may be some residual bits left in *Lstack*. The next logging node will cause these residual bits to be dumped into the trace file.

Using this scheme, the previous loop will produce a trace where it appears as if the loop immediately does the 999-th iteration, and then it exits.

F. Recursion

Recursions can cause a similar bits-spamming problem as loops. The above loop compaction algorithm exploits the property that every iteration of a loop returns to the loop's guard. On the other hand, a recursive function that calls itself multiple times (such as the example below) does not have a natural analogous of the 'loop-guard' concept. So, the above compaction scheme cannot be re-applied for recursions.

What we do is to wrap each recursive call with a dummy single iteration, e.g:

```
fib(n) {
  if (n==0) { return 0 ; }
  if (n==1) { return 1 ; }
  if (n>=10) { log("This may be too big") ; }
  int a,b ;
  int i=0 ; while(i<1) { a = fib(n-2) ; i++ ; }
      i=0 ; while(i<1) { b = fib(n-1) ; i++ ; }
  return a+b ;
}
```

This has the effect that if a recursive call to fib does not pass a logging node, it will not produce any tracing information either. Else, the execution from the first call to fib up to the logging node will be traced.

V. Exceptions

In most programming languages, many types of expressions can potentially throw an exception. In terms of control flow, such an expression introduces implicit branching, with one *implicit normal branch* that corresponds to the instruction's normal execution, and one or more *implicit* *exceptional branches* that correspond to jumps to exception handlers. A typical program contains a lot of such implicit edges. Consider for example:

```
x++ ; a.x = x ; ... ; log("here") ;
```

The two assignments before `log` can throw an exception. The corresponding implicit exceptional branches are distractors of the *log*(..) statement, the assignments themselves become attractors. According to Definition 1, we have to log them as well. This would mean that a normal execution (one that does not throw any exception) that leads to a *log*(..) statement would generate additional tracing information belonging to *all* implicit distractors it passes; and there are many of them. This is too verbose! When an execution throws an exception, we are indeed usually interested in the corresponding tracing information. But when it does *not* throw any exception, we are much less interested in knowing which exceptions and which handlers it thus by-passed.

So, implicit normal branches are not going to be logged. We would want to log log-relevant implicit exceptional branches; but this is problematical for a different reason. To log these edges would mean that we have to instrument all subexpressions in P that can potentially throw an exception. The overhead would be unacceptable. To make it practical, we decide to only log the destination nodes; thus, the exception handlers. Thus, our logs can reveal which exceptions have been thrown, but not the specific subexpression that threw them.

```
log("Preparing") ;
try {
  prepare() ;
  try { x = receive() ; log("Received") ; }
  catch (ExcA a) { log("Ouch") ; }
  catch (ExcB b) { x=-1 ; }
}
catch (Exception e) { }
log("Done") ;
```

Figure 9. A try-catch statement.

Consider the example in Figure 9; assume that `prepare` and `receive` are not log-relevant. An exception handler h is logged if there is a log-relevant implicit exceptional edge e that goes to it (this can only be the case if e is a distractor or attractor of another log-relevant edge). This is the case for all handlers above. In general, in **try** t **catch** h_1 **catch** h_2..., if t or one of the h_k is log-relevant, then *all* handlers in the construct will be either an attractor or distractor, and thus have to be logged.

Now, consider first the following proposal of a tagged version of the statement in Figure 9; *logE*(e) is used to log a thrown exception:

```
try {
  prepare() ;
  try { x = receive() }
  catch(ExcA a) {logE(a); }
  catch(ExcB b) {logE(b); x=-1 ; }
}
catch (Exception e) { logE(e) ; }
```

Suppose the execution throws an *ExcA*. Indeed, this will be logged. The idea is to extend the decoder so that upon reading the logged exception it will *replay* it, and thus duplicate the original flow of control. However, just from the logged exception the decoder will not be able to infer whether the exception was thrown before the second **try** or in the second **try**, which is important to decide to which handler the control should flow.

To determine the right moment to replay the exception, we will do progress counting. A global variable τ : *int* is introduced for this purpose. The method *logE*(e) will now additionally log the value of τ; so in principle, now the decoder has the information to decide when it is the right moment to replay an exception.

The method *tick*() will be used to increase τ by one. We only need to *tick* when P passes points that matter for logging:

1) To distinguish if a logged exception is thrown before or inside a (log-relevant) **try-catch** structure, we tick just before we enter the structure.
2) To distinguish if a logged exception is thrown inside or after a **try/catch/finally**-section of a (log-relevant) **try-catch** structure, we tick when we reach the section's end.
3) For a similar reason, *tag*/*pop*, and *log*/*emit* implicitly call *tick*().

This results in the tagged version in shown Figure 10.

```
tick() ;
try {
  prepare();
  tick();
  try { x = receive()  ; tick(); }
  catch(ExcA a){logE(a); tick(); }
  catch(ExcB b){logE(b); x=-1; tick(); }
  tick(); }
catch (Exception e) { logE(e); tick() ; }
```

Figure 10. Correctly tagged try-catch statement.

```
tick() { check() ; tau++ ; }
```

Figure 11. Decoder's *tick*() also checks exception maturity

The counting of the progress should also be reflected in the decoder. That is, whenever the tagged version calls *tick*(), a *tick*() should be added at the corresponding place in the decoder. Furthermore, *pop* and *emit* implicitly calls *tick*() to match the same calls in *tag* and *log*. The decoder's version of *tick*() is slightly different, as shown in Figure 11. Before it increases its progress counter (τ), it checks whether the current item in the trace is a pair (e, t), representing an exception thrown at time t. If t is equal to the the current value of τ, we say that the exception *e has matured*. The decoder should then consume (e, t) from the trace and replay

e by throwing it. Else, (e, t) is not consumed an the decoder simply proceeds to its next statement.

The resulting decoder is shown in Figure 12.

```
emit(1,"Preparing") ;
tick();
try {
   tick() ;
   try{ emit(4,"Received"); tick(); }
   catch(ExcA a) { emit(5,"Ouch"); tick(); }
   catch(ExcB b) { tick(); }
   tick() ; }
emit(9,"Done") ;
catch(Exception e) { tick() ; }
```

Figure 12. Correct decoder for the try-catch statement.

The full transformation rule is shown in (8). It is for the case when the try-block is log-relevant. Else it will not be transformed.

$$\text{TRYCATCH}\ \frac{\textbf{try}\ S\ \textbf{catch}(e)\ T\ \textbf{finally}\ U}{\begin{array}{ll} \textit{tick}(); & \textit{tick}()\ ; \\ \textbf{try}\ \{\ \ominus S\ ;\ \textit{tick}()\ \} & \textbf{try}\ \{ \\ \textbf{catch}(e)\ \{ & \quad \sharp S\ ;\ \textit{tick}()\ \} \\ \quad \textit{logE}(e)\ ; & \textbf{catch}(e)\ \{ \\ \quad \ominus T\ ;\ \textit{tick}()\ \} & \quad \sharp T\ ;\ \textit{tick}()\ \} \\ \textbf{finally}\ \{\ \ominus U\ ;\ \textit{tick}()\ \} & \textbf{finally}\ \{ \\ & \quad \sharp U\ ;\ \textit{tick}()\ \} \end{array}} \quad \begin{array}{l} S\in\mathbb{L}\ \vee \\ T\in\mathbb{L}\ \vee \\ U\in\mathbb{L} \end{array} \tag{8}$$

Additional handlers are transformed in the same way. However, when only the **finally**-part is log-relevant, we do not actually need to log the handlers (to add *logE* there).

We also have to deal with uncaught exceptions. Such an exception will escape all handlers in the program (and then causes the program to crash). Such an exception is almost always log-relevant, so we need to log it as well, so that the decoder knows when to stop its current execution. To do so the body of the top-level entry point method (e.g., `main`) need to be wrapped by a fake **catch**-clause that catches any exception and rethrows it; the transformed version will then add the needed call to *logE*.

A. Untraced Call-backs

Most programs use standard libraries and other external libraries. When P calls an external method, this method may in turn call back to some method m in P. The latter may perform logging. During decoding, we have to ensure that *dec_m* is called. Normally, this is the responsibility of m's caller's decoder, ensuring that the original execution is faithfully imitated. The problem is that external libraries can not be assumed to have been exposed to our transformation; therefore, it has no decoder and thus, nobody will call *dec_m*. We call such a call back (to m) an *untraced call*.

It turns out that the solution we had to deal with exceptions (Section V) can be reused. Let us suppose it is possible to intercept untraced calls to m at the runtime. When such a call comes, we treat it as if an $Untraced_m$ exception has occurred, and log it (along with the current value of the progress counter). The name of the called method (m) is encoded in the name of the exception. The decoder will then be able to infer when this happened, and furthermore, it knows where to jump to proceed.

To be able to intercept untraced calls, we rename all log-relevant methods with fresh names; e.g., $m(x)$ to mz. In the target application, all calls to m are accordingly modified to calls to mz. Then we re-introduce the method $m(x)$ with the same signature, defined as in Figure 13.

```
m(x) {
   logE(new Untraced_m()) ;
   mz(x)  ; // call the original m
}
```

Figure 13. Wrapper to intercept untraced calls to m.

External methods that call to the original m will still call it with its old name, and thus will call the new m above. So, there we can code the logic of the interception, as shown above.

VI. Proof of Concept

As a proof of concept we built a prototype implementing the transformation discussed before. All Java control structures, except *labelled* break and continue, are handled We use Eclipse Java Development Tools (JDT) that allows Java source code to be analyzed and transformed at the abstract syntax tree (AST) level.

So far we assumed that the program P only has a single top-level entry point, e.g., its `main` method. This does not work for logging, e.g., a GUI application. Such an application is event driven: when a user interacts with it, e.g., by clicking on a button, it generates an 'event', and the corresponding event handler is executed. Each handler acts thus as a top-level entry point. In our implementation, it is possible to annotate multiple methods as top-level. A 'full execution' means an execution of a top-level method m, from its start to its end. Each full execution generates a separate trace file, e.g., named *log_m_timestamp.txt*. From such a name we can infer back to which decoder the file must be given. The trace is actually split into two log files: a *blog_m.txt* file containing the pure bitstring part of the trace, and a *evlog_m.txt* file containing dynamic values and thrown exceptions. This allows the bitstring to be stored more compactly.

To validate that our approach works, we try it on a number of examples, listed in Table I. LOCS is the total lines of code (comments and white lines are not counted). *TrT test suite* is a set of classes consisting of various logged statements that we use to test our implementation. *Reversi* is a small a GUI-based program implementing a game of the same name. This program has multiple top-level entry points. *Nanoxml* is an open source small XML parser. *Barred* is an open source file archiver. Except for the Trt test suite, these programs do not actually do any logging. We artificially add logging statements, by converting most comments to calls to $log(s)$ or $log(s,v)$.

Table I Some Statistics

	#class	*#methods*	LOCS
TrT test suite	33	59	629
Reversi	4	30	473
Nanoxml	25	285	3321
Barred	21	45	2075

We then run the examples on several sample inputs and compare the resulting logs and the logs that we would get if we do not apply our transformation. The results are shown in the table below.

Table II Results

	org (KB)	*tr*	*enh*	*DVR*	*CR*	*ER*
TrT test suite	3.3	1.3	24.1	0.27	0.39	7.3
Reversi	29.6	7,5	87.6	0.09	0.16	3.0
Nanoxml	268	101	3054	0.18	0.38	11.4
Barred	29.2	1.4	111	0	0.05	3.8

Above, *tr* and *ehc* express the size, in KB, of the trace file and enhanced log. It is calculated by counting the number of characters; every character is counted as two bytes. The *org* is the size of the original logging fragments in *ehc*. $CR = tr/org$ is the obtained compaction ratio, and $ER = enh/org$ is an indication of the enhancement factor. So, the compaction factor of 0.38 for Nanoxml means that our generated trace is 0.38× smaller than what we would get from normal logging, whereas its enhancement factor of 11.4 means that after decoding we enrich the normal log with tracing information, roughly by 11.4×. The above way to calculate *ER* is indeed debatable. One may point out that we should instead compare the amount of raw information the logs carry. But this is also not very useful: the trace file can be thought as a minimal representation the raw information that the corresponding enhanced log embodies; but not even a tool can read a trace file without decoding it first. Enhanced logs produced by our implementation are intended to readable by human and parsable by tools. So they do contain some verbosity, but we did try to minimize it (e.g., we did not blow them up to HTML); so, comparing them to the size of the original logs seems reasonable. *DVR* is the ratio of the amount of logged dynamic values in the normal log. The above results indicate that higher *DVR* will decrease the compaction ratio, which is to be expected since, unlike static strings, dynamic values have to actually be logged.

We expected that the run time overhead would be less compared to normal logging, because we would have to do less I/O. However, in our experiments this does not turn out to be the case, as shown in the table below for the Nanoxml and Barred examples. *#calls* is the total number of calls to the *log* function, and *OV* is the resulting total time overhead in ms; *ovc* is the average overhead per number of call, and *ovs* is the average overhead per KB of logged data in the original log. These numbers indeed suggest that the overhead is quite small, but ideally those numbers should be negative. We believe that the implementation can still be improved by choosing more clever data structures in our implementation.

Table III Time overhead

	#calls	*OV* (ms)	*ovc*	*ovs*
Nanoxml	5335	320	0.06	1.19
Barred	660	277	0.42	9.55

With respect to the conclusions suggested by the above results, the following are the threats to their validity.

1) Logging statements were artificially added; as said, by converting comments to calls to *log*. This resulted in quite intensive logging. A program with real logging may log with different intensity. In particular, when a program logs less frequently, it can be expected to also have more attractors/distractors between logging nodes. This affects *CR* negatively, but improves *ER*. *DVR* also matters; higher *DVR* affects *CR* negatively, without improving *ER*.
2) The amount of logging investigated was at most in the order of hundreds of KBs. In particular, we did not investigate large scale logging (e.g., in the order of GBs).

VII. Related Work

Most work in logging has been focused on providing logging infrastructure for various software technologies. Many modern programming languages already come with logging libraries. These provide functions we can use to log messages. They often have a notion of 'logging level' to control the verbosity of the generated logs. There are also alternative libraries such as the Log4x family [10] that provide, e.g., improved APIs or improved performance. Apache Commons Logging offers a set of common logging APIs so that the implementation of the logger can be decoupled and replaced easily. Some SDKs, such as the Google Web Toolkit (GWT) for developing web applications may also come with its own logging library, specialized to the kinds of applications that they target. Most logs, e.g., web servers logs or OS logs, are semi structured [11], where for example types of events and their time stamps can be distinguished, but further information about them are often described in free style strings. Obviously, the more refined the structure is, the more viable they are for analysis. Some debuggers produce deeply structured logs [12]. FITTEST testing framework comes with PHP and Flash loggers that produce deeply structured logs [13], which are used to feed its model and oracle inference tools [8], [14].

Logging statements can be manually added into a program, or automatically inserted through program transformation. For example, this can be done by specifying the logging as a separate 'aspect', which is then weaved into the target program using an AOP tool like AspectJ [15], or using a special log instrumentation tool such as ABCi for Flash [16].

Or, we can use a generic program transformation tool such as Stratego [17]. Our approach can be seen as adding another layer of transformation. Our implementation is ad hoc, using JDT. In retrospect, using a tool like Stratego might have been a better choice, as it allows the transformation to be specified and composed more abstractly.

VIII. Conclusion & Future Work

We have presented a new log encoding scheme. The approach works by transforming the source code of the target program, to make its control flow statements to log bitstrings encoding their flows of control. The produced logs are significantly more compact than traditional logging, while at the same time, when decoded they enhances the resulting normal logs with substantial tracing information. To reconstruct the logs, decoders are needed; they are produced by the same transformation above.

A prototype implementing the approach has been built and tested on some real life programs. The results show 0.05 - 0.4 compaction ratio (2.5 - 20 times more compact), and 3 - 11 enhancement factor.

Future work. We want to investigate if the tracing information can be further enhanced, e.g., by logging runtime types of relevant objects. In theory, if such information is enumerable, the enumeration allows them to be compactly encoded, and thus the logging overhead is minimum.

We want to investigate if the logging scheme can be extended to multi-threaded programs. We believe that our scheme can be straightforwardly tweaked such that each thread writes to its own trace file (which is also good to maintain concurrency). However, when interpreting the resulting logs, we still want to infer what the temporal relations are between entries in the logs of different threads (e.g., does this entry e_1 from the thread T_1 occurs before e_2 from T_2?). Time stamping every bit in the trace is obvious not acceptable. However, we can log the time whenever T_1 manages to obtain a lock c, and when it releases it again. Because two threads cannot at the same time obtain the same lock, this will at least allow us to infer the happen-before relation between the threads.

We want to investigate if the logging scheme can be made more flexible by being able to *dynamically* turn on and off its tracing mode. Currently, it *always* produces tracing information, whether we want it or not. One situation where it would be useful to turn off tracing is when a loop/recursion spam too much tracing bits, despite the compaction scheme that we have applied. Turning tracing off is actually quite easy. However, the decoders relies on the tracing information to be able to correctly do their work. So, when a fragment of the trace is suppressed, the decoders have to be made smarter so that they can fill in the missing fragment on their own, so that they at least are able to continue decoding the rest of the trace.

Acknowledgment. This work is funded by the EU FITTEST project No. 257574.

References

[1] J. H. Andrews, “Testing using log file analysis: tools, methods, and issues,” in Procd. 13th IEEE Int. Conf. on Automated Software Engineering, 1998, pp. 157–166.

[2] J. H. Andrews and Y. Zhang, “Broad-spectrum studies of log file analysis,” in Procd. 22nd Int. Conf. on Software Engineering, 2000, pp. 105–114.

[3] H. Barringer, A. Groce, K. Havelund, and M. Smith, “Formal analysis of log files,” AIAA Journal of Aerospace Computing, Information and Communications, 2010, pp. 365–390.

[4] S. Pachidi, “Software operation data mining: techniques to analyse how software operates in the field,” Master’s thesis, Dept. Inf. & Comp. Sciences, Utrecht Univ., 2011, IKU-3317153.

[5] K. Kowalski and M. Beheshti, “Improving Security Through Analysis of Log Files Intersections,” I. J. Network Security, vol. 7, no. 1, 2008, pp. 24–30.

[6] D. Lorenzoli, L. Mariani, and M. Pezzè, “Automatic generation of software behavioral models,” in Procd. of the 30th int. conf. on Software engineering. ACM, 2008, pp. 501–510.

[7] D. Tu, R. Chen, Z. Du, and Y. Liu, “A Method of Log File Analysis for Test Oracle,” in Scalable Computing and Communications; Eighth International Conference on Embedded Computing, 2009. SCALCOM-EMBEDDEDCOM’09. International Conference on. IEEE, 2009, pp. 351–354.

[8] L. Mariani, A. Marchetto, C. D. Nguyen, P. Tonella, and A. I. Baars, “Revolution: Automatic evolution of mined specifications,” in ISSRE, 2012, pp. 241–250.

[9] A. Sturala, “Record-based Logging,” Master’s thesis, Dept. Inf. & Comp. Sciences, Utrecht Univ., 2011, ICA-3324192.

[10] C. Gulcu, “Log4j delivers control over logging,” Java World, 2000.

[11] A. Schuster, “Introducing the Microsoft Vista event log file format,” digital investigation, vol. 4, 2007, pp. 65–72.

[12] M. Auguston, “A Program Behavior Model Based on Event Grammar and its Application for Debugging Automation,” in AADEBUG, 2nd Int. Workshop on Automated and Algorithmic Debugging, 1995, pp. 277–291.

[13] I. S. W. B. Prasetya, A. Elyasov, A. Middelkoop, and J. Hage, “FITTEST log format (version 1.1),” Dept. of Inf. and Comp. Sciences, Utrecht Univ., Tech. Rep. UU-CS-2012-014, 2012.

[14] I. S. W. B. Prasetya, J. Hage, and A. Elyasov, “Using subcases to improve log-based oracles inference,” Dept. of Inf. and Comp. Sciences, Utrecht University, Tech. Rep. UU-CS-2012-012, 2012.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An Overview of AspectJ,” in ECOOP’01, 2001, pp. 327–353.

[16] A. Middelkoop, A. Elyasov, and I. S. W. B. Prasetya, “Functional instrumentation of ActionScriptPrograms with ASIL,” in Implementation and Application of Functional Languages, ser. LNCS, vol. 7257, 2011, pp. 1–16.

[17] E. Visser, “Stratego: A language for program transformation based on rewriting strategies,” in Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings, ser. LNCS, vol. 2051, 2001, pp. 357–362.