A Method for Scalable and Precise Bug Finding Using Program Analysis and Model Checking

Manuel Valdiviezo, Cristina Cifuentes, and Padmanabhan Krishnan

Oracle Labs
Brisbane Australia
{manuel.valdiviezo,cristina.cifuentes,paddy.krishnan}@oracle.com

Abstract. This paper presents a technique for defect detection in large code bases called model-based analysis. It incorporates ideas and techniques from program analysis and model checking. Model checking, while very precise, is unable to handle large code bases that are in the millions of lines of code. Thus we create a number of abstract programs from the large code base which can all be model checked. In order to create these abstract programs, we first identify potential defects quickly via static analysis. Second we create a program slice containing one potential defect. Each slice is then abstracted using a combination of automatic data and predicate abstraction. This abstracted model is then model checked to verify the existence or absence of the defect. By applying model checking to a large number of small models instead of one single large model makes our approach scalable without compromising on precision.

We have applied our analysis to detect memory leaks and implemented it using aspects of the Parfait static code analysis tool and the SPIN model checker. Results show that our approach scales to large code bases and has good precision: the analysis runs over 1 million lines of non-commented C++ OpenJDK™ source code in 1 hour and 19 minutes, with a precision of 84.5%. Further, our analysis found 62.2% more defects when compared to the dataflow approach used by Oracle Parfait's memory leak checker.

1 Introduction

In this paper we present a technique that combines abstraction and software model checking (SMC), which enables us to detect defects in large code bases. The motivation for this research is the need to develop automated defect finding techniques that are more accurate than purely static analysers and can be made to scale systems consisting of 1 million lines of uncommented code. The technique must also be able to report the results in a few hours on standard desktop machines. To be realistic, we do not demand completeness; thus the technique might miss a few defects. Hence we do *not* aim to verify the original program. But we require high precision (viz., a low false positive rate) as demanded by the consumers of our results. Our aim is to have a precision of more than 80%.

As we are looking for automated techniques, model checking is a potential starting point. Software model checking (SMC) technology is suitable for the

J. Garrigue (Ed.): APLAS 2014, LNCS 8858, pp. 196–215, 2014.
© Springer International Publishing Switzerland 2014

verification of small/medium code bases, up to the low thousands of lines of code. However, it cannot handle large code bases that have millions of lines of code [1].

The TACAS 2013 and 2014 competitions on software verification (`http://sv-comp.sosy-lab.org/2013/results/` and `http://sv-comp.sosy-lab.org/2014/results/`) identify model-checkers that perform well on various benchmarks. All the benchmarks used in the competition are relatively small when compared with our needs. We were unable to use tools identified by them (such as LLBMC [2] or CBMC [3]) on our real code bases which have more than one million lines of uncommented code.

Abstraction [4,1] and bounded model checking [3] are two of the possible techniques to get a handle on such large code bases. In this paper we describe and demonstrate an effective abstraction (also called model generation) technique that can be combined with SMC. Our data and predicate abstraction is totally automatic unlike Bandera [4] which requires manual processing which is just not feasible on our large code bases. The main reason for the efficacy of our approach is the generation of multiple models for a given property. We ensure that each model has only one potential defect. Thus each model will be small enough to be verified using model checking very quickly. This is based on the observation that model checking works very well on small program and our aim is to run many invocations of the model checker on small models. To achieve this we use a defect-driven slicing and abstraction process.

The key steps in our approach are as follows.

1. Given a desired property, we identify all statements where a defect *could* occur. These statements form the list of potential defects.
2. For each potential defect we create a slice of the program that has only the relevant variables and conditions we want to check for.
3. Each slice is converted into a *specialised abstraction* using automatic data abstraction (i.e., discarding irrelevant values, or converting a range of values to a single value related to the property being checked for). Where automatic data abstraction is not possible, a suitable predicate abstraction (i.e., replacing predicates with boolean variables) is used. By using predicate abstraction only in limited contexts, we reduce the cost of predicate solving. This results in small models that are constructed quickly.
4. The resulting models are then verified against the desired property using a model checker.

The novel aspects in our approach include the use of automatic data and predicate abstraction to generate a number of, potentially small, models that can be model checked, and at the same time keeping sufficient information in the model so as to not require refinement based on any counter-example after the model-checking process.

While our approach is general, we use memory leaks as an example to demonstrate the generation of the set of abstractions. In order to handle other defect types, one has to specify a customised abstraction algorithm. This customisation can be based on our technique of using data and predicate abstraction.

The rest of the paper is organised as follows. In Section 2, we survey some related work. In Section 3, we present an example that illustrates our approach. In Section 4, we explain the technical details of our approach while in Section 5, we outline our implementation. In Section 6, we present our experimental results and conclude in Section 7.

2 Related Work

From a performance view point static analysers can be very effective at detecting defects; they often trade speed for accuracy. However, it is often the case that complex analyses are not scalable. ESP [5] represents a general techniques that could be applied to the detection of memory leaks. It uses property simulation to prune the number of paths explored by the analysis. It relies on encoding of temporal safety properties and, in principle, can be used to detect memory leaks. However, the results reported [6] appear to indicate that the approach is very sensitive to the input program. In the context of memory leaks, Sparrow [7] uses interprocedural but non-path-sensitive analysis. Sparrow took about 2 hours to process `binutils-2.13.1`, a small to medium sized program. Saber [8] uses sparse value-flow graph to represent def-use chains and value flows via assignments. Leaks are detected by performing a reachability analysis on this graph. The authors state that Saber is faster than Sparrow and also works on large systems such as `wine-0.9.24`. Unfortunately, these tools are not available and we have been unable to use them in our experimentation.

The idea of using slicing to reduce the complexity of analysis to speed up the verification process has been explored in recent times [9]. The scalability of such techniques is very much an open question, especially as they slice models which by definition are compact. Similarly [10] attempt to verify aspects of operating systems after code slicing. But they admit that they can use model checking only within a limited scope.

It is also possible to use SMT solvers on the slicing to remove false alarms (i.e., verify that the defect is not possible) [11]. However, the results provided by the authors indicate that SMT solvers are unlikely to scale. None of the programs considered are really large. In some cases the SMT solver did not terminate and in other cases it took more than 30 minutes. This appears to be related to the complexity of the path constraints that need to be solved.

SANTE [12] combines static and dynamic analysis to reduce the number of false positives. This is aimed mainly at test generation and they do not use model checking for defect detection. The key idea, like ours, is that slicing can reduce the size of the program that needs to be analysed.

There are numerous approaches to model checking and we summarise a few key ones here. Bandera is a SMC that allows the verification of user-defined properties in Java programs [4]. The checking process applies slicing to the program, user-guided data abstraction over the slice and the resulting abstracted version of the program is model checked. The major drawback of this approach is that user input is required for the data abstraction. Such a manual process is tedious and impossible to apply in practice in large code bases.

The Static Driver Verifier (SDV) [1], based on SLAM is an SMC for verifying user-defined properties on sequential C programs. C programs are abstracted using predicate abstraction with an initial set of predicates derived from the property. SDV then employs iterative counter-example guided abstraction refinement (CEGAR) to determine if the user-defined property is satisfied. However, the authors state that: “SLAM is unable to handle very large programs (with hundreds of thousands of lines of code)” [1].

Similar to SDV, the Berkeley Lazy Abstraction Software verification Tool (BLAST), is a SMC for verifying properties in C programs [13]. It applies a technique called lazy abstraction during the refinement process. While this improves the scalability of the CEGAR approach, we have been unable to use it for our work.

The C–Bounded Model Checker (CBMC) [3] and Low-level Bounded Model Checker (LLBMC) [2] verify properties in C programs via bounded model checking. In CBMC, the C program is abstracted once by unwinding the loop structures (including backward goto statements) according to the ‘unwind’ parameter. Function calls are also inlined. Optionally, slicing can be applied on the C program. LLBMC uses the bit code representation of the C program to perform bounded model checking.

The main drawback of such approaches is their sensitivity to the ‘unwind’ parameter. A small value can reduce the accuracy of the verification, but a large value can increase the runtime unnecessarily. Determining what is the best value needs significant experimentation and determining this value a-priori is not possible for large code bases.

3 Illustrative Example

Memory leak is a common defect in programs written in C. A memory leak happens when memory that has been previously allocated (via ‘malloc’ or similar memory allocation function in C), is not deallocated (via ‘free’ or similar) prior to the program ending.

Figure 1 shows a small C program for motivation purposes. At line 3, 128 bytes are allocated and the starting address of those 128 bytes is stored in pointer ‘p’. At lines 14 and 19, memory pointed to by ‘p’ is deallocated. At lines 17–22, the memory pointed to by ‘p’ is deallocated only when ‘retval’ is equal to -1 and ‘p’ is not equal to NULL. Thus the case when ‘retval’ is equal to -2 is not taken into account and ‘free’ is not called. Therefore, memory leaks at the end of this function.

Figure 2 presents the control flow graph for the example C function using SSA form [14]. In SSA form, each variable is defined exactly once; existing variables are split into separate versions, and a ‘phi’ function is used at merge points. For example, variable ‘retval’ is assigned values at lines 5 and 10. Both of these constant values reach the ‘end’ basic block, therefore, the first intermediate statement in that basic block (statement ‘P11’) is the definition of ‘retval’ as the ‘phi’ function between values -1 and -2. ‘P11’ states that the value of ‘retval’ is either

```
1  int foo {
2    int retval = 0;
3    char *p = malloc(128);
4    if( p == NULL ) {
5      retval = −1;
6      goto end;
7    }
8    FILE *f = fopen("test.c"
         ,"ro");
9    if( f == NULL ) {
10     retval = −2;
11     goto end;
12   }
13   fclose(f);
14   free(p);
15   return 0;
16 end:
17   if( retval == −1 ) {
18     if( p != NULL ) {
19       free(p);
20       p = NULL;
21     }
22   }
23   return retval;
24 }
```

Fig. 1. Motivating Example with a Memory Leak

-1 or -2 depending on which path was followed. Other statements of interest include ‘P0’ which does the allocation of memory, ‘P9’ and ‘P16’ which do the deallocation of memory, and both ‘P10’ and ‘P18’ which are exit points for this function.

As part of our analysis we make use of program slicing [9]. A program slice is the set of statements in a program that may affect the value of a variable at some point of interest; commonly referred to as the slicing criterion. If we use ‘P18’ as our slicing criterion, we are interested in all statements that may affect the value of ‘retval’.

The slice therefore includes the branches into ‘bb7’, namely ‘P17’, ‘P15’ and ‘P13’ and their dependencies, ‘P11’, ‘P12’ and ‘P14’, and the dependencies of these three basic blocks, namely, ‘P3’, ‘P7’, ‘P13’ and ‘P15’ (these last two already in the set), and so on.

In this case the slice contains all statements in the example except for those in the shaded basic block ‘bb4’; i.e., the slice of slicing criterion ‘P18’ are all statements that are not shaded.

We now describe some of the key steps in the abstraction process for this example. As part of the data abstraction process for pointers, we use the values ‘NULL_ADDR’, ‘MEM_ALLOC’ and ‘OTHER’ to indicate a null pointer, a pointer pointing to an allocated block of memory and a pointer pointing to other addresses respectively. The slice in our example has 4 predicates: ‘P1: p == NULL’, ‘P5: f == NULL’, ‘P12: retval == -1’ and ‘P14: p != NULL’. Three of the four predicates can be effectively represented using data abstraction, viz., using ‘NULL_ADDR’ . As a result, only one boolean variable (say ’b’) needs to be created to keep track of the predicate ‘retval == -1’. For this predicate, the instruction at ‘P11’ defines ‘retval’. The instructions ‘b = true’ and ‘b = false’ are added to predecessor basic blocks ‘bb’ and ‘bb3’, respectively, prior to the last branching instruction, to abstract the incoming values of the phi function (-1 and -2, respectively). Thus by using data abstraction first, we reduce the number of extra variables that need to be introduced for predicate abstraction.

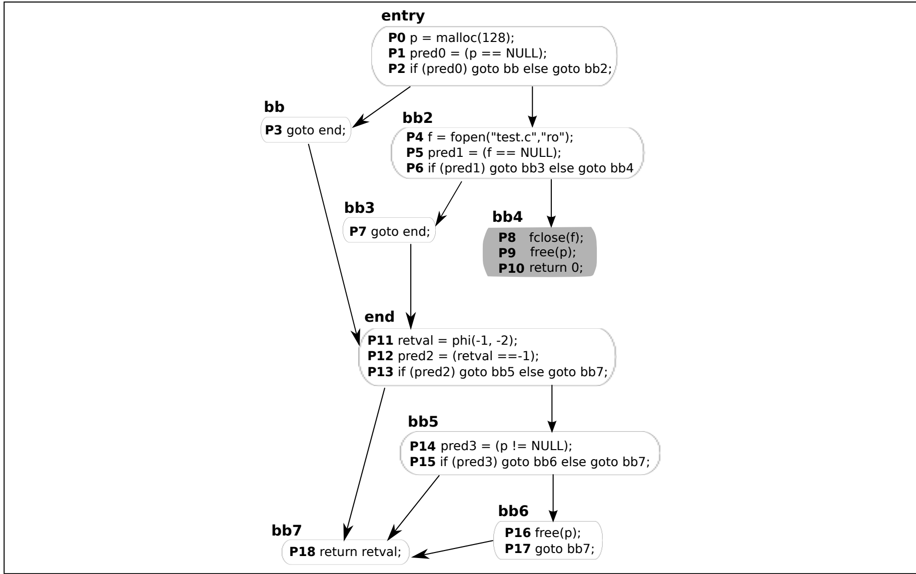

Fig. 2. Control Flow Graph in SSA Form for the Example of Figure 1

To handle the memory allocation at 'P0', two abstracted instructions are created. The first is a declaration of a variable that keeps track of the pointer 'p', and the second is noting that the result of invoking an external library function ('malloc') can return one of two values; the newly allocated address or NULL if there is not enough memory available. This indecisive result is expressed by using a non-deterministic selection statement which covers the two cases 'p = NULL' and 'p = MEM_ALLOC'. Similarly the call to 'fopen' at 'P4' defines the value of variable 'f'. We assume that the 'fopen' can only return either 'NULL' (which indicates failure) or 'OTHER' (which indicates success) since the return value is a pointer and there is no memory allocation involved.

Although, not present in the above example, we show how integer values are handled. For instance, a particular data abstraction rule could define that some integers can be abstracted to the range {'below1', 'between1&9', 'above9'}. Based on this, a control predicate 'x<1' can be expressed as 'x==below1'; therefore, 'x' can be data abstracted and no extra boolean variable needs to be added to the model. On the other hand, the control predicates like 'y==5' requires a new boolean variable since it cannot be represented using the earlier data abstraction rule. To reiterate, data abstraction followed by predicated abstraction reduces the introduction of extra variables.

In the next section we describe our analysis that combines program analysis and model checking techniques. The description of the specialised abstraction will explain how the values used in the above example arise.

In the next section we describe our analysis that combines program analysis and model checking techniques. The description of the specialised abstraction will explain how the values used in the above example arise.

4 Model-Based Analysis

Recall that the aim of our model-based analysis is to use model checking techniques to find defects such as memory leaks in large C code bases effectively. That is, the analysis should take only a few hours to complete code that has around a million lines. In contrast to most SMCs, where one model is generated per program, we generate multiple models per program. We use a specialised abstraction that aims to reduce the size of the models. Model checking is then performed separately over each model resulting in reduction in the search space.

Given a defect type (e.g., memory leak), we use a demand-driven approach to identify all statements in the program where that defect *may* happen. We call these locations "potential defects". The list of potential defects is created by using a static analyser that runs very quickly. For each potential defect, we create a model using a specialised abstraction that is checked by a model checker.

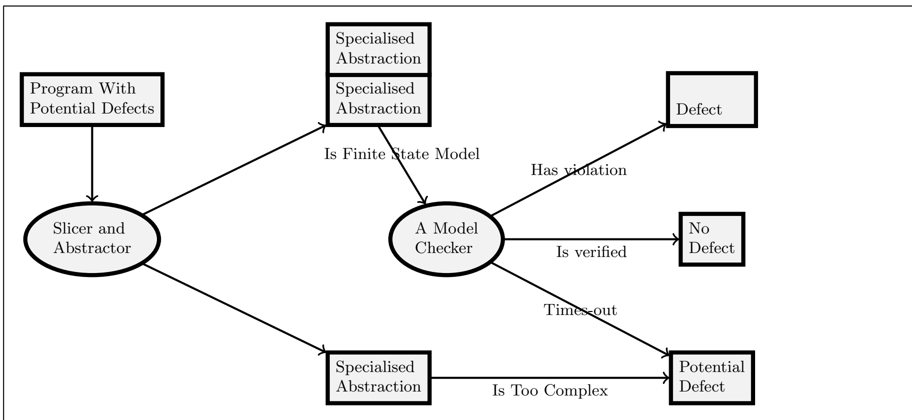

Fig. 3. Architecture of the Model-based Analysis Approach

Figure 3 illustrates our approach to model-based analysis: for each potential defect we first determine all the other statements in the program that are dependent on the statement of the potential defect; this step effectively creates a slice of the program starting at the potential defect statement, taking into account only variables and conditions that are relevant for the analysis to be applied. We then reduce the slice to a specialised abstraction which in turn is transformed into a finite state model that is fed into a model checker.

Automatic data abstraction is used to discard irrelevant ranges of values unrelated to the property to be checked. We also use predicate abstraction on statements when it is not possible to use data abstraction, resulting also in smaller abstracted models. This combination of data and predicate abstractions alleviates the expensive predicate solving, and reduces the complexity of the resulting model; again, improving performance of the analysis.

Owing to the use of automatic abstraction techniques it is possible that an abstraction proves too complex to transform into a model. In such cases the particular potential defect is not analysed further. The complex abstractions result because of the presence of unsupported operations for the data abstraction mechanism and also because of limitations in the predicate solver. In contrast to the CEGAR approach, where multiple iterations of generation of an abstraction may happen, we only abstract once. We generate simpler models by specialising the property (i.e., the defect type in our case) to be checked. Thus our technique is faster than CEGAR based approaches but less accurate. This tradeoff enables us to handle large code bases without reducing the value of the reported defects.

Once the model is run through the model checker, either a counterexample is generated; in which case the potential defect violates the property being checked for and is therefore a defect, or no property violation happens; in which case the potential defect is not a defect.

Algorithm 1. High Level Algorithm

```
procedure DefectSpecificModelCheck(program)
    potentialDefects := GetPotentialDefectsList(program)
    defects := ∅
    for each pd in potentialDefects do
        slice := Slicing(program, pd)
        if slice is executable then
            model := SpecialisedAbstraction(slice, pd)
            if model is not empty then
                result := ModelCheck(model, fixedProperty)
                if fixedProperty is not satisfied then
                    defects := defects ∪ { pd }
                end if
                potentialDefects := potentialDefects \ { pd }
            end if
        end if
    end for
    Print(defects)
end procedure
```

The Procedure DefectSpecificModelCheck in Algorithm 1 depicts our model-based analysis at a high level. First a list of 'potentialDefects' is generated for the program. This is a simple static analysis pass. For each potential defect, a slice of the code is obtained. If the resulting slice is a self-contained piece of code that can be executed, a model of it is generated via specialised abstraction. If a non-empty model is generated, it is run by a model checker and determined to be safe or unsafe; unsafe results are placed in the 'defects' list. Other cases lead to the potential defect remaining in the list of potential defects. The slicing and model checking components of the algorithm are standard. We explain our specialised abstraction in the next section.

4.1 Specialised Abstraction

As mentioned earlier, our specialised abstraction for defect types makes use of data abstraction and predicate abstraction. Procedure SpecialisedAbstraction in Algorithm 2 describes the specialised abstraction for a given slice 'slice' and a potential defect 'pd'. The algorithm keeps track of a set of boolean variables ('boolVariables') and a set of data variables ('dataVariables'), as well as a list of abstracted instructions ('model').

Algorithm 2. Specialised Abstraction Algorithm

```
function SpecialisedAbstraction(slice,pd)
    boolVariables := ∅; dataVariables := ∅
    predicates := GetControlStatementPredicates(slice)
    for each pred in predicates do
        if pred cannot be expressed using DataAbstraction then
            boolVar := CreateBoolVariable(pred)
            boolVariables := boolVariables ∪ { boolVar}
        end if
    end for
    model := <>
    for each inst in slice do
        modelInst, dataVariables := ApplyDefectSpecificAbstraction(inst,
          pd, boolVariables, dataVariables)
        if modelInst is empty then
            return empty
        else
            model := model ⌢ modelInst
        end if
    end for
    return model
end function
```

For a given slice, we first determine all predicates in the control statements. If the predicate cannot be expressed using the specific defect data abstraction, a boolean variable is created for it and added to the set 'boolVariables'.

Each instruction in the slice is processed by 'ApplyDefectSpecificAbstraction' to generate the abstracted instruction ('modelInst') and update the set of data variables 'dataVariables'. Predicate abstraction is applied to predicates associated with boolean variables (i.e., in the set 'boolVariables'). If the instruction is too complex for the abstraction at hand, an empty model is returned. Otherwise the model is extended ($\frown$ is just concatenation) with the model representing the current instruction (*inst*) being processed.

Next we explain the details of the data and predicate abstraction (denoted by 'ApplyDefectSpecificAbstraction') for finding memory leak defects. Recall that we will create one model per potential defect. We consider a potential memory leak a pair of memory allocation and return statements. Thus each model will

have only one allocation that may leak at exactly one exit point. For memory leak detection, the 'dataVariables' of interest are pointers. A pointer value is represented by its abstracted address, address space and its offset. The abstracted address, used in the data abstraction process, can either be NULL ('NULL_ADDR'), point to the allocated memory in question ('MEM_ALLOC_ADDR'), or point elsewhere ('OTHER_ADDR'). The address space attribute keeps track of relevant information of the memory contained in the address of the pointer; there are three possible values. Pointers that point to memory containing an address to an allocated memory are marked as 'PARENT_ALLOC_ADDR'. For example, a double pointer 'p' (i.e.,'void **p') will be flagged as 'PARENT_ALLOC_ADDR' if the memory it points to (i.e.,'*p') includes a memory allocation (e.g., '*p = malloc(..)'). This information allows for the detection of indirect deallocations or escapes of 'MEM_ALLOC_ADDR'. In cases where a pointer address is reachable from outside the function being analysed (e.g., argument passed by reference), it is marked as 'ESCAPE_ ADDR'. This way it is possible to identify when the 'MEM_ ALLOC_ADDR' escapes. If the pointer does not point to a parent compound data type and does not escape, its value is 'NONE'. Last, the offset of the pointer is stored as an integer, which is needed for supporting arithmetic operations.

The arithmetic operations that can be represented by this abstraction are limited to additions and subtractions between pointers and integers. As a result, only the offset section of pointers is affected in these operations. In the case of logical operations, our approach only supports equals and not equals predicates. The address space attribute is ignored when computing comparisons as they do not represent the value of the address itself. The particular case of comparing between two pointers evaluating to 'OTHER_ADDR' is handled by assigning 'true or false' non-deterministically.

The address space attribute of pointers is modified as a side effect of definitions of external pointers, memory writes and memory copies. First, when a pointer is defined externally (e.g., a pointer returned by a library function), the address space of that pointer is set to 'ESCAPE_ADDR'. Secondly, we need to propagate the address space attribute when a 'child' pointer is stored in a memory pointed by a 'parent' pointer. The 'parent' pointer is flagged as 'PARENT_ALLOC_ADDR' if the 'child' pointer address is a memory allocation or it is marked as 'PARENT_ALLOC_ADDR'. On the other hand, the 'ESCAPE_ADDR' flag is propagated to the 'child' pointer from the 'parent' pointer if it is the case. Finally, memory copies (e.g., using memcpy(...)) sets the destination pointer as 'PARENT_ALLOC_ADDR' if the source pointer is marked as such. In all of the three cases stated above, our algorithm declares that memory leak is not possible and defines an end state in the model when a pointer address space needs to be set to 'PARENT_ALLOC_ADDR' and 'ESCAPE_ADDR' at the same time.

The address and address space abstractions are summarised in Figure 4.

$Address \in \{NULL_ADDR, MEM_ALLOC_ADDR, OTHER_ADDR\}$

$AddressSpace \in \{NONE, PARENT_ALLOC_ADDR, ESCAPE_ADDR\}$

$Offset \in \mathbb{Z}$

Fig. 4. Address data abstraction

Concretely, we represent pointers as integers: the address, address space and offset of the abstracted representation of pointers. They are extracted by using arithmetic modulus operations as defined in Figure 5.

$pointerAddress(ptr) \equiv (|\ ptr\ |\ \text{MOD}\ 10\ )\ \text{MOD}\ 3$

$pointerAddressSpace(ptr) \equiv (|\ ptr\ |\text{—}\ \text{MOD}\ 10\ )\ \text{DIV}\ 3$

$pointerOffset(ptr) \equiv ptr\ \text{DIV}\ 10$

Fig. 5. Operations to extract elements from pointers represented as integers

There are two limitations to using specialised abstraction. First, it does not support the analysis of user-defined properties, and second, the resulting model cannot be guaranteed to be non spurious. It may be inaccurate due to predicates that may be missing in the model. The first limitation results from the fact that each property needs a particular algorithm for the analysis. However, this is not a requirement in our case as we are interested in checking for known types of defects for which effective algorithms have been developed. To minimise the effects of the second limitation, the analysis accepts this fact and just leaves the potential defect as a potential defect, rather than attempting to generate a more accurate model.

4.2 Example Revisited

We now show the working of the memory leak abstraction technique on our running example presented in Figure 1. Recall that the slice for the criterion 'P18' is all the statements in non-shaded basic blocks. This slice has 4 predicates: 'P1: p == NULL', 'P5: f == NULL', 'P12: retval == -1' and 'P14: p != NULL'. Three of the four predicates can be effectively represented using our data abstraction, since 'NULL_ADDR' is a possible abstracted value. As indicated earlier, only one boolean variable needs to be created to keep track of the predicate 'retval == -1'.

We illustrate the processing of a couple of instructions. The instruction at 'P0' allocates memory via 'malloc' and stores the result in 'p'. This instruction

is modelled by two abstracted instructions: a declaration of a variable that keeps track of the pointer 'p', and the result of invoking an external library function that can return one of two values; the newly allocated address or NULL if there is not enough memory available. This indecisive result is expressed by using a non-deterministic selection statement which covers the two cases 'p = NULL' and 'p = MEM_ALLOC'. This is shown in the first **if − fi** statement in Figure 6.

The instruction at 'P11' defines the variable 'retval'; this variable affects the boolean variable associated with predicate 'retval == -1'. Assume the boolean variable is named 'b'. The instructions 'b = true' and 'b = false' are added to basic blocks 'bb' and 'bb3', respectively, prior to the last branching instruction, to abstract the incoming values of the phi function (-1 and -2, respectively).

The instruction at 'P4' is a call to the library function 'fopen' and defines the value of variable 'f'. This instruction generates a non-deterministic selection statement to represent the result of 'fopen' and an assignment that makes 'f' an escape address. In the first construct, we assume that the 'fopen' can only return either 'OTHER' or 'NULL' since the return value is a pointer and there is no memory allocation involved. The address is flagged as 'ESCAPE_ADDR' for safety as we assume that this address can be potentially reached interprocedurally. This last statement is not relevant in this particular example as there is no memory write to this address. However, it prevents other cases from reporting false positives.

4.3 Function Summaries and Interprocedural Support

We conclude the discussion of our approach by noting our use of standard function summaries to handle interprocedural analysis [15]. That is, function summaries of each function are created and then used at each calling site. A function summary is a collection of pre and post conditions that encapsulates how the inputs and outputs of a function are affected in the context of a function call. These predicates can represent relevant effects for the memory leak detection analysis such as pointer escapes, memory copies, memory allocations and deallocations. The summaries of functions from external library functions can be defined in a configuration file. In particular, summaries of common functions of the C library, such as 'malloc' and 'free', are used in the analysis.

Whenever a function summary is missing for a given (external) function, the algorithm makes use of the worst case scenario for the defect at hand. For example, for memory leak defects, we can safely assume that every pointer input is escaped and that every pointer output is in an escaped abstracted address space. Further, the return value is non-deterministically defined in this case to avoid missing defects that are not directly related with such calls.

5 Implementation

We have implemented our model-based analysis for detecting memory leaks using the Parfait static code analysis tool [16] and the SPIN [17] model checker. Given

our abstraction technique, the model checker we use need not have support for memory leaks.

Our slicing implementation is performed in two passes: a backward pass to calculate the control and data dependencies from the exit point back to the point of interest (the allocation), and a forward pass to track the uses of the allocation statement. This implementation makes use of Parfait's pointer alias analysis.

To implement predicate abstraction, we make use of Parfait's predicate module. When this module cannot determine the value of an abstracted boolean variable, e.g., the predicate is too complex for the module to solve, we assign a value non-deterministically. This module is fast but less precise than a theorem prover, again, as a tradeoff between precision and scalability.

Our implementation has some limitations in the abstraction and the interprocedural support. Our specialised abstraction method is occasionally unable to generate a model because of the use of a simple predicate solver. For instance, it is not able to express boolean variables, representing predicates, in terms of other boolean variables, and it cannot resolve predicates containing floating point values. Our interprocedural support is not complete. As explained in Section 4.3, we rely on Parfait's existing function summaries for detecting interprocedural defects. We have not extended Parfait to fully support our model-based analysis needs. So, some information is not considered in the generation of these summaries and, therefore, our algorithm can miss relevant information for the analysis.

We translate our specialised abstraction into the Promela which is the input language to the SPIN model checker. There are two features of the Promela language that deserve explanation as they differ from traditional programming languages. Promela's control flow is based on whether a statement is executable or not [17]. A statement is executable if it evaluates to a non-zero integer value; therefore, every statement in Promela returns a value. A statement can be an expression on its own, and expressions like '0;' are not executable as they do not evaluate to a non-zero value. We take advantage of this property of the language and use non-executable statements to specify end states. This combined with Promela's modelling of non-determinism using guarded statements enables us to represent different behaviours.

The generated Promela model for our memory leak example shown in Figure 1 is presented in Figure 6. It has been slightly modified to aid readability. The model contains only one active process, 'myProcess', which is enough for evaluating sequential properties such as memory leak. Two auxiliary global boolean variables are included in the construction of the model: 'memoryLeak' and 'exit'. The 'memoryLeak' variable will evaluate to true when the memory in question is allocated and has not been either freed nor escaped. The 'exit' variable is set to true in the block containing the exit statement in the potential defect. As a result, the logical temporal logic (LTL) property that expresses memory leak freedom is represented as `[](exit -> !memoryLeak)`, which means "at any state in the model, if exit holds true, memoryLeak is false".

```
bool memoryLeak = false;
bool exit = false;
active proctype myProcess() {
int p; //pointer p
int f; //pointer f
bool b; //representing
        //retval == -1
bool pred0, pred1;
bool pred2, pred3;
entry:
if
  :: true ->
     memoryLeak = true;
     p = MEM_ALLOC;
  :: true -> p = NULL;
fi;
pred0 = addrSpace(p) == NULL;
if
  :: (pred0) -> goto bb;
  :: else -> goto bb2;
fi;
bb: b = true; // retval = -1;
goto end;
bb2:
if
  ::true -> f = OTHER;
  ::true -> f = NULL;
fi;
f = setAddrSpace(f, ESCAPE_ADDR);
pred1 = addrSpace(f) == NULL;
if
  :: (pred1) -> goto bb3;
  :: else -> 0;
fi;
bb3: b = false; // retval = -2;
goto end;
end:
pred2 = b;
if
  :: (pred2) -> goto bb5;
  :: else -> goto bb7;
fi;
bb5: pred3 = addrSpace(p) != NULL;
if
  :: (pred3) -> goto bb6;
  :: else -> goto bb7;
fi;
bb6:
if
  :: addrSpace(p) == MEM_ALLOC ->
       memoryLeak = false; 0;
  :: else -> skip;
fi;
goto bb7;
bb7: exit = true;
0;
}
```

Fig. 6. Promela model for example of Figure 1

The Promela model is then passed to SPIN to perform the model checking. If SPIN reports an error, the memory leak is reported as such. We make use of a timeout in case the SPIN processing time takes too long. In practice, test runs indicate that 10 seconds is sufficient time prior to timeout for the size and complexity of the models our analysis generates.

6 Experimental Results

We measure the effectiveness of our technique and report both precision and recall [18]. We evaluated our approach by running two sets of experiments. The first measured the precision and recall of the results against existing benchmarks from the program analysis community. The second measured precision and performance against a large open source code base. For this case it is not possible to measure recall as the list of all defects in the large code base is not known. We also compare the results produced by the LLBMC model-checker [2] and results produced by a purely static-analysis approach using Parfait [16]. We chose LLBMC because, unlike other model-checkers for C, it supports the detection of memory leaks.

6.1 Evaluation of Precision and Recall Against Benchmarks

We use the subset of memory leak benchmarks available in the NIST SAMATE [19] suite and the Error Detection Test Suite from Iowa State University [20]. These suites contains small benchmarks (the average number of lines of code without comments/blank lines is also shown) with known memory leaks. Furthermore each program has one defect per benchmark. Knowing the location of the defect in the code allows us to determine whether the results of our analysis are correct (hence a true positive), incorrect (hence a false positive) or whether defects were missed all together (hence a false negative). Precision and recall are then computed based on these values. It should be noted that we are not including benchmarks with allocation to global pointers as our implementation does not consider those cases as memory leaks.

Table 1. Small Benchmark Results

Benchmark	Total Defects	Avg LOC	Tool	True +	False +	False -	Precision	Recall	Time (sec)
SAMATE	50	22	Model-based	44	0	6	100%	88%	34
			LLBMC	39	0	11	100%	86.67%	6.38
			Parfait	44	0	6	100%	88%	6.53
IOWA	25	35	Model-based	17	0	8	100%	68%	20.44
			LLBMC	22	0	3	100%	88%	7.67
			Parfait	13	0	12	100%	52%	1.05

Table 1 shows the results of this evaluation. For each benchmark suite we list the total number of defects in the benchmarks, the reported true positives, the number of false positives and false negatives, and compute the precision and recall for the analysis results. As can be seen, all reports produced by the analysis in both benchmarks are correct, hence a false positive rate of 0 and therefore precision of 100%.

The false negatives for the model-based approach in these benchmarks are mostly due to the incomplete function summaries. Thus, our algorithm is unable to process relevant statements or has to be more conservative when abstracting calls to those functions; e.g., assume that every parameter in the call escapes. LLBMC also suffers from a similar problem. More than half the number of false negatives are because LLBMC does not recognise `strdup` as allocating memory or because of exceptions. LLBMC ran out of memory on three of the programs in the SAMATE suite. In terms of runtime, our model-based approach will generate models for all potential memory leaks in the code, whereas the LLBMC approach will stop when it first encounters a memory leak defect.

Our model-based approach generates 125 potential defects for the various programs in SAMATE. For these 125 potential defects 122 models were generated, i.e., 3 models were too complex. The model-checking process is able to prove violations in 44 of the 122 models. Thus it missed finding 6 defects. Similarly,

for the IOWA data set, our approach generates 37 potential defects for which 32 models were generated with 5 models being too complex. The model-checking process verified 17 out of the 32 models.

Parfait will also process all potential memory leaks in the program, however, it uses less expensive data flow techniques which have been optimised over the years. Thus its performance on small benchmarks is excellent both in terms of accuracy and run-time.

The main conclusion we wish to draw from this experiment is that model checking does work for small programs. There is no advantage in generating models on small programs as shown by the significantly larger time taken by the model-based analysis. The model checking process (using LLBMC) is effective although in the case of the benchmarks from the SAMATE set, Parfait is much better than LLBMC. The next section evaluates the effectiveness of slicing and the specialised abstraction technique as applied to a large system.

6.2 Evaluation Using OpenJDK

We ran our analysis over the OpenJDK 7 build 136, on a Sun Ultra24 machine with Core2Duo 3.3Ghz with 6GB RAM of memory. The OpenJDK version has more than 1.4 million lines of non-commented C/C++ code. We used version 5.2.2 of the SPIN model checker. For this system we compare our approach with only the dataflow analysis built into Parfait. Unfortunately LLBMC was unable to run successfully on the above system; thus we are unable to present any results for it. Also tools such as Sparrow and Saber were not available for our experimentation. So no comparison with their approaches can be made.

Although our implementation uses Parfait, the memory leak detection in Parfait is a separate pass that can be run in stand-alone mode. These features of Parfait are independent from each other and hence the comparison does not suffer from any internal biases.

Since neither us nor the authors of the code know where all memory leaks are in the OpenJDK code base, we can only measure precision of the results produced by the analysis, as measuring recall requires knowledge of the location of all memory leaks. Precision is measured by manually inspecting each report produced by the analysis and determining whether the report is correct (i.e., true positive) or incorrect (i.e., false positive). Given the industrial nature of the code base, and its size we used this code base to measure the runtime performance of the analysis.

Table 2. OpenJDK Benchmark Results

Tool	Total Reports	True Positive	False Positive	Relative False Negative	Precision	Performance
Parfait	38	37	1	25	97.37%	55 sec
Model-based	71	60	11	2	84.5%	1:19 hr

Table 2 shows the results of this evaluation. Of the 71 defects reported by the analysis, 60 are correct and 11 are false positives, leading to a precision of 84.5%. The precision in this case is less when compared to the SAMATE and IOWA benchmarks. This is because the latter benchmarks are small and not necessarily fully representative of real code, i.e., they lack in complexity of the code. The false positives included in this test were caused by limitation in the implementation of the slicing module. In some cases our slicing algorithm fails to include uses of parent pointers in the slice and, thus, relevant escapes are not added to the model. We also measure the *relative* false negative for both tools based on the true positive results. For each true positive, we determine whether the other tool reports it or not; if it does not, then it is a false negative. 35 reports were found by both tools. So Parfait missed 25 memory leaks reported by model-based analysis, whereas our approach only missed 2 reported by Parfait. That is, 62.2% more defects were correctly reported.

These results were expected as the model-based approach is more exhaustive and thus more computationally expensive than the data flow technique. The extra performance runtime is still within acceptable time for scaling the analysis to millions of lines of code.

Although the general algorithm reduces the number of model construction cancellations by making safe assumptions and using non-deterministic assignments, we are still forced to abandon the analysis in several of these cases. We also found some data and predicate abstraction clashes; e.g., operations composed of both predicate and data abstracted variables. These instances are handled by discarding predicate abstraction and applying data abstraction on the fly when possible. Otherwise the analysis is cancelled. For example for the OpenJDK code base, we only generate models for 67% of the potential defects.

We also examined the minimum, maximum and average number of states of the models generated by the analysis. The smallest model had 7 states while the largest model had close to 1.8 million states. The average size of the model was about 92,000 states. For models that had a defect, the smallest model had 12 states and the largest model had about 800,000 states and the average size was 14,000 states. Of the 4,186 models generated only 13 models timeout given the 10 second limit. We can therefore conclude that model checking was, in general, effective. We are investigating if there is any correlation between the number of states in the models and the overall performance. Intuitively, large models which do not have a defect lead to reduction in performance as all states need to be examined. However, a large model which has a defect that can be found without computing the entire space will not affect performance.

6.3 Threats to Validity

We now discuss a few threats to validity. The first is related to the selection of benchmarks. We have chosen IOWA and SAMATE as representative of small programs and OpenJDK for a large system. Experimenting with other systems may yield different results. In our evaluation we have converted the specialised abstraction to Promela and used the SPIN model checker. We could have

translated the abstraction to C (and used CBMC [3]) or to LLVM bit code (and used LLBMC). While the exact timing will vary, in all these cases we do not think that this would change the overall validity of our approach. The initial list of the potential defects has a significant influence on the process. That is, larger the list the more abstractions that need to be created. If one can use a more sophisticated (and hence potentially more expensive) static analyser, our results could be improved. But the identification of the optimal point is an open question. Also we are using the function summaries from Parfait. This results in a fair comparison for OpenJDK but also points to an area that could be improved.

7 Conclusion and Future Work

In this paper we have presented our model-based analysis approach to finding defects, which makes use of model checking techniques in conjunction with program analysis. The aim of our approach was to develop a defect detection mechanism that can scale to thousands and millions of lines of code, without loss in precision, at the expense of missing some defects. Our approach has been applied to a version OpenJDK that has approximately 1.4 million lines of code which is much larger than programs used in the TACAS 2013 and 2014 benchmarks and tools such as Saber [8].

The key to our analysis is the use of a specialised abstraction that relies on both data and predicate abstraction and the use of multiple models (one per potential defect) to generate a number of small models each of which can be model checked. Thus we are able to leverage the strengths of program analysis and model checking. As our abstraction process limits the size of the models and hence could have false negatives.

Our implementation results for memory leak detection show that the analysis scales well to large code bases without detracting from precision, at the expense of missing some defects. When compared to data flow analysis, our analysis was much slower but reported 62% more defects. The runtime performance of our analysis is reasonable to include this analysis in a static code analysis tool that runs over millions of lines of code.

There are two main avenues for further research. The first is a more detailed comparison. For instance, we could compare our approach with other techniques developed for memory leak (e.g., if we get access to Saber). Second, we believe that our analysis can be applied to other types of defects which needs validation.

Acknowledgements. Initial work was conducted by the first author under Prof. Hayes's supervision. We thank Daniel Wainwright and Matthew Johnson for their assistance with our experiments.

References

1. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Communications of the ACM 54, 68–76 (2011)
2. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 146–161. Springer, Heidelberg (2012)
3. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
4. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Hongjun, Z.: Bandera: Extracting finite-state models from Java source code. In: Proceedings of the International Conference on Software Engineering, pp. 439–448 (2000)
5. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), pp. 57–68. ACM Press (June 2002)
6. Dor, N., Adams, S., Das, M., Yang, Z.: Software validation via scalable path-sensitive value flow analysis. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 12–22. ACM (2004)
7. Jung, Y., Yi, K.: Practical memory leak detector based on parameterized procedural summaries. In: Proceedings of the 7th International Symposium on Memory Management (ISMM), pp. 131–140 (2008)
8. Sui, Y., Ye, D., Xue, J.: Static memory leak detection using full-sparse value-flow analysis. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA), pp. 254–264. ACM (2012)
9. Yatapanage, N., Winter, K., Zafar, S.: Slicing behavior tree models for verification. In: Calude, C.S., Sassone, V. (eds.) TCS 2010. IFIP AICT, vol. 323, pp. 125–139. Springer, Heidelberg (2010)
10. Park, M., Byun, T., Choi, Y.: Property-based code slicing for efficient verification of OSEK/VDX operating systems. In: Proceedings of the First International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS), pp. 69–84 (2012)
11. Kim, Y., Lee, J., Han, H., Choe, K.M.: Filtering false alarms of buffer overflow analysis using SMT solvers. Information and Software Technology 52(2), 210–219 (2010)
12. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: Proceedings of the ACM Symposium on Applied Computing (SAC), pp. 1284–1291 (2012)
13. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
14. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4), 451–490 (1991)
15. Hampapuram, H., Yang, Y., Das, M.: Symbolic path simulation in path-sensitive dataflow analysis. In: Proceeding of PASTE, pp. 52–58. ACM Press (2005)

16. Cifuentes, C., Keynes, N., Li, L., Hawes, N., Valdiviezo, M., Browne, A., Zimmermann, J., Craik, A., Teoh, D., Hoermann, C.: Static deep error checking in large system applications using Parfait. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 432–435. ACM (2011)
17. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual, 1st edn. Addison-Wesley Professional (2011)
18. Anderson, P.: The use and limitations of static-analysis tools to improve software quality. CrossTalk: The Journal of Defense Software Engineering, 18–21 (2008)
19. NIST: National Institute of Standards and Technology SAMATE Reference Dataset (SRD) project (January 2006), http://samate.nist.gov/SRD
20. Luecke, G.R., Coyle, J., Hoekstra, J., Kraeva, M., Li, Y., Taborskaia, O., Wang, Y.: A survey of systems for detecting serial run-time errors. Concurrency and Computation – Practice and Experience 18(15), 1885–1907 (2006)