Structural Testing of Executables*

Sébastien Bardin Philippe Herrmann
CEA, LIST
Software Reliability Lab
Boîte 65, Gif-sur-Yvette, F-91191 France
E-mail: first.name@cea.fr

Abstract

Verification is usually performed on a high-level view of the software, either specification or program source code. However in certain circumstances verification is more relevant when performed at the machine code level.This paper focuses on automatic test data generation from a standalone executable. Low-level analysis is much more difficult than high-level analysis since even the control-flow graph is not available and bit-level instructions have to be modelled faithfully. We show how "path-based" structural test data generation can be adapted from structured language to machine code, using both state-of-the-art technologies and innovative techniques. Our results have been implemented in a tool named OSMOSE *and encouraging experiments have been conducted.*

1. Introduction

The verification task is generally performed at the specification level (functional testing, model checking) or at the programming language level (structural testing, static analysis). In the latter case, by programming language we mean structured languages such as C or Java. It may look surprising that verification techniques do not check the machine code of the software under verification. After all, the machine code is truly what the computer executes. Actually, binary-level analysis is considered more difficult than other analyses, while being redundant with them.

We agree with the first point and explain why binary-level analysis is difficult later in this section. However, we claim that in certain circumstances machine code is (unfortunately!) the most relevant level at which to perform verification.

*Work partially funded by EDF and the *Software Factory/MoDriVal* project of the French cluster SYSTEM@TIC PARIS-REGION.

Here are three industrial cases where an analysis on the executable would be valuable.

1. In the critical systems'industry, a company may not have access to the program source code of a piece of software it has acquired. For example when the company is not a major customer for the vendor. Then these executables have to be certified without any programming language description.

2. In aeronautics, the DO-178B standard [15] imposes that verification must be performed on the binary level as soon as the conformity between the high level code and the machine code cannot be ensured. Since this binary level analysis is very expensive, constructors prefer to avoid any technology which would blur the conformity. Including optimising compilers, which would increase performances and lower costs.

3. In the computer security domain, an optimising compiler can produce a non-secure executable from secure source code as reported in [5]. The problem is due to very standard data-flow optimisations. For example, the compiler removes an operation consisting in setting to zero some memory value which is not used later in the program. The trick is that this operation was intended to erase a password value copied in clear text. While this operation was indeed useless from the functional point of view, it was essential from the security point of view. A similar bug was found during the Windows security push 2002 [12].

In the last case, a binary-level analysis is mandatory to detect the bug since the program source code is absolutely correct. This is known as the WYSINWYX phenomenon: *What You See Is Not What You eXecute* [5].

Relevance of binary-level analysis. We claim that binary-level analysis is relevant in at least two situations: when no high-level source code is available or when the increase of precision is essential. It is quite common that a company cannot have access to a high-level documentation, either the vendor does not provide the source code (Commercial Off-The-Shelf software) or the source code is sim-

ply lost (legacy code). Standard source code analysis relies on the assumption that the compiler preserves the program semantics. While it is realistic for standard reachability properties and standard compilation techniques, it cannot be trusted anymore in the case of strong safety/security requirements and highly sophisticated optimisations.

A third situation could be that of programs written in a combination of a structured language and an assembly language, typical of embedded systems. However this situation is significantly different from a standalone executable analysis, since we can rely on additional high-level information from both the compiler and the source code, like the symbol table or possible values of a `switch`-like instruction.

Major difficulties of binary-level analysis. Machine code analysis is undoubtedly more difficult than any higher-level analysis. The main problem is the so-called *Intermediate Representation recovery (IR recovery)*: since an executable is nothing more than a sequence of bits (see below), we have no information about any basic control-flow fact (such as functions, loops or variables) usually given for free in higher-level analysis.

```
:100D43000A0A4320636F6D70696C65722064656D78
:100D53006F6E73747261746965F6E2070726F6772F5
:100D6300616D0A0A00496E707574206F7065726157
:100D730074696F6E3A20272B272028414444292089
:100D83006F7220272D27202853554229203F20000A
:100CEA00759852758920758869758DF37BFF7A0D21
:100CFA007943120862120DDB8E648F65120DDB8E4A
:100D0A00668F677BFF7A0D7968120862120E0BEF05
:100D1A00B42B03D38001C3921130110CE56525670A
:100D2A00FFE5643566FE800BC3E5659567FFE564FC
:090D3A009566FE120DF780BD2242
:100E1C00496E707574204E756D626572203F2000AE
:100DBA008B688A69896A120E0BAB68056AE56AAAAA
:100DCA00697002056914F9EF120C63FFEFB40AE6C1
:010DDA0022F6
```

Actually we do not even know the number of instructions in the program since different instructions may have different sizes, instructions can overlap, there is no syntactic difference between instructions and data and finally we cannot determine reliably targets of dynamic jumps, i.e. `goto` instructions whose destination is evaluated at run-time.

Moreover, when performing binary-level analysis, low-level mechanisms have to be taken into account precisely while for higher-level analysis, coarser abstractions are usually sufficient. The most obvious one is machine arithmetic. Machine integers behave differently from usual integers. For example, considering 32-bit long unsigned integers, the operation 4294967295+1 returns 0. Floating-point numbers also behave very differently from real numbers. Actually, while machine integers can be modelled quite precisely by modulo arithmetic, there is no nice standard theory for floating-point numbers. Another low-level construct difficult to analyse are hardware and software interrupts.

Finally there is a great diversity of hardware architectures and instruction sets (ISA). They differ in terms of size of memory words, physical memory layout, interrupts and instructions. All ISA share some common operations such as arithmetical or bit-wise logical operations but there exist subtle variations (carry and overflow), optimised versions and even "exotic" instructions like machine-code string copy or decimal conversion.

Bit-vector theory. The bit-vector theory [6, 18] formalises standard machine instructions at the bit level. Formulae are interpreted over vectors of bits of a fixed length. Instructions include basic read and write operations, signed and unsigned views of bit-vectors, modulo arithmetic, logical bit-wise operations and other low-level instructions such as shift, rotation or concatenation. Floating-point arithmetic is usually not considered though it can be encoded. Satisfiability in bit-vector theory is decidable since the interpretation domain is finite.

The OSMOSE tool. The OSMOSE tool aims at performing automatic test data generation on standalone executable files. The test selection is white box since we do not consider any information other than the executable itself. The tool can be used to debug a program or to build a test set achieving some structural coverage criteria based on the control-flow graph, like instruction or branch coverage. This kind of test set is mandatory in certain critical domains, for example aeronautics [15]. The tool can find *"intrinsic"* bugs, i.e. executions which are undoubtedly faulty independently of the specification of the software, like division by 0, violation of the call-return policy, read of an unallocated memory cell or jump to an incorrect instruction. Functional bugs are out of scope since we do not have access to any executable specification.

We focus on reactive systems, commonly found in embedded software. Reactive systems can interact with an environment *via* sensors and actuators. In this case, a test data is an initial valuation for input data and a sequence of values read on each sensor. The user has to provide a description of the environment, declaring volatile memory addresses.

A very strong requirement of the OSMOSE project is to be as independent as possible from any particular architecture or instruction set, so that users can add their own architectures without any assistance from the developers of OSMOSE. This is achieved through a generic software architecture arranged around a notion of generic machine code. The tool currently handles three processors: the Intel 8051, the Motorola 6800 and the Freescale PowerPC 550. The first two processors are 8-bit and the last one is 32-bit. A consequence of being generic is that OSMOSE runs tests in simulation mode rather than in exact mode. This is essential unless running the tool on the exact architecture targeted by the executable under test.

Technologies. Binary-level analysers must first build a high-level model of the software under investigation. Then verification techniques may be used. Our verification technology is based on structural test data generation by computing a path predicate from a control path and solving this predicate. Our predicates are expressed in the bit-vector theory, so we also need a constraint solver for this theory. It turns out that the OSMOSE tool is organised around three basic technologies:

- *Path-based test data generation.* We generate tests from a white box perspective, more precisely we follow the recent and promising path-based test data generation approach [10, 20, 23]. To the best of our knowledge it is the first time this approach is used for binary-level testing. The core idea of the test data generation technique is to select a first path π_1 in the control-flow graph (here: in our high-level model) and extract its *path predicate* P_{π_1}. *The path predicate* P_π *is a constraint over the program input values such that any test case whose input data satisfy* P_π *will follow the path* π. A test data covering the path π_1 is found as a solution of P_{π_1}. Then a new path π_2 is selected and the process is iterated until the coverage objective has been achieved.

  A major improvement of path-based test data generation is the concept of concolic execution [20]. It means that a concrete execution is running in parallel to the symbolic execution, collecting relevant information along the concrete execution path to help the symbolic execution. In the original approach, the concrete execution is used to find a feasible initial path and to discover on-the-fly the program's CFG [10, 20, 23], or to approximate complex instructions like non-linear constraints or library function calls [10, 20].

- *Bit-vector constraint solving.* The test generation method described above relies on solving path predicates. While test data generation tools from high-level descriptions [10, 13, 20, 23] are usually based on integer constraints (classically bounded arithmetic [13, 23] or linear arithmetic [10, 20]), our constraints are expressed in the bit-vector theory. Our approach is based on the constraint programming paradigm [1], which is very flexible and allows us to encode all "exotic" instructions we may find in instruction sets. We write our own constraint solver on top of an existing one for usual integer constraints.

- *IR recovery.* We use an innovative combination of static and dynamic analysis to build an abstract high-level model of the software. First, a static analysis creates a coarse model. Then if new parts of the program are discovered during the test data generation phase the high-level model is updated and a static analysis is re-launched. It does not need to be very precise since a complementary dynamic analysis is performed, avoiding difficulties inherent to purely static techniques [4, 5].

Limitations. Like most test data generation tools [10, 20, 23], we do not consider floating-point arithmetic constraints. Some solutions have been proposed in the constraint programming community. But they mostly relax the problem to real arithmetic [13] or suffers from slow convergence phenomenon [3]. We also do not address interrupts, and are not aware of any verification technology handling this issue.

Contributions. There are three main contributions in this paper.

1. It is the first time a path-based structural test data generation technique is applied directly at the binary level. Our work pinpoints the main issues and shows how to adapt existing techniques from structured languages to machine code. We show also how the testing perspective simplifies crucial issues like IR recovery.
2. We propose innovative solutions for certain aspects: an enhanced concolic execution geared towards alias handling, dynamic target detection and early pruning of the path search ; and a combination of static and dynamic approaches to solve the IR recovery problem.
3. These results have been implemented in the first structural test data generation tool for executables. Our implementation and first experiments demonstrate the feasibility of the approach. The tool is also largely architecture-independent and can currently handle three different architectures and machine codes (8051, 6800, PowerPC 550).

Related work. We are not aware of any work dedicated to structural test data generation on standalone executables. Some verification tools work on low level description, but they have access to additional information from the program source code. We can cite for example jMoped [7, 19] or tools from the commercial company Absint [21], geared towards performance estimation rather than verification. Our testing technology is inspired by the path-based approach with concolic execution pioneered in [23] and extended in [10, 20]. However this work considers structured languages (C or Java) and ideal arithmetic. Finally IR recovery tools work on machine code, but do not perform any verification task. While commercial tools like [22] have problems with dynamic jumps, dedicated static analysis techniques have been developed recently [4, 5]. A more extensive description of related work is given in section 6.

Outline. The remaining part of the paper is organised as follows. The next two sections describe our core technologies: test data generation in section 2 and IR recovery in section 3. Section 4 presents the OSMOSE tool and its implementation. Section 5 describes some experiments with the tool. Finally section 6 discusses related work and section 7 concludes and presents future work.

2. Test data generation

Our test data generation algorithm follows the path-based principle: the idea is to enumerate all paths and for each path to compute its path predicate and solve it. The solution is a test data covering the path. The procedure is parametrised by a structural termination criterion, like instruction or branch coverage. Once the full coverage is achieved, the test data generation stops. We use a bounded depth-first traversal of the control-flow graph to enumerate all paths in a recursive manner. This is a standard strategy [10, 20, 23] which allows constraints to be added incrementally, and requires only a minimal change to get a new path predicate by reusing the path prefix up to the last choice point in the program.

Compare to [10, 20, 23], we work on an (abstract) control-flow graph (ACFG) rather than on the program itself. The ACFG allows us to use static techniques for IR recovery in addition to the dynamic exploration of the executable. The test data generation algorithm and the IR recovery mechanism are deeply interwoven and the ACFG may be updated during the test generation. For the sake of simplicity, we consider in this section that the ACFG is precomputed once and for all before the test data generation is launched. The whole IR recovery mechanism is described in section 3.

ACFG nodes can be either blocks (**block**) of basic instructions like common arithmetic and bit-wise logic operations, function calls (**call**), conditional statements (**ite**), static jumps to a predefined target (**goto**) or dynamic jumps to a computed target (**cgoto**).

Algorithm 1 presents the basic idea of our test data generation algorithm. Choice points in a machine code program are conditionals and dynamic jumps. For conditional, we just force the search to take the *"if"* or *"else"* branch by adding to the path predicate the conditional or its negation. In the case of dynamic jumps, we explore each possible target by constraining the argument of the jump (usually an arithmetic expression over registers) to take each possible value in turn. Static jump does not modify the path predicate. Function calls are inlined and managed as static jumps. Basic instructions are translated into formulae by the procedure `atomic`. The ACFG is given by its nodes with methods `.addr` and `.next` to access the address of the instruction (in the executable) and its successor nodes in the ACFG. The external procedure `solve` returns a solution of a constraint or the `unsat` exception in case of unsatisfiability. We present the algorithm for an all-path coverage termination criterion. To adapt the algorithm to other criteria, the program must keep a set of uncovered items U, and each time a path predicate is solved, items covered by the execution are removed. The program stops as soon as U is empty.

Some important implementation details are omitted. First formulae are added incrementally to our solver to take advantage of incremental solving and detect infeasible paths early. Second the user can set up different parameters to limit the search (depth bound or time-out for the constraint solver). Finally a *concolic execution* is preferred to a purely symbolic execution.This feature is described below.

algorithm GENTEST1(node_init)
input : initial node node_init
parameter : atomic, node.addr, node.next
output: set of test data Res
1: Res $\leftarrow \emptyset$
2: REC(node_init, $\top$)
3: **return** Res
procedure REC(node, Φ)
input : node, constraint Φ
parameter : Res, atomic, node.addr, node.next
output: no result, update Res
1: **Case** node **of**
2: | $\varepsilon \rightarrow$ */* end node */*
3: **try** $S_p \leftarrow$ SOLVE(Φ) ; Res $\leftarrow$ Res $\cup \{S_p\}$
4: **with** unsat $\rightarrow$ ();
5: **end try**
6: | block instr $\rightarrow$ REC(node.next, $\Phi \wedge$ ATOMIC(instr))
7: | goto tnode $\rightarrow$ REC(tnode, Φ)
8: | call fnode $\rightarrow$ REC(fnode, Φ)
9: | ite(cond,inode,tnode) $\rightarrow$
10: REC(inode, $\Phi \wedge$ cond);REC(tnode, $\Phi \wedge \neg$cond)
11: | cgoto expr $\rightarrow$
12: **for all** tnode $\in$ node.next **do**
13: REC(tnode, $\Phi \wedge$ expr $=$ tnode.addr)
14: **end for**
15: **end case**

Algorithm 1: Basic test data generation algorithm

Concolic testing. We follow the concolic principle with two executions (concrete and symbolic) running in parallel. We enhance the approach by adding a third *"semi-concrete"* execution dynamically detecting constant values at each step of the execution (see Algo. 3 on page 11).

- The new semi-concrete execution is used to prune the path search by detecting on-the-fly trivial cases of infeasible paths (e.g. conditional or dynamic jumps evaluated over constant values) and avoid calls to the constraint solver on inconsistent formulae.
- The concrete execution is classically used to detect a

first feasible path but also in an innovative way to handle alias constraints (see below) and to dynamically detect new targets for dynamic jumps (see section 3).

Alias. We say that two memory cells are in an alias relationship when one of them contains the address of the second. Aliasing is known to be a very difficult point in software analysis since tracking variable modifications becomes much more problematic. In presence of aliases the path predicate is no longer strong enough to ensure that the right path will be followed at execution.

It turns out happily that aliasing is a bit less difficult from a testing perspective than from a static one, since we do not need to compute a safe approximation of all possible alias relationships. We use the following solution: the concrete execution is analysed to extract the aliasing relationships existing in the concrete trace and add them to the path predicate. In a sense, our algorithm is modified to deal with (path,alias)-predicates rather than path predicates only. The good point is that the solution found (if any) is sure to follow the right execution path. The bad point is that this (path,alias)-predicate may be infeasible while the path is feasible with another alias constraint. A solution is to enumerate a fixed number of alternative alias constraints for this path by relaxing some of the constraints. Then we need to check that the generated data input does lead to an execution following the right path.

This technique allows us to discover aliasing relationships depending only on the memory layout. This is orthogonal to [20] where syntactic alias relationships are extracted from the C program, mainly from type declarations and alias expressions in branch conditions.

Functions. Functions are inlined. Recursive functions are allowed since the bounded depth first search prevents us from infinite looping. A modular analysis of function calls would be more satisfactory. However it is not clear how to perform such a modular analysis for structural test data generation. [11, 14] propose some solutions. The first one seems to be quite inaccurate because of a very basic call-context management, and the second one needs code annotation which is unrealistic for machine code.

Constraint solving. We choose to rely on a generic solving technique, namely constraint programming [1], rather than theory-specific algorithms. It is then easy to adapt new instructions while keeping reasonable performance. Constraint programming is mainly limited to theories over finite domains. Happily the bit-vector theory falls into this scope.

Considering a formula (or constraint) ϕ on a set of variables V in a *bounded* domain D, constraint programming is essentially a clever exploration of all partial valuations of V to find a solution to ϕ. Two main steps are interleaved and iterated until a solution is found (or the absence of solution is proved): *search* and *constraint propagation*. The search is a standard depth-first one with labelling and backtracking. At each step a variable is assigned a value from its domain. Once all variables are assigned, the valuation is checked against the formula. If it is not a solution, backtracking allows to make new choices. When neither labelling nor backtrack are possible, the formula is proved to be unsatisfiable. To avoid "blind" labelling as much as possible and speed up the search, constraint propagation mechanisms reduce variable domains at each step of the search through propagation rules. For example, consider the formula $y \leq x$ and supposes that the domain of y equals to the interval $[0, 1000]$ and that variable x has just been labelled with the value 42. Then the domain of y is reduced to $[0, 42]$ by propagation rules.

Constraint programming is a flexible paradigm to model and solve problems, and it is quite efficient at finding quickly a solution for "easy-to-solve" formulae, i.e. formulae having many solutions. However it can suffer from the so-called *"slow convergence phenomenon"* on "difficult-to-solve" formulae and inconsistent formulae. It is why we try to get rid of infeasible path predicates early in the test data generation algorithm rather than in the constraint solver.

We wrote a constraint solver for bit-vectors on top of an existing library for integer constraint programming developed in the model-based testing tool GATeL [13]. See section 4 for more implementation details.

3. IR recovery

We use an innovative combination of static and dynamic analysis to build an ACFG of the software. The static analysis does not need to be either complete or correct since the dynamic analysis will distinguish between valid jump targets and invalid ones. Hence the static analysis relies on light-weight techniques and its goal is to cheaply guide the dynamic analysis. The dynamic technique is based on slight modifications of the test data generation algorithm.

Algorithm 2 presents our technique. The static algorithm (STATICPROPAGATION) and the dynamic one (a modified version of GENTEST) are interleaved and iterated in the following manner. STATICPROPAGATION updates a map from dynamic jump instructions to potential address targets (`TargetCache`). The map itself is used as an entry of STATICPROPAGATION so that targets discovered in earlier calls to the procedure are not forgotten in later calls because of $\top$ spreading. Then the straightforward procedure BUILD creates an ACFG from the executable, the jump-to-target map and the entry-point of the file. Finally the test generation algorithm GENTEST is launched on the ACFG.

When a new target is discovered dynamically, the *exception* `newTarget` is thrown and caught by the top-level algorithm, the jump-to-target map is updated and the whole process is iterated starting on the new map. We describe STATICPROPAGATION and the modification of GENTEST below.

```
algorithm IR-RECOV(exec,iadd)
input: executable exec, initial address iadd
output: a test suite and the ACFG
 1: TargetCache ← ∅
 2: Loop
 3:    TargetCache ← STATICPROPAG(exec,iadd,TargetCache)
 4:    ACFG ← BUILD(exec,iadd,TargetCache)
 5:    try
 6:       return (GENTEST(ACFG.init_node),ACFG)
 7:    with exception
 8:       | newTarget (jump,taddr) →
 9:            TargetCache ← TargetCache ∪ {(jump,taddr)}
10:    end try
11: end loop
```

Algorithm 2: IR recovery mechanism

Static analysis. Our static analysis is mostly a standard constant propagation (over finite sets of constants rather than singleton) except that: (1) when abstract dynamic jump targets are not precise enough (i.e. evaluate to $\top$ in the abstract) we do not propagate values to all instructions ; (2) when abstract alias relationship are not precise enough we do not propagate values to all aliased memory cells.

Hence this static analysis does not compute a safe overapproximation of the program. In our context, missing targets is an issue because we may miss some paths of the program, but having too many false targets is also an issue because this will lead to many infeasible paths in the ACFG, and the test generation technique may suffer from slow-convergence phenomena. Since missing targets may be discovered dynamically, we adapt the static analysis to avoid the second case, at the price of incompleteness.

Dynamic analysis. The ACFG is also discovered on-the-fly. This requires modifying the **cgoto** case of the concolic test data generation algorithm. When a new target is discovered, an exception is thrown and caught by the IR recovery algorithm. The ACFG is updated accordingly and the test data generation algorithm continues. There are two reasons why a new target can be discovered: (1) it can be discovered by the concrete execution; (2) once all targets have been treated, an additional path predicate is computed constraining the target expression to take an undiscovered value. We present in Algorithm 3 on page 11 a precise description of the test data generation algorithm, including concolic execution and IR recovery mechanisms.

Correction and completeness. Our static analysis is neither complete (missing targets) nor correct (false targets). However, the data input generation algorithm ensures that false targets will not generate false tests, and it may discover missing targets dynamically. One could add an additional safe static analysis to detect whether or not all dynamic jumps are saturated, i.e. all their targets have been discovered. This is not yet done in our tool.

Discussion Purely static techniques for IR recovery are either too coarse or very sophisticated [4, 5] and difficult to implement for the non-expert because they aim at computing a both safe and tight overapproximation. In a dynamic perspective, completeness can be relaxed.

On the other hand, a purely dynamic discovery of the executable structure is feasible but suffers from two drawbacks. First, dynamic methods cannot ensure that all dynamic targets have been explored. Second, in constraint programming, equality constraints are more efficiently solved than disequality constraints. It is therefore more efficient to discover quickly possible targets and try to reach them by solving equality constraints than iteratively solving disequality constraints to discover targets.

4. The OSMOSE tool

Our results have been implemented in the OSMOSE tool. OSMOSE is an automatic binary-level analyser. It takes as input the executable (currently: Intel 8051, Motorola 6800 or PowerPC 550), the name of the hardware architecture and instruction set, a structural coverage objective (currently: instructions, branches or paths) and, optionally, a description of the environment. Outputs are mainly a high-level representation of the software under analysis, a set of test data and a report stating the bugs encountered, the coverage achieved by the test suite and unreached branches or instructions. The user view is described in figure 1.

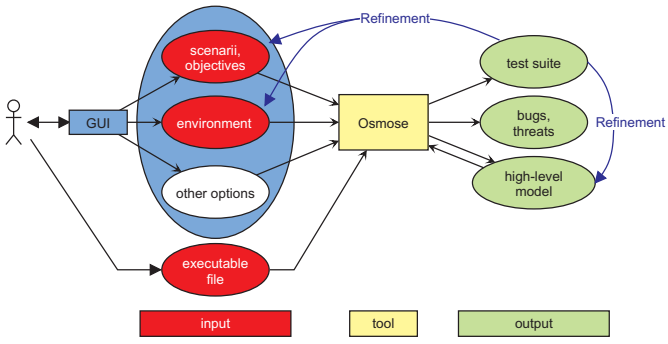

Figure 1: User view of OSMOSE

Generic machine code and simulation. To ensure the independence of the tool from any specific hardware ar-

chitecture and machine code, we work on a generic machine code parametrised by a generic architecture description. The concrete machine code is first translated into the generic machine code by a translation module. All analyses are performed on the generic machine code and one needs only to write a specialised translation module to integrate a new architecture. The processors currently supported by OSMOSE are listed in table 1.

Processor	Manufacturer	Date	Generation
6800	Motorola	1975	8 bits
8051	Intel	1980	8 bits
PowerPC 550	Freescale	1997	32 bits

Table 1. Processors supported by OSMOSE

The generic machine code implies that OSMOSE runs tests in simulation mode rather than in exact mode like other structural test tools. This is mandatory unless OSMOSE can be run on the exact architecture targeted by the executable under test, which is unrealistic for most processors.

Input/output. Inputs are the executable, the hardware architecture name and a description of the environment. Outputs are mainly the set of test data with the coverage measurement and a ACFG of the software. The interface is currently textual.

Environment. The environment is modelled by defining some memory cells as volatile, meaning that they correspond to inputs and can be modified randomly at any step of execution. Algorithms of sections 2 and 3 are modified to handle read-operations on volatile memory cells. They return the "top" value in the static analysis, a random value in the concrete execution and a new variable in the symbolic execution. In the presence of an environment, a test data is composed of a valuation of input values and a sequence of read values for each volatile memory cell.

Parameters. The tool provides command-line options and a configuration file to set different parameters of the analysis algorithms. For example the depth of the path-search (test data generation), the time-out value (constraint solver) or the size of the abstract domain (constant propagation). A particular programming effort has been devoted to extracting from the code as many parameters as possible, so that the user can easily set up their values.

Which guarantees? While reported bugs are guaranteed to be real bugs in the simulation mode of OSMOSE (validation on the concrete hardware architecture may be required), the ACFG and the coverage measure are approximations. If all dynamic jumps are saturated then the ACFG is a safe over-approximation (because of false targets) of the real CFG and the coverage measure is an under-approximation which the user can rely on. In the other case, the ACFG may contain both false and missing targets due to unsaturated dynamic jumps. The ACFG cannot be used as a safe over-approximation anymore. The ACFG still provides interesting information to the user, but the coverage measure is not faithful anymore. A solution is to add a standard safe static analysis to detect whether or not all dynamic jumps are saturated and report it to the user. This is not done yet.

Architecture. The software architecture is presented in Figure 2. The tool engine uses two external modules: a translation module from dedicated machine code to our generic one and a bit-vector theory constraint solver.

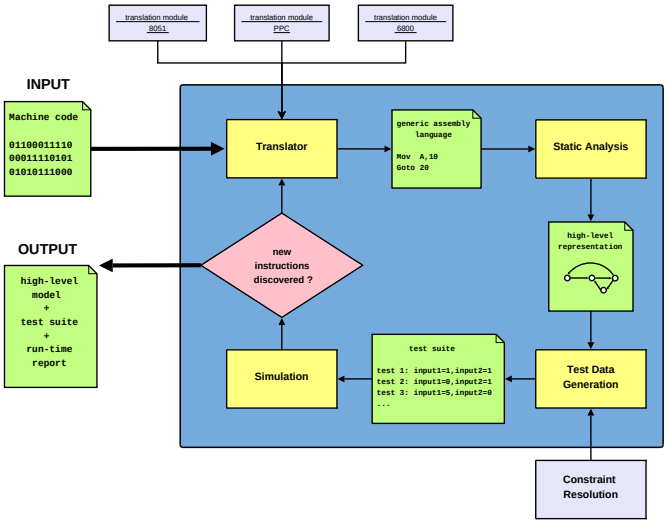

Figure 2: Software architecture of OSMOSE

Implementation. OSMOSE is written in OCaml, a functional language with strong static typing and high-level features like functors (parametrised modules) which have proven very useful for the generic software architecture. The constraint resolution engine is built upon the bounded arithmetic solver developed for the model-based testing tool GATeL [13]. We wrote a layer implementing the bit-vector theory on top of it. GATeL and our extension are written in the constraint logic programming system ECLiPSe [2]. The resolution engine is plugged into the OCaml source code using the C language as an intermediate. The program contains 22 kloc of OCaml (6kloc more for the three translation modules), 3.5 kloc of ECLiPSe and 1.5 kloc of glue in C. OSMOSE has been tested on an Intel PC running Linux.

5. Experiments

We evaluate OSMOSE on a set of small C programs compiled to the 8051 and to the PowerPC 550. These experiments do not intend to prove that OSMOSE is able to generate test data for real-life programs, but rather to demonstrate the feasibility of the ideas exposed in this paper.

We consider six different programs. `msquare` (40 loc) reads a volatile square matrix and check if the matrix is magic or not. The number of constraints grows exponentially with the size of the square matrix. `hysteresis` (30 loc) simulates a finite-state machine reading growing inputs until a maximal threshold is reached, then decreasing inputs until a minimal threshold is reached, and so on. The rate of variation is bounded. This example needs an environment and long sequence of tests (about 30 input readings and 250 branch conditions). `merge` (60 loc) is the well-known sorting algorithm. The program contains functions, vectors and aliases. `cell` (20 loc) is a small but tricky example (alias constraints) taken from [10]. `triangle` (20 loc) is a standard academic puzzle. `list` (20 loc) is a small example manipulating linked lists.

We used the following cross-compilers: `sdcc` for the 8051 and `gcc` for the PowerPC. We turned off optimisations to avoid too many modifications of the program. Noticeably, unoptimised executables appear to be more difficult to analyse than optimised ones. It is also worth noting that on `merge` and `triangle`, the two executables are very different. The `sdcc` compiler tends to add many function calls and bit-wise operations, especially in the presence of C pointers.

Results. Evaluations have been performed on a PC equipped with an Intel Pentium M 2Ghz and 1.2 GBytes of RAM, running Linux Ubuntu 6.10. The time-out for the solver was set up to 1 minute. Results are summarised in Table 2 for the 8051 and Table 3 for the PowerPC 550. For each C program, we report statistics about the executable (number of instructions and branch conditions), the branch coverage achieved and the computation time (in seconds). This coverage is exact since there are only easy-to-solve dynamic jumps in these programs. When two coverages are mentioned, the first one is reported by OSMOSE and the second one is evaluated w.r.t. feasible branches only. Memory consumption is not reported since it was very low, always smaller than 10 MBytes.

Comments. The tool performs well on almost all examples, with a computation time often smaller than 10 seconds and a 100% coverage of feasible branches on all examples but one. Noticeably, performances are almost always similar for the two processors while the size of variable domains grows from 2^8 for the 8051 to 2^{32} for the PowerPC. This is surprising since a main issue of constraint programming is the scalability w.r.t. the domain size. An explanation may be that most path predicates are solved with small values.

name	I	C	Branch cover	Time
msquare 3×3	272	23	82%-100%	5.2
msquare 4×4	274	23	86%-100%	129
hysteresis	91	8	100%	35
merge	56	12	100%	110
triangle	102	19	52%-100%	0.5
cell	23	4	100%	2.4
list	13	3	100%	1.2

I: #instructions, C: #conditional branches, Time in seconds

Table 2. Experiments for 8051 (8 bits)

name	I	C	Branch cover	Time
msquare 3×3	226	15	93%-100%	3.8
msquare 4×4	226	15	??	≥ 300
hysteresis	76	8	100%	36
merge	188	8	100%	2.5
triangle	40	9	100%	0.7
cell	18	4	100%	0.4
list	15	3	100%	1.1

I: #instructions, C: #conditional branches, Time in seconds

Table 3. Experiments for PowerPC (32 bits)

OSMOSE performs badly on two examples: `merge` for 8051 and `msquare` 4×4 for PowerPC. In both cases, the compilation step adds many bit-wise operations which are not efficiently handled by our constraint solver.

6. Related work

We are not aware of any other technique specialised in test data generation at the binary level, with the executable as the only input. However, a few tools work on low-level code with additional high-level information. There are also tools linked to different aspects of our work, mainly test data generation and IR recovery.

Low-level code verification. Some verification tools may be thought of as working at low level. However they are different from OSMOSE since they have access to high-level information from the program source code or the compiler, like the symbol table, targets of dynamic jumps for `switch`-like instructions and so on. In this category, we can cite jMoped and the tools from the Absint company. jMoped [7, 19] is a verification tool for Java programs working mainly at the bytecode level. While former versions aimed at full verification, the last one is devoted to test data generation. The core technology is based on BDD model-checking of weighted pushdown systems. Note that Java bytecode is rather high-level compared to usual machine

code. Tools from the Absint company [21] work on assembly languages with information from the C program. Their products are actually geared towards non-functional properties like estimation of maximal stack height or worst-case execution time. Their core technology is based on static analysis.

IR recovery (on executables). Commercial tools like IDA Pro [22] have problems with dynamic jumps. Specific static analysis techniques have been developed recently for this issue [4, 5]. Since the goal is to compute statically a safe and tight over-approximation, the technology is very sophisticated. The same team has also developed a verification technology based on model-checking the recovered abstract model [17] but we are not aware of any practical experiments and evaluation. These IR recovery techniques are difficult to implement for the non-expert because they target both completeness and tightness of approximations.

Since we consider the problem from a testing perspective, we can relax the completeness requirement. Moreover, thanks to our combination of static and dynamic steps, we can also relax the correctness requirement on the static step. This greatly simplifies the implementation of the static part, while correctness is (easily) ensured by the dynamic step.

Path-based structural test data generation. Our test data generation technology is inspired by tools like DART [10], CUTE [20] and PATHCRAWLER [23]. They work at the programming language level (C for all three and also Java for CUTE). All of these three tools rely on path predicate solving, bounded depth-first search and concolic execution. Premises of concolic execution can be found in PATHCRAWLER to find a feasible initial path and discover the CFG on-the-fly, while the current concept has been explicitly introduced and popularised by DART and CUTE. Each of these tools has its own specific features. DART and PATHCRAWLER have specific techniques to handle functions in a modular way [11, 14]. CUTE provides an hybrid test generation algorithm mixing both structural generation and random generation, which is proved to enhance the achieved coverage and the bug detection abilities [16].

Compared to OSMOSE, these tools work on a structured language and do not have to face the IR recovery problem. Considering only the test data generation technique, there are three other main differences.

- *Machine arithmetic*. Both DART and CUTE work on linear arithmetic (with simplex-based solvers and approximations of non-linear constraints) and PATHCRAWLER works on bounded full arithmetic (with constraint programming). OSMOSE is in a sense more complete than these tools since non-linear constraints are not approximated and low-level mechanisms of machine arithmetic like overflows are taken into account. This increase in precision has a cost but it seems imperative to discover typical security flaws.

- *Alias*. CUTE and PATHCRAWLER take advantage of the C program under verification to discover syntactic potential alias relationships, typically through type declarations and pointer expressions in branch conditions. However they cannot detect alias relationships depending only on the memory layout. On the contrary, OSMOSE does not have access to any high-level information but the concolic execution is modified to discover on-the-fly some alias relationships depending on the memory layout.

- *Concolic execution.* The concolic execution has been first used to find a feasible initial path and explore the program on-the-fly [23], and then to approximate "difficult instructions" [10, 20] like non-linear constraints and library function calls. We enhance the concolic execution with a third semi-concrete execution used to detect infeasible paths early and prune the path search. We also take advantage of the concrete execution to discover alias relationships and dynamic jump targets.

Constraint-based structural test data generation. InKa [8, 9] performs structural test data generation on C programs through constraint solving. In this approach the whole program is translated into an equivalent constraint programming problem, while the techniques presented so far translate only one path a time for efficiency issues. Noticeably, InKa includes a solver for floating-point arithmetic constraints [3].

7. Conclusion and future works

Verification at the binary level is much more difficult than higher-level analysis mainly due to the absence of any exact control-flow graph. However, this machine-code analysis may be the most relevant one in case of strong security requirements or even the only option left when no higher-level documentation is available. We have shown in this paper how to perform path-based structural test data generation on a standalone executable. We adapt existing technologies to the specific issues appearing in binary-level analysis, and we also develop innovative techniques, for example to solve the IR recovery problem. The results have been implemented in a tool named OSMOSE and encouraging experiments have been conducted.

This work is just a preliminary step demonstrating the viability of automatic structural test data generation on standalone executables. There are at least three directions for future work. First, we need to improve our test data generation technology to scale up to real-life programs. There are

different possibilities, from modular generation [11, 14] to hybrid generation [16] or dedicated constraint solving techniques. Second, we aim at improving the user interface of the tool to allow more interaction. No verification tool can claim to be completely automatic and user guidance has proven to be useful. Finally, safe (static) IR recovery techniques [4, 5] could give assurance about the quality of the abstract control-flow graph by detecting unsaturated jumps.

Detailed test data generation algorithm

Our test data generation algorithm is presented in section A, including concolic execution and dynamic IR recovery. Some parts are still omitted like alias management.

References

[1] K. R. Apt. Principles of Constraint Programming. Cambridge University Press, 2003.

[2] K. R. Apt and M. Wallace. Constraint Logic Programming using Eclipse. Cambridge University Press, 2007.

[3] B. Botella, A. Gotlieb and C. Michel. Symbolic execution of floating-point computations. In *STVR* vol. 16, 2006.

[4] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC 2004*. Springer.

[5] G. Balakrishnan, T. Reps, D. Melski and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *IFIP Working Conference on Verified Software: Theories, Tools, Experiments*. 2005.

[6] D. Cyrluk, O. Möller and H. Rueß. An Efficient Decision Procedure for the Theory of Fixed-Sized Bit-Vectors. In *CAV 1997*, Springer.

[7] J. Esparza and S. Schwoon. A BDD-based Model Checker for Recursive Programs. In *CAV 2001*, Springer.

[8] A. Gotlieb, B. Botella and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *ISSTA 1998*. ACM.

[9] A. Gotlieb, B. Botella and M. Watel. Inka: Ten years after the first ideas. In *ICSSEA 2006*.

[10] P. Godefroid, N. Klarlund and K. Sen. DART: Directed Automated Random Testing. In *PLDI'2005*. ACM.

[11] P. Godefroid. Compositional dynamic test generation. In *POPL 2007*. ACM.

[12] M. Howard. Some bad news and some good news. Microsoft Developer Network, October 2002, `http://msdn2.microsoft.com/en-us/library/ms972826.aspx`.

[13] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions: GATeL. In *ASE 2000*. IEEE.

[14] P. Mouy. Automatisation du test de tous-les-chemins en présence d'appels de fonctions. PhD thesis, INSTN, 2007.

[15] Software Considerations in Airborne Systems and Equipment Certification. *RTCA 1992*.

[16] R. Majumdar and K. Sen. Hybrid Concolic Testing. In *ICSE 2007*. IEEE.

[17] T. Reps, S. Schwoon, S. Jha and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. In SCP, october 2005.

[18] A. Stump, C. W. Barret, D. Dill and J. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *LICS 2001*, IEEE.

[19] D. Suwimonteerabuth, F. Berger, S. Schwoon and J. Esparza. jMoped: A Test Environment for Java programs. In *CAV 2007*, Springer.

[20] K. Sen, D. Marinov and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE 2005*. ACM.

[21] `http://www.absint.com/`

[22] `http://www.datarescue.com/`

[23] N. Williams, B. Marre and P. Mouy. On-the-Fly Generation of K-Path Tests for C Functions. In *ASE 2004*. IEEE.

A Detailed test data generation algorithm

Our test data generation algorithm is presented in Algo. 3, including concolic execution and dynamic IR recovery. Some parts are still omitted like alias management.

algorithm GENTEST2(node_init)
input : initial node node_init
parameter : atomic, node.addr, node.next, mem_init
output: set of test data Res

1: Res $\leftarrow \emptyset$
2: REC(node_init, $\top$,mem_init,mem_init)
3: **return** Res

procedure REC(node, Φ, C, S)
input : node, formula Φ, concrete state C, semi-concrete state S
parameter : Res, atomic, node.addr, node.next, update, eval, addr.node
exception: newTarget(node,addr)
output: no result, the procedure updates Res

1: **if** (termination or Φ unsat or depth_bound) **then return** ();
2: **else**
3: **Case** node **of**
4: | $\varepsilon \rightarrow$
5: **try** $S_p \leftarrow solve(\Phi)$; Res $\leftarrow$ Res $\cup \{S_p\}$
6: **with** unsat or time_out $\rightarrow$ ();
7: **end try**
8: | block instr $\rightarrow$
9: REC(node.next,$\Phi \wedge$ atomic(instr),update(C,instr),update(S,instr))
10: | goto tnode $\rightarrow$ REC(tnode,Φ,C,S)
11: | call fnode $\rightarrow$ REC(fnode,Φ,C,S)
12: | ite(cond,inode,tnode) $\rightarrow$
13: **case** eval(cond,S) **of**
14: | true $\rightarrow$ REC(inode,Φ,C,S); /* *constant value* */
15: | false $\rightarrow$ REC(tnode,Φ,C,S); /* *constant value* */
16: | symbolic $\rightarrow$ /* *non-constant value* */
17: REC(inode, $\Phi \wedge$ cond,C,S);
18: REC(tnode, $\Phi \wedge \neg$cond,C,S)
19: **end case**
20: | cgoto expr $\rightarrow$
21: **if** eval(expr,C) $\notin$ node.next **then** /* *new target discovered* */
22: **exception** newTarget(node,eval(expr,C));
23: **else**
24: **case** eval(expr,S) **of**
25: | constant addr $\rightarrow$ REC(addr.node,Φ,C,S) /* *constant value* */
26: | symbolic $\rightarrow$ /* *non-constant value* */
27: **for all** tnode $\in$ node.next **do**
28: REC(tnode, $\Phi \wedge$ expr $=$ tnode.addr,C,S)
29: **end for** /* *the following line forces to discover new target* */
30: REC(tnode, $\Phi \wedge \bigwedge_{t \in \text{node.next}}$ expr $\neq$ t.addr,C,S)
31: **end case**
32: **end if**
33: **end case**
34: **end if**

Algorithm 3: Detailed test data generation algorithm

B Details on experiments

The `msquare` program

```
#define taille 4
volatile char entry[taille][taille];

int main() {
  char sum = 0;

  char matrix[taille][taille];
  char sumcol[taille];
  char sumlig[taille];
  char sumdiag[2];
  char i,j,magic_number, success;

  for(i=0 ; i < taille ; ++i)
    for(j= 0 ; j < taille ; ++j)
      matrix[i][j] = entry[i][j];

  for(i=0 ; i < taille ; i++) {
    sumlig[i] = 0;
    sumcol[i] = 0;
  }

  for(i=0 ; i < taille ; i++)
    for(j=0 ; j < taille ; j++)
      sumlig[i] += matrix[i][j];

  for(i=0 ; i < taille ; i++)
    for(j=0 ; j < taille ; j++)
      sumcol[i] += matrix[j][i];

  sumdiag[0] = 0;
  sumdiag[1] = 0;
  for(i=0 ; i < taille ; ++i) {
    sumdiag[0] += matrix[i][i];
    sumdiag[1] += matrix[i][taille-i-1];
  }

  for(i=0 ; i < taille*taille ; ++i)
    for(j=i+1; j < taille*taille ; ++j)
      if (matrix[i/taille][i%taille] ==
          matrix[j/taille][j%taille]
         )
        goto end;

  magic_number = sumdiag[0];

  if (sumdiag[1] != magic_number)
    goto end;

  for(i=0; (i < taille) ; ++i)
    if (
        (sumcol[i] != magic_number) ||
        (sumlig[i] != magic_number)
       )
      goto end;

  success = 1;
  return 1;

end:
  success = 0;
  return 0;
}
```

The `hysteresis` program

```
volatile int portIn;
volatile int minSeuil, maxSeuil;

int getPort(void) {
  return portIn;
}

typedef enum { Down, Up} tstate;

int main() {
  int readVal,predVal = 0;
  tstate currentState = Down;
  tstate predState = Down;

  while(1) {
    readVal = getPort();
    if ((readVal>predVal+10)||(predVal>readVal+10))
       goto bad_spec;
    if ((currentState==Down)&&(readVal>maxSeuil))
       currentState = Up;
    else if ((currentState==Up)&&(readVal<minSeuil))
       currentState = Down;
    if ((predState==Up)&&(currentState==Down))
       goto end;
    predState = currentState;
    predVal = readVal;
  }

 end:
  return 1;

 bad_spec:
  return -1;
}
```

The `triangle` program

```
volatile unsigned char ain,bin,cin;
typedef enum { NORMAL,EQUI,ISO } ttriangle;
typedef unsigned char uchar;

void swap(uchar *a, uchar *b) {
  unsigned char tmp = *a;
  *a = *b;
  *b = tmp;
}

ttriangle getType(uchar a,uchar b,uchar c){
  ttriangle ret = NORMAL;
  if (a < b) swap(&a,&b);
  if (a < c) swap(&a,&c);
  if (b < c) swap(&b,&c);
  if (a == b)
    if (b == c) ret = EQUI;
    else ret = ISO;
  else  if (b == c) ret = ISO;
  return ret;
}

int main() {
  if ((ain == 0) && (bin == 0) && (cin == 0))
    return -1;
  return getType(ain,bin,cin);
}
```

The `merge` program

```
#include<iostream>
#define SIZE 4

volatile int tab_in[SIZE];

void merge(int tab_to_merge[],int limit,int size){
  int tab_copy[SIZE];
  int i, tmp1, tmp2;
  int *ptr1, *ptr2, *plimit, *psize, *pdest;
  ptr1 = tab_to_merge;
  plimit = tab_to_merge + limit;
  ptr2 = plimit;
  psize = tab_to_merge + size;
  pdest = tab_copy;

  while((ptr1 < plimit) && (ptr2 < psize)) {
    tmp1 = *ptr1;
    tmp2 = *ptr2;
    if (tmp1 > tmp2) {
      *pdest = tmp2;
      ptr2++;
    } else {
      *pdest = tmp1;
      ptr1++;
    }
    pdest++;
  }
  while(ptr1 < plimit) {
    *pdest = *ptr1;
    ptr1++;
    pdest++;
  }

  while(ptr2 < psize) {
    *pdest = *ptr2;
    ptr2++;
    pdest++;
  }

  for(i = 0 ;  i < size ; ++i)
    tab_to_merge[i] = tab_copy[i];
}

void mergesort(int tab_to_sort[], int size) {
  int ihalf,i;
  if (size <= 1) return;
  ihalf = size /2;
  mergesort(tab_to_sort, ihalf);
  mergesort(tab_to_sort + ihalf, size-ihalf);
  merge(tab_to_sort, ihalf, size);
}

int main() {
  int i;
  int tab[SIZE];
  for(i = 0 ; i < SIZE ; ++i) tab[i] = tab_in[i];
  mergesort(tab,SIZE);
  return 0;
}
```

The `cell` program

```
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int g(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
  if (x > 0)
    if (p != 0)
      if (g(x) == p->v)
if (p->next == p)
  while(1) {}
  return 0;
}

cell *pcellinit;
int xinit;

int main(){
  testme(pcellinit,xinit);
  return 1;
}
```

The `list` program

```
#define GOAL_LENGTH 10

struct mylist {
  struct mylist* next;
};

volatile struct mylist pentrylist;

int main() {
  struct mylist* pcurrent = &pentrylist;
  int i,success;

  for(i = 0 ; i < GOAL_LENGTH ; ++i) {
    if (pcurrent == 0) goto end;

    pcurrent = pcurrent-> next;
  }

  if (pcurrent != 0) goto end;
  else { success = 1 ; while(1) {} }


 end:
  success = 0;
  return 0;
}
```