OSMOSE: Automatic Structural Testing of Executables[‡]

Sébastien Bardin[*,1], Philippe Herrmann[1]

[1] *CEA LIST, Software Safety Laboratory,*
Point Courrier 94, Gif-sur-Yvette, F-91191 France
`first.name@cea.fr`

SUMMARY

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. The paper shows 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. Results have been implemented in a tool named OSMOSE and encouraging experiments have been conducted. **Copyright © 2009 John Wiley & Sons, Ltd.**

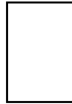
Received 21/08/2008; Revised xxx

KEY WORDS: machine code analysis ; automatic testing ; IR recovery ; concolic execution

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) for structured languages such as C or Java, but rarely at the machine-code level. Actually, binary-level analysis is considered more difficult than other analyses, while being redundant with them. However, machine code analysis is relevant in at least three situations: when no high-level source code is available, when the compiling process cannot be trusted 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 simply lost

[*]Correspondence to: CEA LIST, SOL/LSL, Gif-sur-Yvette, F-91191 France. E-mail:sebastien.bardin@cea.fr
[‡]This article is an extended version of results presented at ICST 2008 [9].
Contract/grant sponsor: Work partially funded by EDF, the *Software Factory/MoDriVal* project of the French cluster SYSTEM@TIC PARIS-REGION and the *Arpège/Bincoa* project of Agence Nationale de la Recherche (ANR).



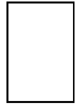
(legacy code). Standard source code analysis relies on the assumption that the compiler preserves the program semantics. While it is realistic for standard reachability properties and standard compilation techniques, it cannot be trusted anymore in the case of strong safety/security requirements and highly sophisticated optimisations [15]. Moreover, it is not uncommon that exotic processors, typically found in embedded systems, come with buggy (commercial) compilers. Finally, certain kinds of properties require to analyse the program as close as possible to the concrete behaviour, e.g. quality of service (QoS) properties like worst-case execution time and maximal stack-height estimation.

Major difficulties of binary-level analysis. Machine code analysis is different from higher-level analysis (languages or models) because of three specific difficulties: IR recovery, low-level data manipulation and low-level control manipulation. (1) The first problem is the so-called Intermediate Representation recovery (IR recovery): since an executable is nothing more than a sequence of bits, no basic control-flow information is available (such as functions, loops or variables), while it is given for free in higher-level analysis. Actually even the exact number of instructions in the program is unknown since instructions may have different sizes, instructions can overlap, there is no syntactic difference between instructions and data and finally the exact set of targets of dynamic jumps cannot be statically determined (a jump is dynamic when its destination is only known at run-time). (2) *Low-level data manipulation.* Low-level operations on data have to be taken into account precisely while for higher-level analysis, coarser abstractions are usually sufficient. The most obvious one is machine arithmetic. 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. (3) *Low-level control manipulation.* In high-level language, control-oriented instructions are clearly separated from data-oriented instructions. Moreover, calls to sub-routine are encapsulated within a clean functional abstraction (bindings of arguments, local context, return to the caller). None of these good programming patterns are present in machine code. Control-oriented instructions are just assignments of specific registers and function calls are nothing more than jumps and push / pop of arguments. Finally there is a great diversity of hardware architectures and instruction sets (ISA), differing both in terms of physical memory layout and instruction sets.

Bit-vector theory. The bit-vector theory (see for example the book of Kroening and Strichman [32] for an overview) formalises standard machine instructions at the bit level. Formulae are interpreted over vectors of bits of a fixed length. Instructions include modulo arithmetic with signed and unsigned views of bit-vectors, logical bitwise operations and other low-level instructions such as shift, extraction and concatenation. Floating-point arithmetic is usually not considered though it can be encoded. Note that in the rest of the paper, only the quantifier-free fragment of bit-vector theory is considered. Satisfiability in bit-vector theory is decidable since the interpretation domain is finite, but complexity is high, even for the quantifier-free fragment since SAT can be easily encoded into it.

The OSMOSE *tool.* OSMOSE aims at performing automatic test data generation on standalone executable files. The test selection is white box since no other information than the executable itself is considered available, with coverage-based stop criteria. The OSMOSE tool targets 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

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls



read on each sensor. The user has to provide a description of the environment, declaring volatile memory cells. The two main outputs of the tool are an abstraction of the control-flow graph of the program (ACFG) and a test suite (test data plus exercised execution path) with an approximation of the coverage measure. Test data are exact, in the sense that at run-time the program launched on these data follows exactly the expected execution paths. On the other hand, the ACFG is an approximation of the ideal CFG of the program, neither complete nor correct: the ACFG can both report false instructions / branches and miss legal instructions / branches.

The issue of the oracle (*"does the test pass or fail?"*) is not addressed by OSMOSE. However, the test suite can be exported to an external automated oracle if available, for example in approaches such as back-to-back testing or parametrised unit testing [48]. Moreover, the tool can still find *"intrinsic"* bugs, i.e. executions which are undoubtedly faulty independently of any specification, such as division by zero, jump to an incorrect instruction and violation of programmer-defined assertions.

OSMOSE can be used for standard correctness testing activities (on executable files), as well as for assistance in executable behaviour comprehension. However, since the ACFG and the coverage measure are just indicative, the tool alone cannot currently be used to generate test suites achieving a certain level of structural coverage, as is required in safety critical systems like aeronautics. An *ad hoc* solution (implemented in OSMOSE) is to let the user specify overapproximations of targets of dynamic jumps. An interesting alternative would be to connect OSMOSE to one of the very few tools performing safe ACFG reconstruction [33].

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 Generic Assembly Language (GAL). The tool currently handles three processors: the Intel 8051, the Motorola 6800 and the more recent Freescale PowerPC 550.

Technologies. Binary-level analysers must first build a high-level model of the software under investigation. Then verification techniques may be used. The test data generation technology is white box, based on symbolic execution [16, 24, 23, 29, 45, 47, 51]: a path predicate is computed from a control path, solving this predicate leads to a test data exercising the path. Path predicates are expressed in the quantifier-free fragment of bit-vector theory, so a constraint solver for this theory is required. It turns out that the OSMOSE tool is organised around four basic technologies: structural test data generation, bit-vector constraint solving, IR recovery and Generic Assembly Language.

A major improvement of symbolic execution is the concept of concolic execution [23, 45], also referred to as mixed execution [16] or dynamic symbolic execution [47]. 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 CFG [23, 45, 51], or to approximate complex instructions like non-linear constraints or library function calls [23, 45]. Note that concolic execution can be seen as a mix between pure constraint-based test data generation approaches [29] and search-based test data generation techniques [25, 26, 30, 31]. As such, it shares common ideas with the dynamic symbolic execution approaches developped in the 1990s [39].

The test generation method described above relies on solving path predicates. While test data generation tools from high-level descriptions [23, 37, 45, 51] are usually based on integer constraints (classically bounded arithmetic [37, 51] or linear arithmetic [23, 45]), constraints are here expressed in

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

the bit-vector theory. The OSMOSE solver is based on the Constraint Programming paradigm [2, 42], which is flexible enough to encode all "exotic" instructions one can find in instruction sets.

The IR recovery mechanism combines in an innovative way both static and dynamic analyses. First, a (global) static analysis creates a coarse model. Then if new parts of the program are discovered during the test data generation phase (by the concrete execution or the symbolic execution) the high-level model is updated and the (global) static analysis is re-launched. The static analysis does not need to be very precise since a complementary analysis is performed via concolic execution, avoiding difficulties inherent to purely static IR recovery techniques [14, 15, 33, 34].

To ensure the independence of the tool from any specific hardware architecture and machine code, all analysis are performed on a generic assembly language (GAL) parametrised by a generic architecture description. Native machine code is first decoded into native assembly language, then translated into GAL. All analyses are performed on the GAL description of the program, since every GAL instruction comes with three (parametrised) semantics: concrete semantics for emulation / simulation, logical semantics for symbolic execution (test data generation) and abstract semantics for static analysis (IR recovery). Benefits are twofold. First, integrating a new instruction set into OSMOSE requires only to specify the architecture, to provide a decoder for the native instruction set and to translate each native instruction into a semantically equivalent sequence of GAL instructions. Other semantics are derived automatically from the concrete one. Second, integrating a new semantics (e.g. for a new analysis) for all supported ISA requires to define it only for about thirty instructions.

These technologies are organised as shown in figure 1. The machine code is partially decoded and translated into the internal generic assembly language (IR recovery, GAL), then the test data generation algorithm is launched (concolic execution, bit-vector solving). During this phase, new instructions can be discovered. The decoder is launched again, the IR is updated and test data generation is applied using an enriched CFG.

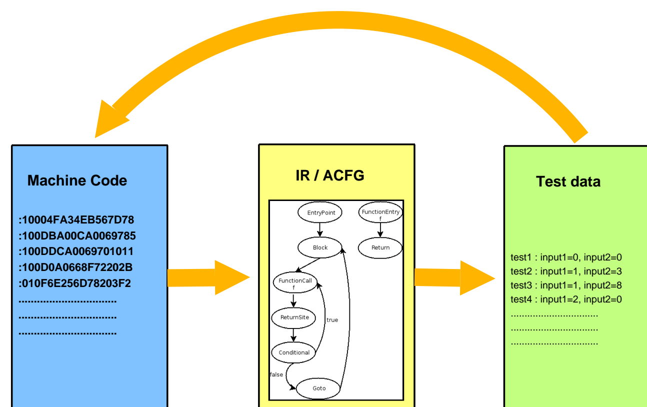
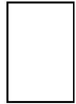


Figure 1: OSMOSE workflow

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr



Limitations. Floating-point arithmetic and interrupts are not handled, like in almost all verification technologies. Moreover, self-modifying code cannot be taken into account. However, it can be detected and signalled. It is worth noticing that many others (if not all) "nasty" low-level control mechanisms such as dynamic jumps and modification of the return address through stack overflow can be taken into account, as well as instruction overlapping and recursive functions.

Contributions. This paper addresses the problem of designing an efficient tool for the automatic analysis (testing and IR recovery) of executables. There are four main contributions in this work.

- The specific issues of machine code analysis (compared to structured languages) are identified, and the paper shows how to adapt two existing frameworks for test data generation of structured programs (namely, concolic execution and Constraint Programming) to machine code, pinpointing the main difficulties. This work proposes also a very innovative solution to the IR recovery issue in a test data generation framework (i.e. where completeness of the recovery can be relaxed), involving a combination of static, symbolic and concrete approaches. It turns out that concolic execution simplifies dramatically the IR recovery problem.
- The paper presents also an innovative software architecture, based on a Generic Assembly Language parametrised by an architecture template, to add easily new instruction sets in an analysis framework. This work has been done independently of the one by Lim and Reps [36]. It was already developed, but only sketched, in the previous OSMOSE conference article [9].
- Finally, it is shown how Constraint Programming can be used in a solver for the quantifier-free fragment of the bit-vector theory. The OSMOSE solver manages all bit-vector constraints generated by the GAL language. It is built on top of a bounded integer CP solver. This is a very different approach from usual bit-vector solvers, rather based on a bit-sequence view of variables and SAT/SMT solving [6, 22].
- These results have been implemented in a structural test data generation tool for executables. First experiments demonstrate the feasibility of the approach. The tool is largely architecture-independent and can currently handle three different architectures and machine codes (8051, 6800, PowerPC 550).

Related issues. Machine code analysis may seem close to low-level C program analysis and Java bytecode analysis. This paragraph pinpoints similarities and differences. The analysis of low-level C programs, typically found in embedded systems or operating systems, is indeed very close to machine code analysis: low-level manipulations on data and control are possible, either directly (arbitrary type casting, bitwise operations, pointers on functions, `longjump/setjump` instructions), or indirectly (embedded assembly language, stack overflow to modify return address). However, it must be clear that currently, most automatic analysis tools for C programs consider a "clean" subset of the language, usually excluding all the previous nasty mechanisms. This nice subset can be treated as a structured language, and is really far from low-level C programs. On the opposite, Java bytecode analysis has almost nothing in common with machine code analysis. Java bytecode has been designed with safety concerns in mind: many error-prone features are forbidden and some interesting facts can be verified statically. For example: strong static typing (boolean, integer, reference), validity of dynamic jumps, local context and call-return policy.

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Outline. The remaining part of the paper is organised as follows. Section 2 gives an overview of architectures and instruction sets. The next four sections describe OSMOSE core technologies: Generic Assembly Language in section 3, test data generation in section 4, IR recovery in section 5 and bit-vector solving in section 6. Section 7 presents the OSMOSE tool and its implementation. Section 8 describes some experiments with the tool. Finally section 9 discusses related work and section 10 concludes and gives directions for future work.

2. A taste of machine code

This section provides a simple introduction on machine code, architecture and instruction set. This is a very limited overview, but it should be sufficient to understand the remaining part of the paper. More complete presentation can be found in standard books [46].

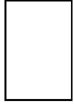
A machine code for a given processor is a language directly understandable by this processor. This program is a sequence of instructions encoded in binary format. A typical instruction has the form: `opcode arg1 arg2 ... argn`, where `opcode` is the encoding of a basic command of the processor (`add`, `move`, etc.). Decoding a single instruction is straightforward since an opcode is never the prefix of another opcode, and all the relevant information (e.g. number of arguments and their sizes) depends only on the opcode (possibly with additional prefix / postfix information). Once an instruction has been decoded and executed, the processor searches for the next instruction to execute. The next instruction is usually located just after the end of the current one, but not always (e.g. `jump`). In almost every case, the potential successors of the current instruction are known statically. The only exception is the case of dynamic jumps, i.e. `jump` whose operand depends on the execution, like `jump R` with `R` a register. Dynamic jumps are the main reason for the IR recovery being so problematic.

Even if there are many different hardware architectures, they all share common ideas from the Von Neumann architecture. Basically, a processor can be seen as an automaton extended with a small number of variables (registers), each one containing exactly a word-size long sequence of bits (shortly, *a m-word*). In addition to these registers, an additional memory (typically RAM) allows to store a huge amount of information. The memory can be seen as a very large array, storing a m-word in each cell. The memory is conceptually divided in two parts: information local to function calls (caller address, local values) is stored in the *(call) stack*, and global / persistent information is stored in the *heap*. But usually there is no physical separation between the stack and the heap. Two registers play a special role: the `PC` register *(program counter)* and the `SP` register *(stack pointer)*. Stack pointer indicates the address (in memory) of the top of the stack. Program counter indicates the current instruction. At the end of an instruction, `PC` is updated with the next instruction address and control jumps to it. In most instruction sets, `PC` can be modified *via* dedicated instructions only.

A processor comes with a finite set of instructions. Most instructions can be seen as a sequence of affectations of the form $\text{lhs} \leftarrow f(\text{rhs}_1, \ldots, \text{rhs}_n)$. Control instructions include jumps, calls and returns. Jumps can be either static: the jump target is known statically (e.g. `goto 100`); or dynamic: the jump target is computed at run-time (e.g. `goto R0`). Jumps can also be conditional. Calls and returns to functions are only convenient shortcuts enforcing the way the call stack is used. The typical effect of a `call` instruction is to store the current address on top of the stack, to increase the stack pointer and to jump to the callee address. The typical effect of a `return` instruction is to retrieve the value on top of the stack, which should be the caller address, to decrease the stack pointer and to

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr



jump to the caller address. This is the general picture. Depending on the architecture, the stack pointer may be incremented or decremented, and this modification may appear before or after retrieving the return address from the stack. Instruction sets usually provide also a large range of data instructions. Most common ones include: data transfer from / to memory, (machine) integer arithmetic, floating-point arithmetic and other bit-vector operations. Noticeably, integer arithmetic is performed *modulo* the value of the largest representable integer, and integer operations update various flags (predefined memory location), typically to record the occurrence of overflows.

3. Generic Assembly Language (GAL)

To ensure the independence of OSMOSE from any specific hardware architecture and instruction set, all analyses are performed on a Generic Assembly Language (GAL) parametrised by an architecture description. Native machine code is first decoded into native assembly language, and then translated into a GAL program. All analyses are performed at the GAL level, since every GAL instruction comes with three (parametrised) semantics: concrete semantics for emulation/simulation, logical semantics for symbolic execution (test data generation) and abstract semantics for static analysis (IR recovery). Figure 2 gives a flavour of this framework. Benefits of such an architecture are twofold. On the one hand, additional cost for integrating a new instruction set into OSMOSE boils down mainly to specifying the architecture and translating each native instruction into a semantically equivalent sequence of GAL instructions. The instruction set is then fully supported by all analysers in OSMOSE (emulation, symbolic execution, static analysis), the symbolic semantics and the abstract semantics being deduced from the concrete one. On the other hand, adding a new semantic (for a new kind of analysis) requires to define it for only about thirty basic instructions.

Abstract architecture and GAL program. An abstract architecture X is defined by a pair $X = \langle \mathcal{R}, \mathcal{M} \rangle$ where $\mathcal{R}$ is a finite set of variables ranging over m-words and $\mathcal{M}$ is a finite set of (disjoint) arrays of m-words, called *memory regions* (shortly, m-regions). Each m-region is indexed by non-negative integer values, called addresses. A *memory location* (m-loc) is defined as being either a register or the cell of a m-region. The size of m-words stored in each m-loc is fixed a priori for each m-loc. Intuitively, variables represent registers and m-regions represent different physical memories, for example RAM and ROM. A GAL program P is a tuple $P = \langle \mathcal{R}, \mathcal{M}, A, I \rangle$ where $\langle \mathcal{R}, \mathcal{M} \rangle$ is an abstract architecture, A is the set of addresses of valid instructions and I is a map from A to GAL instructions. GAL instructions are composed of more primitive micro-instructions. Indeed, a GAL instruction is a sequence of data (micro-) instructions followed by a control (micro-) instruction. Data instructions (**assign**) are multi-affectations. Basic control instructions are static jumps (**goto**) and dynamic jumps (**cgoto**). Moreover, these jumps can be combined with a conditional statement (**ite**). Note that instruction **goto** is a special case of **cgoto**, however they are distinguished from each other in GAL since they are managed in very different ways by automatic analysers. GAL instructions obey the following grammar:

GAL instr	::=	(data)* control
control	::=	jump | **ite** (cond,jump,jump)
jump	::=	**goto** addr | **cgoto** rhs
data	::=	**assign** $(\text{lhs}_1, \ldots, \text{lhs}_n) \leftarrow f(\text{rhs}_1, \ldots, \text{rhs}_n)$

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

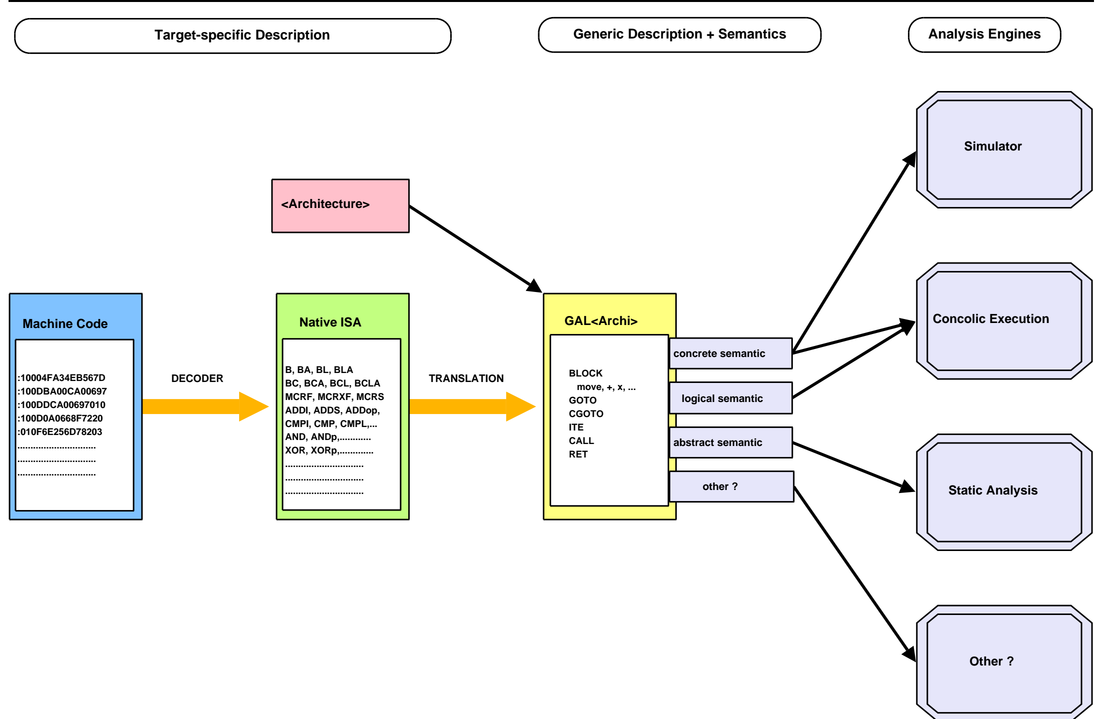


Figure 2: Generic Architecture with GAL

Left-hand side and right-handside operands are defined by

rhs ::= bv | r | m[rhs] | restrict(rhs,i,j)

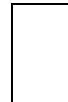
lhs ::= Ignore | r | m[rhs]

where bv is a m-word, $r \in \mathcal{R}$ is a register, $m \in \mathcal{M}$ is a m-region, *restrict(rhs,i,j)* denotes the extraction of the sub-bitvector of rhs from bit i to bit j (i and j are integer constants) and *Ignore* is a dummy operand, used when a multi-affectation does not need all results from an operation (typically flags).

Operators available in multi-affectations include usual arithmetic operators (signed and unsigned versions), usual bitwise operators, usual bit-vector manipulations such as shifts and rotations and a C-like `compare` operator.

Semantics. Considering a GAL program $P = \langle \mathcal{R}, \mathcal{M}, A, I \rangle$, a *configuration* c of P is either a pair $c = (addr, val)$ where $addr$ is an address in A and val is a valuation of each m-loc, i.e. a map from m-locs to m-words; or the special configuration $error$. The operational semantic is given in an imperative way. Undefined values, e.g. division by zero or out-of-bound memory access, are modelled by the special value $\bot$. Thus, partial functions from bit-vectors to bit-vectors are turned into total functions from bit-vectors to the union set of bit-vectors and $\bot$. Given a configuration $(addr, val)$, its successor configuation $succ$ is defined by:

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls



- Let $I(addr) = d_1 \ldots d_n\ ctrl$, where d_i's are data instructions and $ctrl$ is a control instruction.
- Let val'' be the valuation obtained from val after having performed the sequential multi-affectations $d_1, \ldots, d_n$; if an exception occurs during multi-affectations then $val' = \bot$ else $val' = val''$;
- Let $addr'$ be equal to $eval(ctrl, val')$ where $eval$ is defined recursively by:
  - $eval(goto\ k, v)$ is equal to k,
  - $eval(cgoto\ rhs, v)$ is equal to the evaluation of rhs over v (may evaluate to $\bot$),
  - $eval(ite(cond, jump1, jump2), v)$ is equal to $eval(jump1, v)$ if $cond$ holds true over v, it is equal to $eval(jump2, v)$ if $cond$ holds false, and it is equal to $\bot$ if $cond$ evaluates to $\bot$.
- If $val' \neq \bot$ and $addr' \neq \bot$ and $addr' \in A$ then $succ = (addr', val')$, else $succ = error$.

Note that the $error$ configuration itself has no successor.

Encoding an ISA into GAL. A typical ISA is encoded into GAL in the following manner. A variable is associated to each ISA register and a m-region is associated to each physical memory (in most architectures, one m-region for the RAM is sufficient). Some auxiliary variables may be useful for complex instructions requiring a micro-code level modelling. The program counter variable can be removed, since its effect is captured by GAL control micro-instructions. Most ISA instructions are directly translated in a sequence of two GAL micro-instruction: one for data and one for control. Side-effects such as flag updating are modelled with multi-affectations. One can notice that there is no **call** or **ret** instructions. They are modelled as multi-affectations followed by a (dynamic) jump.

Adding a new architecture/ISA in OSMOSE. Adding a new architecture and instruction set to OSMOSE requires three different artifacts: a decoder for the instruction set, an instantiated architecture and a translation from the instruction set into GAL.

- *Provide a decoder for the instruction set.* Given an address in the program source code, the decoder returns the corresponding instruction (in the native instruction set) or fails if the instruction is not defined. Implementing such decoders is a tedious task, but not difficult. Some decoders are publicly available.
- *Fill the architecture template.* It is required to define the size of m-words, available registers and memory regions. Moreover, one must specify which m-loc are writable, which m-loc can contain program instructions (to detect self-modifying code), and which m-loc represent the program counter and the stack pointer.
- *Translate ISA into GAL.* It is the more demanding task. It requires mostly to translate every basic native operand (like a flag) into an m-loc, to translate each native instruction into an equivalent sequence of GAL instructions and to specify the size of each instruction.

Discussion. The modelling presented so far does not take into account that the program code itself is stored somewhere in the memory, allowing in some cases the program execution to modify the program code. Thus self-modifying code cannot be modelled. However, most other "nasty" low-level control mechanisms can be taken into account: dynamic jumps, stack overflow, modification of return

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

address and even overlapping of instructions and run-time decryption of instructions as long as GAL instructions can be added incrementally into the program map. Considering the GAL language, the main limitations is that during the translation process, a native instruction can only be translated into a pure sequence: no loop is allowed. It may cause some problems for instructions involving loops in their micro-code. Finally, note that there is a strict separation between those GAL micro-instructions which affect control and those GAL micro-instructions which affect data. Hence, it is not possible to model an ISA instruction which would or not affect the program counter, depending on the value of its operands (imagine an architecture where the PC value is stored in RAM, and an instruction like `ram[x] := y`). However, this is currently a very theoretic limitation, since no common architecture provides such a feature.

4. Test data generation

4.1. Basic ideas

The OSMOSE test data generation technology follows the idea of symbolic execution. This is an old idea [29], but automatic tools implementing this idea for software have blossomed recently [16, 23, 24, 45, 47, 51]. The central notion of path predicate is introduced first.

Definition 1 (Path predicate) *Given a program P of input domain D and π a path of P, a path predicate of π is a formula φ_π on D such that if $V \models \varphi_\pi$ then execution of P on V follows the path π.*

The two main ideas behind symbolic execution are that: (1) a solution to a path predicate φ_π for a given program P is actually a test data exercising path π, with potential applications in structural testing; and (2) a path predicate φ_π for a path π can be computed by keeping track of logical relations among variables along the execution, rather than just their concrete values. Figure 3 shows how a symbolic execution is performed on a path of a small program.

Loc	Instruction	Symbolic exec
0	`input(y,z)`	new vars Y_0, Z_0
1	`y++`	$Y_1 = Y_0 + 1$
2	`x := y + 3`	$X_2 = Y_1 + 3$
3	`if (x < 2 * z)` *(branch True)*	$X_2 < 2 \times Z_0$
4	`if (x < z)` *(branch False)*	$X_2 \geq Z_0$

Path predicate for path 0→1→2→(3,T)→(4,F)
$Y_1 = Y_0 + 1 \wedge X_2 = Y_1 + 3 \wedge X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$

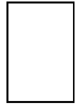
Path predicate projected on input
$Z_0 - 4 \leq Y_0 < 2 \times Z_0 - 4$

Figure 3: Symbolic execution along a path

The basic procedure for test data generation by symbolic execution consists in choosing a path, computing and solving its path predicate, recording the solution (if any) as a test datum, and iterate until a termination criterion is reached. The termination criterion is typically structural, like path,

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr



instruction or branch coverage. Compared to previous work where no global representation of the program is required [23, 45, 51], the technique developed here works on an (abstract) control-flow graph (ACFG). The ACFG allows to use static techniques for IR recovery and path pruning [11]. ACFG nodes can be either sequences of multi-affectations (**assign**), conditional statements (**ite**), static jumps (**goto**) and dynamic jumps (**cgoto**). For the sake of simplicity, in this section the ACFG is considered precomputed once and for all before the test data generation is launched. Hence, each **cgoto** instruction comes with a predefined set of possible successors. 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. Section 5 discusses how to build the ACFG incrementally.

Algorithm 1 presents the basic idea of OSMOSE test data generation algorithm. The procedure uses a bounded depth-first traversal of the control-flow graph to enumerate all paths in a recursive manner. This is a standard strategy [23, 45, 51] 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. Choice points in a machine code program are conditionals and dynamic jumps. For conditional, the procedure just forces the search to take the *"then"* or *"else"* branch by adding to the current path predicate Φ the condition or its negation. In the case of dynamic jumps, the procedure explores each possible target by constraining the argument of the jump (usually an arithmetic expression over registers) to take each possible known value in turn. Basic instructions are translated into formulae by the procedure `atomic`. The external procedure `solve` returns a solution of a constraint or the `unsat` exception in case of unsatisfiability. The procedure is presented 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.

4.2. Concolic execution

Concolic execution [23, 45] is a recent major improvement to symbolic execution. Basically, a concolic execution is a concrete execution and a symbolic execution running in parallel, the concrete one collecting relevant information along the execution path to help the symbolic execution. Concrete information is used to perform approximation when the symbolic execution encounters instructions which are either impossible to model in the given path predicate theory, or whose modelling leads to constraints too costly to solve. A typical mechanism is the *concretization* of a variable: at some point of the symbolic execution, a variable is forced to be equal to its current concrete value over the concrete execution, limiting the symbolic resolution to all paths going through this concrete value. Concrete execution can be used for example to follow feasible paths only [23, 45, 51], to approximate non-linear arithmetic constraints, to deal with library function calls and multiple levels of pointer dereferencement [23, 45]. Concolic execution is a recent approach, however it has been quickly recognised as very promising and many different recent works are based on this approach [9, 12, 16, 24, 47]. The main advantages of concolic execution compared to other purely static techniques (and especially symbolic execution) are twofolds: concolic execution is very robust against "difficult-to-analyse" instructions or programming features: it can always concretize it rather than just ignore it or stop the analysis; concolic execution offers a control on the trade-off between performances and completeness of the resolution, depending on which constraints are concretized.

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

```
algorithm GENTEST1(node_init)
input : initial node node_init
output: set of test data Res
 1: Res ← ∅
 2: REC(node_init, ⊤)
 3: return Res
procedure REC(node, Φ)
input : node, path predicate Φ
output: no result, update Res
 1: Case node of
 2: | ε →        /* end node */
 3:       try S_p ← SOLVE(Φ) ; Res ← Res ∪ {S_p}
 4:       with unsat → ();
 5:       end try
 6: | assign affect-list → REC(node.next, Φ ∧ ATOMIC(affect-list))
 7: | goto tnode → REC(tnode, Φ)
 8: | ite(cond,inode,tnode) →
 9:       REC(inode, Φ ∧ cond);
10:       REC(tnode, Φ ∧ ¬cond)  /* branching */
11: | cgoto expr →
12:       for all tnode ∈ node.next do
13:          REC(tnode, Φ ∧ expr = tnode.addr)     /* branching */
14:       end for
15: end case
```

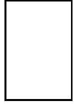
Algorithm 1: Basic test data generation algorithm

Note that concolic execution can be seen as a mix between pure constraint-based test data generation approaches [29] and search-based test data generation techniques [25, 26, 30, 31]. As such, it shares common ideas with dynamic symbolic execution from the 1990s [39].

The OSMOSE test data generation technology follows the concolic principle. 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 5). A simplified view of the concolic test data generation procedure is presented in Algorithm 2. The main procedure REC now takes two different inputs: the current path predicate Φ and the current concrete memory state C. Concrete memory states come with basic functions for update and condition evaluation. Initially, C is the map filled with 0 (denoted 0). Branching is more difficult to handle in the concolic case than in the symbolic case. Indeed, Φ and C must always be consistent with the path prefix. While this property is easily ensured for Φ, it is not the case for the concrete memory state: at each branching instruction, C is consistent with only one of the successors. The symbolic execution algorithm is modified in the following way to ensure consistency at each step between C and the path prefix. When a branching instruction is encountered, first the conditional is evaluated w.r.t. C and the concolic execution is launched along that path (the search follows the concrete path). Then on backtracking, the current path predicate augmented with the new branching condition is solved *immediately*. If there is no solution, the branch is infeasible and the search backtracks. Otherwise, the solution can be used to derive a new concrete memory state C' consistent with both the current path prefix and the desired new branch. Deriving C' is straightforward if the path predicate contains all intermediate variables (like in OSMOSE). If the path predicate contains only entry variables, it is always possible to relaunch a concrete execution from these entries and get the concrete memory state at the instruction point where

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr



it is needed. In Algorithm 2, subprocedure `update_C_for_branching` takes as input a solution of a path predicate and outputs a new concrete memory state C' consistent with the corresponding path prefix.

```
algorithm GENTEST2(node_init)
input : initial node node_init
output: set of test data Res
 1: Res ← ∅
 2: REC(node_init, ⊤, 0)
 3: return Res
procedure REC(node, Φ, C)
input : node, formula Φ, concrete state C
output: no result, the procedure updates Res
 1: Case node of
 2: | ε →
 3:       try S_p ← solve(Φ) ; Res ← Res ∪ {S_p}
 4:       with unsat → ();
 5:       end try
 6: | assign affect-list → REC(node.next,Φ ∧ atomic(affect-list),update(C,affect-list))
 7: | goto tnode → REC(tnode,Φ, C)
 8: | ite(cond,inode,tnode) →
 9:       case eval(cond,C) of
10:       | true → /* concrete execution follows the if branch */
11:          REC(inode, Φ ∧ cond, C);
12:          try /* find C' compatible with the else branch */
13:             S_p ← solve(Φ ∧ ¬cond);
14:             C' ← update_C_for_branching(S_p);
15:             REC(tnode, Φ ∧ ¬cond, C');
16:          with unsat → ();
17:          end try
18:       | false → ....... /* symmetric of the true case */
19:       end case
20: | cgoto expr →
21:       tnode ← (eval(expr,C)).node /* concrete successor */
22:       REC(tnode, Φ ∧ expr = tnode.addr, C)
23:          for all nd ∈ node.next - {tnode} do
24:             try /* find C' compatible with node.addr */
25:                S_p ← solve Φ ∧ expr=nd.addr;
26:                C' ← update_C_for_branching(S_p);
27:                REC(nd, Φ ∧ expr = nd.addr, C' )
28:             with unsat → ();
29:             end try
30:          end for
31: end case
```

Algorithm 2: Concolic test data generation algorithm

4.3. Advanced concerns

This subsection describes advanced concerns about the symbolic/concolic execution algorithm. For the sake of simplicity, these features have been omitted in Algorithm 2.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

IR Recovery. The test data generation algorithm and the IR recovery mechanism are deeply interwoven and the ACFG may be updated during the test generation step. The whole IR recovery mechanism is described in section 5.

Functions. Functions are inlined. Recursive functions are allowed since the bounded depth first search prevents the search 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. Recent papers [1, 27, 38] propose some solutions, but they focus mainly on purely functional procedures while procedures with side-effects are preeminent in machine code programs.

Alias. Two m-locs are said to be 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. It turns out that aliasing is a bit less difficult from a testing perspective than from a static one, since it is not required to compute a safe approximation of all possible alias relationships in order to generate relevant test data. The following solution is used: the concrete execution is analysed to extract the aliasing relationships existing in the concrete trace and add them to the path predicate. The good point is that the solution found (if any) is sure to follow the right execution path. The bad point is that this new predicate is stronger than required, and it may be infeasible while the path is actually feasible with another alias constraint. This technique allows to discover aliasing relationships depending only on the memory layout. This is orthogonal to the technique developed in CUTE [45] where possible syntactic alias relationships are extracted from the C program, mainly from type declarations and alias expressions in branch conditions.

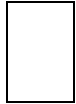
Optimisations. Concolic execution engines can be improved in two orthogonal ways, either reducing the number of path prefixes to explore (path pruning) or reducing the cost of each call to the solver (formula simplification). Path pruning techniques implemented in OSMOSE [11] include discarding path prefixes which cannot reach new instructions or branches and preventing the search from backtracking in deep nested calls. Formula simplifications include the slicing of instructions which do not affect control expressions along the path, as well as splitting the path predicate into independant subformulas solved separately. Moreover, OSMOSE procedure is also enhanced with a “semi-concrete” execution dynamically detecting constant values at each step of the execution, allowing to prune the path search by detecting on-the-fly trivial cases of infeasible paths. This can be seen as a combination of formula simplification through constant propagation and incremental lightweight solving.

Formula simplification techniques such as formula splitting and constant propagation are rather common in concolic execution tools [16, 24, 45, 47], some of them performing even incremental solving [51]. Path pruning techniques are more original. However, smart heuristics geared towards faster coverage [16, 24, 47] and modular or lazy test data generation [1, 27, 38] allow also to reduce the search space.

5. IR recovery

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr



5.1. Motivations

Purely static techniques for IR recovery are either too coarse or very sophisticated [14, 15] and difficult to implement for the non-expert because they aim at computing a both safe and tight overapproximation. In a testing perspective, completeness can be relaxed and the analysis is much easier. Actually, once the problem is relaxed, a purely dynamic discovery of the executable structure is feasible. However, the dynamic approach suffers from three drawbacks. First, dynamic methods cannot ensure that all dynamic targets have been explored, while in some cases even simple static analyses do. While such a safe IR recovery is not mandatory for common testing practices, it is absolutely imperative for the validation of critical systems. Second, concolic execution relies on very precise and potentially expensive theorem proving techniques, while simple static analyses (e.g. constant propagation) are cheap. Third, some recent path-pruning optimisations for concolic execution require a global view of the program structure [11]. This section describes an innovative combination of static analysis and concolic execution to solve the (weak) IR recovery issue typically arising in test data generation.

5.2. Existing solutions

There exist *ad hoc* static techniques to address the IR recovery problem, mainly the brute force method and naive static discovery. In the brute force approach, one tries to decode an instruction at every byte (or word) of program. The resulting set of instructions is a safe upper-approximation, however it does not recover legal transitions between instructions and reports too many false instructions. The naive static approach is a recursive traversal starting from the initial instruction. Decoding goes on if next instructions are known statically and stops otherwise. This technique is useful unless dynamic jumps or violations of call-return policy are encountered. Theses approaches can be improved in a few ways. First if the header (part of the program added by the compiler/linker) is available, entry points of most functions may be known, which may allow to recover parts of the program "hidden" by a dynamic jump. Second, when the compilation toolset used to create the executable is known, some dynamic jumps may be resolved by a syntactic analysis of the machine code to detect standard compilation patterns. However, these pieces of information are not always available (e.g. the header may have been stripped off because of strict memory limitations), they cannot be completely trusted (the header may have been forged) and anyway they are not sufficient to solve the general problem. Interestingly, recent researches have been conducted to develop both safe and tight IR recovery mechanisms based on advanced static analysis [14, 15, 33, 34], but they are difficult to implement for the non-expert since they interleave many different analyses.

5.3. IR recovery in OSMOSE

Test data generation techniques like concolic execution do not require a perfect IR recovery: (1) the recovered ACFG may contain illegal instructions since the test data generation engine do not consider infeasible paths; (2) the recovered ACFG may miss some legal instructions: it will not affect the relevance of test data generated by the procedure, and test is incomplete in essence. Hence the IR recovery mechanism can be both incorrect and incomplete. But on the other hand, the more precise the ACFG is, the more efficient and relevant the test data generation will be, since the procedure will not

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

waste time on infeasible paths and generated tests will exercise the program more thoroughly. Since the static analysis does not need to be either complete or correct, it relies on lightweight techniques and its goal is to cheaply guide the concolic procedure for IR recovery. The concolic-based recovery requires only slight modifications of the OSMOSE test data generation algorithm. Algorithm 3 presents the big picture of this approach. The static algorithm (STATICPROPAGATION) and the concolic one 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 remembered. 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 by the concolic procedure, an 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. The procedure STATICPROPAGATION and the modification of GENTEST2 are described 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 ← STATICPROPAGATION(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
```

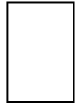
Algorithm 3: IR recovery mechanism

Static analysis. The static analysis is mostly a standard constant propagation (over finite sets of constants rather than singleton) except that when abstract dynamic jump targets are not precise enough (i.e. evaluate to the “don’t know” abstract value), their values are not propagated to all possible instructions (i.e. the analysis does not try to decode every address in the executable file). Hence this static analysis does not compute a safe over-approximation of the program. In an automatic testing context, missing targets is an issue because it may lead to missing 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 data generation procedure may waste a lot of time trying to explore them. Since missing targets may be discovered dynamically, the static analysis part of the IR recovery mechanism is adapted to avoid the second case, at the price of incompleteness.

Concolic execution. The ACFG is also discovered during concolic execution. It requires to modify the **cgoto** case of Algorithm 4. There are two reasons why a new target could be discovered: (1) it can be discovered by the concrete execution; (2) for each dynamic jump, once all targets have been treated, an

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr



additional path predicate is computed to constrain the target expression to take an undiscovered value. If it succeeds, the solution leads to a new target. Algorithm 4 presents the modification of Algorithm 2 to take into account IR recovery mechanisms. Each time a new target is discovered in the concolic execution procedure, an exception is thrown and caught by Algorithm 3.

```
/* only modifications of algorithm 2 are shown */
procedure REC(node, Φ, C)

 1: Case node of
 2: ...............
 3: | cgoto expr →
 4:     if eval(expr,C) ∉ node.next then    /* new target */
 5:       exception newTarget(node,eval(expr,C));
 6:     else
 7:       ...............
 8:       for all n ∈ node.next - {tnode} do
 9:         ...............
10:       end for /* the following lines try to discover new target */
11:       try
12:         S_p ← solve(Φ ∧ ⋀_{t∈node.next} expr ≠ t.addr) ;
13:         C' ← update_C_for_branching(S_p);
14:         exception newTarget(node, eval(expr,C')); /* new target */
15:       with unsat → ();
16:       end try
17:     end if
18: ...............
```

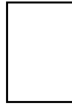
Algorithm 4: IR recovery via concolic execution

Correction and completeness. The concolic-based IR recovery mechanism is correct, in the sense that it can only find legal targets. However, it is obviously not complete because of missing targets. The static analysis-based IR recovery mechanism is neither complete (missing targets) nor correct (false targets). Thus the ACFG is an approximation of the real CFG. However, generated test data are still correct (in the sense that they follow the intended path at execution), since they are generated *in fine* by the concolic execution procedure, which ensures that false targets will not generate false test data. Note that it can be checked easily whether the static analysis is complete or not: it is the case when all addresses represented by abstract expressions attached to dynamic jumps are decoded. In that case, the ACFG is a safe overapproximation of the ideal CFG of the program, and coverage measurement can be safely trusted (i.e. it is an underapproximation of the truly achieved coverage). However, this ideal case is not expected to happen often considering the simple analysis carried out.

Discussion. Even though safe IR recovery is not mandatory for common testing practices, it is absolutely imperative for the validation of critical systems. In that case, a safe recovery can be obtained by systematically decoding all abstract targets [33, 34]. However, in this setting the simple static analysis presented so far would probably recover too many false targets, and more advanced static analyses for IR recovery should probably be used [14, 15, 34].

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr



6. Bit-level Constraint solving

The concolic procedure requires a solver for the quantifier-free fragment of bit-vector theory. This solver relies on a generic solving technique, namely Constraint Programming [2, 42], 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, which is the case for the bit-vector theory.

6.1. Bit-vector theory

The bit-vector theory (see for example the book of Kroening and Strichman [32] for an overview) formalises standard machine instructions at the bit level. Formulae are interpreted over vectors of bits of a fixed length. Instructions include modulo arithmetic, logical bitwise operations and other low-level instructions such as shift, rotation or concatenation. For example, the following formula falls into the scope of bit-vector theory:

$$((x + 100) >> 2) \leq b :: c$$

where $>>$ and :: denote respectively logical right-shift and concatenation operators. 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.

6.2. Principles of constraint programming

Considering a formula (or constraint) ϕ on a set of variables V in a *bounded* domain D, Constraint Programming (CP) 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 backtracking 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 (or *propagators*), removing values of the domain of a variable which are not involved in any solution of the constraint. For example, here are the two propagators for constraint $x < y$ on variables x and y, with interval-based domains D_x, D_y defined by $D_x = [l_x, L_x]$ and $D_y = [l_y, L_y]$:

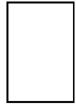
(Propagator 1) if $L_x \geq L_y$ then $D_x = [l_x, L_y - 1]$

(Propagator 2) if $l_y \leq l_x$ then $D_y = [l_x + 1, L_y]$

The following example shows how labelling and propagation are interleaved. Consider the formula $x < y$ and suppose that domains D_x and D_y of x and y are respectively the intervals $[20, 1000]$ and $[0, 900]$. First, by propagation rules, domains are reduced to $D_x = [20, 899]$ and $D_y = [21, 900]$. No more reduction can be obtained, so a labelling step is performed. Suppose that y is labelled with the value $42 \in D_y$. The domain of x is then reduced by propagation to $D_x = [20, 41]$. Finally, suppose that x is labelled with value $23 \in D_x$. A solution to the constraint $x < y$ has been found.

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr



Constraint Programming is a flexible paradigm to model and solve problems on finite-domain theories. It can encode easily any reasonable constraint on finite domains, and it is quite efficient at finding quickly a solution for "easy-to-solve" formulae, i.e. formulae having many solutions. Moreover, one can control the trade-off between efficiency of resolution and implementation effort by designing more or less complex propagators. However it can suffer from the so-called *"slow convergence phenomenon"* on "difficult-to-solve" formulae and inconsistent formulae. For example, try the search-propagation paradigm on constraint $x < y \wedge x > y \wedge D_x = [0..1000] \wedge D_y = [0..1000]$. The OSMOSE solver makes use of such traditional local propagators in addition with more global propagators aimed at detecting early some specific kinds of inconsistencies and avoid as much as possible slow convergence phenomena [19].

Constraint Programming is a paradigm rather than a technology defined once and for all, and it has a number of degrees of freedom. Thus, implementing a CP-based solver requires to instantiate at least the following parameters: (1) representation of the domain of variables; (2) search algorithm (kind of traversal, choice of the next variable to be labelled and next value to be assigned, backtracking procedure); (3) propagation rules; and (4) mechanisms against slow convergences, typically detecting early certain causes of unsatisfiability.

6.3. OSMOSE solver

The OSMOSE constraint solver for bit-vectors is written on top of an existing CP-based solver for bounded integer constraints developed for the model-based testing tool GATEL [37]. The philosophy behind the solver is the following: bit-vectors are mostly manipulated as integers, relying on the arithmetic solver as much as possible. Low-level constraints are encoded into integer constraints when possible. Otherwise, resolution is mostly delayed until enough information is obtained on the operands. Yet, a few constraints identified as bottlenecks during experimentation have been optimised for specific cases appearing in case-studies.

Bounded arithmetic solver. This paragraph describes briefly the main characteristics of the CP-based bounded arithmetic solver of GATEL. Representation for domain variables is based on intervals plus congruence [35], allowing to express for example: $x \in [0, 10000] \wedge x \equiv 0$ modulo 1024. The search algorithm is depth-first with chronological (standard) backtracking. The next variable to be labelled is chosen according to the most constrained input variable, and the next value to be assigned is the minimal value in the domain. Finally, the solver incorporates a specific mechanism against slow-convergence: in addition to the local propagators presented so far, the solver maintains a global view of all difference constraints appearing in the problem (i.e. inequalities of the form $x - y \leq k$, with $k \in \mathbb{Z}$) and performs standard satisfiability checking on this subset. For example, the slow convergence phenomenon pointed out early in this section can be discovered by this mechanism. A similar technique is described by Feydy, Schutt and Stuckey [19].

The bit-vector layer. As previously stated, bit-vectors are considered as their (unsigned) integer representation and the solver relies on standard arithmetic as much as possible. Three kind of translations from bit-vectors to arithmetic are used: direct translation, delayed translation and a mixture of both. Moreover, some specific optimisations are implemented.

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

Direct translation to standard arithmetic: some constraints on bit-vectors can be directly encoded into arithmetic constraints without any loss in precision, using operations like integer division and remainder to extract specific portions of a bit-vector. Such constraints include for example: extraction, concatenation, rotation, logical and arithmetic shifts, bitwise not, modulo arithmetic and flags management. Here are two examples of such direct translations. The constraint `concat(NA,A,NB,B,NR,R)`, meaning that `R` (of length `NR`) is the concatenation of `A` (size `NA`) and `B` (size `NB`), is equivalent to: $R = 2^{NB} \times A + B$. The constraint `add(N,A,B,R,FC)`, meaning that `R` (of size `N`) is the result of the addition of `A` and `B` (both of size `N`) and that `C` is the resulting carry flag, can be encoded (with an additional variable) as: $\mathtt{C} = (A + B)/2^N \wedge \mathtt{R} = (A + B) \mod 2^N$.

Delayed translation to standard arithmetic. In case that a bit-vector constraint is too costly to translate directly into an arithmetic constraint (typically: bitwise `xor`), CP allows to delay the translation. Indeed, propagators can have the form: *"once two arguments of the constraints are instantiated, launch arithmetic constraints* $c_1, \ldots, c_K$*"*. In the worst case, when an efficient arithmetic translation does not exist or one does not have time to design it, it is even possible to wait for all input variables of the constraints to become instantiated, then compute directly output variables.

Mixture of direct and delayed translation to standard arithmetic. Finally, many constraints can be encoded into a mixture of both direct and delayed translation into arithmetic: a direct *approximated* translation into arithmetic allows for efficient (but overapproximated) propagators, while the delayed translation ensures correctness of the whole result. A typical example is the bitwise `and` and `or` constraints: `R = A and B` can be approximated with $R \leq A$ and $R \leq B$, however this translation is not exact and a delayed translation must be used in addition.

Optimisations for specific cases. It may be the case that a constraint has no efficient translation into arithmetic *in general*, while this constraint is always used in a very specific pattern that can be efficiently encoded into arithmetic. Consider the following example: a bitwise `and` constraint is used with one of its operands being constant. It is a quite common situation since it corresponds to a "mask" operation in the original program. In this case, the solver performs two optimisations: first it tries to recognise typical masks corresponding to extraction operation (e.g. x `and` 7 extracts the three lower bits of x), and whenever it succeeds, the solver uses the direct translation of the extraction constraint; second, if no common mask has been recognised, the longest sequence of 0 contained into the constant (starting from the least significant bit) is found, permitting to deduce a congruence on the result.

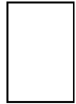
Discussion. The translation from bit-vector to arithmetic makes an intensive use of *division* and *mod* constraints. It may be costly, as non-linear constraints are not always efficiently handled in Constraint Programming. However, it seems to be reasonable considering the first experiments of section 8. An explanation may be that in most cases one input operand only is a variable (the other one being constant), and keeping track of congruences directly in the domain of variables deals very efficiently with multiplications / division by constant. As will be shown in section 8, specific optimisations for masks and bitwise operations are crucial on some examples.

7. The OSMOSE tool

Results described so far have been implemented in the OSMOSE tool. OSMOSE is an automatic machine code analyser. It takes as inputs the executable file, architecture and ISA identifiers, a structural

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr



coverage objective 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 bugs encountered, the coverage achieved by the test suite and unreached branches or instructions. The interface is currently textual.

Generic machine code and simulation. All analyses are performed on the GAL generic assembly language and one needs only to write a specialised translation module to integrate a new architecture. Processors currently supported by OSMOSE are: Motorola 6800 (8-bit), Intel 8051 (8-bit), Freescale PowerPC (32-bit). 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.

Environment. The environment is modelled by specifying some memory locations as volatile, meaning that their value can change non-deterministically at any step of execution (typically, because of data acquisition from a sensor). Algorithms of sections 4 and 5 are modified to handle read-operations on volatile memory locations. They return the abstract "don't know" value in the static analysis, a random value in the concrete execution and a free 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.

Coverage objectives. It is possible to declare the structural coverage objective to achieve. Objectives are defined by three parameters: the nature of items to be covered, the minimal coverage to achieve and the relevance of subfunction coverage. Items to be covered can be paths, instructions or branches (in this paper, an `ite` instruction counts for 1 instruction and 2 branches). The minimal coverage measurement to achieved is a value ranging from 0% to 100%. Finally, test data generation can be performed in a unit testing fashion (trying to cover only the target function) or in an integration testing fashion (trying to cover in addition all items of subfunctions).

Which guarantees? While test data are always exact, in the sense that the execution path is exactly the one predicted by OSMOSE, the ACFG and the coverage measure are approximations. Indeed, the ACFG may contain both false and missing targets. It still provides interesting information to the user, but the coverage measure is not safe (under-approximation) in most cases.

How to use it? OSMOSE is designed for correctness testing of executable files. Since the ACFG and the coverage measure are approximated, the tool alone cannot currently be used to generate test suites achieving a certain level of structural coverage, as is required in safety critical systems like aeronautics. If a safe coverage measure is mandatory or if the recovery is too coarse, the user can still manually provide an over-approximation of the sets of dynamic jump targets. However, it may be a cumbersome activity. An interesting alternative would be to connect OSMOSE to one of the very few tools performing (a truly) safe ACFG reconstruction [33].

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

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

developed for the model-based testing tool GATEL [37]. A layer implementing the bit-vector theory has been written on top of it. GATEL and the bit-vector layer are written in the Constraint Logic Programming system ECLIPSE [3]. 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 is compiled on an Intel PC running Linux.

Currently, the OSMOSE tool is not publicly available.

8. Experiments

This section presents various experiments carried out with OSMOSE. First, the IR recovery abilities of the tool are evaluated on handcrafted C programs, very small but representative of the kind of dynamic jumps that can occurs in embedded systems (function pointers and compilation of `switch` instructions). The commercial disassembler IDA Pro [50] is used as a witness. Second, the test data generation abilities of OSMOSE are evaluated on various examples, ranging from standard academic puzzles to small open source code and preliminary experiments on industrial code. The witness is random testing. All evaluations have been performed on an Intel Pentium M 2Ghz with 1.2 GBytes of RAM running Linux Ubuntu 6.10. The following cross-compilers have been used: `sdcc` for the 8051 and `gcc` for the PowerPC. Optimisations have been turned off to avoid too many modifications of the program. Unoptimised executables appear to be even more difficult to analyse than optimised ones.

8.1. IR recovery

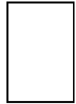
Description. Experiments are performed on four small C program compiled to PPC 550 machine code with `gcc`. These programs are rather representative of the reasons for which dynamic jumps may be introduced into an executable file through compilation from a C program. Moreover, most dynamic jumps found in embedded systems are expected to follow these schemes. The four examples work as follows: `fptr0` is a simple example, with only one function pointer assigned dynamically to a constant value; `fptr4` contains an array of function pointers; `switch0` contains a simple `switch` statement with 5 cases, translated by `gcc` into a dynamic jump; `switch-array` looks like `switch0` except that the operand of the `switch` statement involves an array expression. Note that all branches of the original C program are feasible.

OSMOSE is used with both the static and the dynamic recovery mechanisms, i.e. the tool provides an abstract control-flow graph and test data covering this graph. The tool is evaluated against IDA Pro [50], a commercial disassembler working with a combination of naive static propagation, pattern matching and brute force decoding. IDA Pro is also able to use the information stored in the header of the executable.

Results. Characteristics of the programs and results of IDA Pro and OSMOSE are summarised in Table I. For each example, the table provides the numbers of instructions (I), branches (Br), dynamic jumps (DJ) and their targets (T), as well as the percentage of recovered instructions (RI) and recovered dynamic branches (RDB), i.e. branches from a dynamic jump to one of its targets. Computation time and memory consumption are not reported since they were both very low.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



name	#I	#Br	#DJ (#T)	RI IDA	RDB IDA	RI OSMOSE	RDB OSMOSE
fptr0	8	1	1 (1)	100%	0%	100%	100%
fptr4	41	7	1 (3)	100%	0%	100%	100%
switch0	40	7	1 (5)	100%	0%	100%	100%
switch-array	141	23	1 (5)	100%	0%	100%	100%

I: instructions, Br: branches, DJ: dynamic jumps, T: targets of dynamic jumps
RI: recoverd instructions, RDB: recovered dynamic branches

Table I: Evaluation of the IR abilities of OSMOSE

Both tools recover all the instructions (without introducing any spurious instruction), but while IDA Pro is unable to recover any of the dynamic branches, OSMOSE find them all, without introducing any spurious branch. Thus, while in theory OSMOSE is neither correct nor complete, on these example it recovers each time the *exact* control-flow graph.

Comments. It may look surprising that IDA Pro does not find any dynamic branch while recovering all instructions. Actually, the executable files respect the ELF format which indicates the location of the code in the executable file. Using this information plus brute force decoding allow IDA Pro to retrieve all instructions. However, executable files with less additional information may cause serious problems to IDA Pro. Interestingly, OSMOSE was always faster than IDA Pro (by a factor of two), while it recovers more information and generates test data achieving a full branch coverage. Finally, OSMOSE launched with only the dynamic recovery was also able to recover all dynamic edges.

8.2. Test Data Generation

Experiments are performed on 23 programs written in C compiled to PPC executables or 8051 executables, for a total of 30 executable files. The C programs are divided into three classes: handcrafted programs, open source programs and embedded programs.

The handcrafted C programs are the following. `msquare` reads a 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` simulates a finite-state machine reading inputs slowly increasing 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 sequences of tests. `merge` is the well-known sorting algorithm. The program contains functions and aliases. `cell` is a small but tricky example [23]. `triangle` is the standard academic puzzle. `list` is a small example manipulating linked lists.

There are two small programs taken from industrial programs. `check-pressure` is a small 8051 program performing data acquisition and actuator activation when pressure is too high. `buf_Get` is part of a circular buffer module of an industrial network monitoring application. It is designed to extract a frame from a buffer and return a buffer status.

Open source functions include `strlen` and `strtok` from the standard C library (glibc version 2.4) and two functions from the open source program GNU Go (version 1.2), `countlib` and `findcolor`. `strtok` aims at splitting a string into tokens separated by delimiters. `strlen`

Copyright © 2009 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

computes the numbers of characters of a C-style string. Both these functions are called once in a main function: their return value is then tested against a specific value to increase the difficulty of the test. Function `findcolor` determines the color of a piece depending on the colors of neighbor, while `countlib` computes the liberty of a color piece at a given location.

Finally, functions `aircraft0` to `aircraft9` are taken from an embedded power-controller program for aircraft engines running on PowerPC 550, provided by Hispano-Suiza [28]. The source code is a low-level C program with embedded assembly language. Machine code counts around 30,000 instructions structured into 250 functions. Functions `aircraft0` to `aircraft4` have low call-depths (between 0 and 2) while functions `aircraft5` to `aircraft9` have higher call-depths (from 4 to 10).

Protocol. The objective is to cover all branches. Test data generation is performed in an integration testing way for all small examples and in a unit testing way for the largest ones, i.e. the tool does not try to cover branches of callees, and backtrack is allowed only when the call depth is less or equal to 2, 0 being the procedure to cover. OSMOSE is evaluated against a random test generator. This generator has been written on top of the simulation engine of OSMOSE. On each example, the random generator is asked to generate K test data, K being defined by $K = max(1000, T_O \times 20)$, where T_O is the number of test data generated by OSMOSE. Test data generation stops once full coverage is achieved.

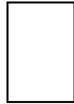
Results. Results are summarised in Table II. Statistics are reported for each executable (number of instructions and branches). The table reports also the branch coverage achieved and the computation time for both tools, and the number of tests generated by OSMOSE. The Mode column indicates whether experiments are performed in a unit testing approach (*unit*) or in an integration testing approach (*full*). In unit testing mode, the number of branches and instructions in the callees is also reported. `Time` is the computation time obtained with optimisations for bitwise operations discussed in section 6. Memory consumption is not reported since it was very low, always smaller than 10 MBytes. The correctness of test data returned by OSMOSE has been checked manually on small examples (against coverage and path information). Especially, on the `msquare` program, OSMOSE does return a test data corresponding to a magic square.

OSMOSE performs well on almost all examples, with a computation time often smaller than 1 minute (23 cases out of 30) and a 100% coverage on 21 cases, while 27 cases show a coverage greater or equal to 80%. Moreover, OSMOSE reports a very poor coverage on only two examples. Comparisons with the random generator turn almost always in favour of OSMOSE, both in terms of coverage, computation time and test efficiency (ratio between coverage and number of tests). For example, random test generation achieves a coverage greater or equal to 80% in only 9 cases. Random generation beats OSMOSE only in two examples.

Results suggest that OSMOSE performances are sufficient for unit testing of low call-depth functions on industrial case-studies. OSMOSE requires indeed less than 1 minute to achieve a 98% coverage of a function with 140 branches, and less than 10 minutes to achieve a 87% coverage of a function with 500 branches and 2247 instructions. However, performances decrease when used on functions of high-call depth (aircraft examples 7 to 9) with many instructions in subcallees. Note that actually OSMOSE has been tested on 40 functions of the aircraft program with low call-depth (between 0 and 4): very good

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr



name	Proc	Mode	I	Br	OSMOSE cover	OSMOSE time	OSMOSE #tests	random cover	random time
msquare 3×3	c509	full	272	46	100%	10	43	63%	150
msquare 4×4	c509	full	274	46	100%	120	123	67%	150
hysteresis	c509	full	91	16	100%	60	35	56%	40
merge	c509	full	56	24	100%	20	70	45%	100
triangle	c509	full	102	38	100%	1	15	71%	25
cell	c509	full	23	8	100%	1	10	87%	25
list	c509	full	13	6	100%	1	3	100%	10
msquare 3×3	ppc	full	226	30	100%	10	34	56%	110
msquare 4×4	ppc	full	226	30	82%	60	125	50%	120
hysteresis	ppc	full	76	16	100%	60	251	20%	60
merge	ppc	full	188	16	100%	1	2	100%	1
triangle	ppc	full	40	18	100%	1	19	77%	10
cell	ppc	full	18	8	100%	1	8	50%	10
list	ppc	full	15	6	100%	1	4	66%	34
check-pressure	c509	full	59	10	100%	10	4	90%	160
buf-get	ppc	full	262	18	100%	10	14	66%	600
strtok	ppc	full	316	40	100%	450	183	90%	180
strlen	ppc	full	134	18	100%	10	22	94%	120
findcolor	ppc	full	283	36	97%	800	328	61%	800
countlib	ppc	full	328	44	100%	120	48	54%	300
aircraft0	ppc	full	237	36	100%	10	19	40%	20
aircraft1	ppc	full	290	140	98%	60	43	64%	100
aircraft2	ppc	full	201	72	100%	10	37	35%	20
aircraft3	ppc	full	977	190	50%	60	3	96%	60
aircraft4	ppc	full	2347	500	87%	600	15	68%	600
aircraft5	ppc	unit	121 / 4103	2 / 509	100%	1	2	100%	10
aircraft6	ppc	unit	250 / 425	18 / 34	94%	100	9	83%	120
aircraft7	ppc	unit	506 / 15640	20 / 2790	80%	20	4	75%	500
aircraft8	ppc	unit	957 / 30969	14 / 4952	14%	10	3	50%	500
aircraft9	ppc	unit	627 / 31793	74 / 5034	77%	600	12	63%	600

Proc: processor, I: #instructions, Br: #branches, Time in seconds
cover: branch coverage achieved, unit: coverage of top function only

Table II: Evaluation of the test generation abilities of OSMOSE

coverage (100%) was achieved in 31 cases, bad coverage ($<$ 50 %) was achieved in 4 cases, and good coverage (between 70% and 95%) was achieved in the last 5 cases.

Comments. OSMOSE performances on the handcrafted programs 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. Note also that the former version of OSMOSE [9] does not terminate on `msquare 4×4` (PPC) within 5 minutes, while the optimised version terminates in 40 seconds (nonetheless, full coverage is still not achieved).

It can be noticed that random generation is rather slow here. Since tests are run in simulation, their execution is much more expensive than in a usual random testing framework.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

9. Related work

This section describes various techniques and tools related to OSMOSE, from machine code analysers to concolic execution and bit-vector solving.

Test data generation from machine code. This article is an extended version of a paper presented at ICST 2008 [9]. Compared to the conference paper, this article adds a new section on the Generic Assembly Language, a new section on the bit-vector solver, more experiments, an up-to-date section on related work and more detailed explanations on various aspects. Two other structural testing tools for machine code based on concolic execution have been developed recently: SAGE [24] at Microsoft and BITSCOPE [12] at CMU. Moreover, the tool PEX [47] developed also at Microsoft is geared toward test data generation from .NET bytecode, whose abstraction level lies somewhere in between real machine code and Java bytecode.

SAGE aims mainly at debugging, and the tool is dedicated to the x86 architecture. Concolic execution is enhanced with an original heuristics (instead of the standard DFS) geared toward faster coverage and bug finding, and various optimisations are implemented to reduce the number of calls to the solver. Path predicates are expressed in linear arithmetic, non-linear constraints being abstracted via concretization. Successful experiments have been performed on real-life programs. Notice that the problem of IR recovery in presence of dynamic jumps is not mentioned in the paper. A reason may be that the authors target implicitly bug finding for non-critical systems. As a consequence, they may not consider issues such as maximal coverage or native code comprehension.

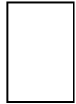
BITSCOPE intends to help the user to understand the behaviour of a malware. The tool is dedicated to x86 executables for a Windows operating system. This restriction allows BITSCOPE to take into account system calls, via an instrumentation of the QEMU emulator. BITSCOPE uses concolic execution (referred to as *mixed execution*) with path predicates expressed into bit-vector theory. Formulas are solved by an external solver, currently STP [22]. Original features of the tool are undoubtedly the ability to take system calls into account *both concretely and symbolically*, to perform goal-oriented test data generation in a concolic execution setting (i.e. to generate a test data to exercise a particular instruction of the program) and to take advantage of hand-crafted summaries (for standard C procedures on strings) to speed-up the analysis. The technical report [12] focuses mainly on the application of BITSCOPE to malware dissection. IR recovery in presence of dynamic jumps is not addressed in the paper. It may be a problem since one of the main goals of the tool is to help the user to understand all possible behaviours of an executable.

PEX [47] is geared toward test data generation from .NET bytecode. In this context, the dynamic resolution of virtual or inherited methods may pose problems similar to the one of dynamic jumps. The PEX conference paper [47] explains briefly that the tool is able to reason about type constraints to handle this issue. It would be interesting to investigate further the relationship between dynamic jumps found in high-level bytecodes such as Java and .NET and dynamic jumps found in machine code. Note also that PEX relies on a concolic execution engine equipped with an efficient coverage-based search heuristics [52].

IR recovery and static analysis of machine code. A few commercial tools are already available. The disassembler IDA Pro [50] addresses the problem of IR recovery. Tools from the Absint company [49] are actually geared towards verification of non-functional properties like estimation of maximal stack

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



height or worst-case execution time, with a core technology based on static analysis. Both tools support a wide range of architectures and instruction sets. They address IR recovery in similar ways: mostly by pattern matching and very basic static analysis, with potential help from the executable header and user annotations. As a consequence, results can be quite inaccurate on dynamic jumps.

Second, safe static analysis techniques have been developed recently [8, 14, 15, 33, 34]. Since the goal is to compute statically a safe and tight over-approximation, the technology is very sophisticated. These IR recovery techniques are difficult to implement for the non-expert because they target both completeness and tightness of approximations. Note that while the IR technology presented by Reps *et al.* [14, 15] may be implemented in a safe way, the current implementation in CODESURFER/X86 [8] is not: the analysis can only recover missing edges among a predefined set of instructions (actually, the output of IDA Pro), but no missing instruction can be recovered. The tool JAKSTAB performs a truly safe recovery [33]. Reps *et al.* have also developed a verification technology for executable files based on model-checking the recovered abstract model [43], but no practical experimentation have been reported.

In this paper, considering the problem from a testing perspective allows to relax the completeness requirement. This greatly simplifies the implementation of the static part, while correctness is ensured by the concolic step.

Generic machine code analysis. Lim and Reps have designed a generic framework called TSL [36] to easily specify a new ISA/analyser into CODESURFER/X86. TSL is based on a ML-like language whose basic connectors are overloaded according to the semantic required by the analyser. TSL shows obvious connections with the framework developed here. The main difference is on the power of the description language: TSL is based on a true programming language, while in GAL a native instruction can only be encoded into a sequence of micro-instructions, forbidding any nested loops. This restriction may cause problems for a few complex instructions with loops in their micro-code.

Connex problems. Some problems are closely related to the analysis of native machine code. First, some works aim at ensuring the conformance between machine code and high-level source code, thus reducing machine code analysis to source code analysis. This line of research includes certified compilation [13] as well as invariant preservation checking [40]. Though very interesting, these approaches are limited to applications where the source code is available, while it is not always the case. Second, a few recent verification tools for the C language take into account the low-level semantic of data, for example the test data generation tool EXE based on concolic execution [16] and the verification tool CALYSTO [10] based on static analysis. However, these tools do not address the low-level semantic of control in C programs. EXE relies on the bit-vector solver STP and a best-first search enumeration of paths similar to the one of SAGE. Finally, many works have been conducted on Java bytecode verification, for example the test data generation tool JMOPED [17, 44]. 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. Moreover, inherited methods and virtual methods are not handled.

Concolic execution. As already mentioned, concolic execution-based tools have literally blossomed up since a few years. In addition to BITSCOPE, EXE, SAGE and PEX already mentioned, other

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

works include DART [23], CUTE [45] and PATHCRAWLER [51]. These three tools work at the programming language level (C for all three and also Java for CUTE). They 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 [27, 38]. 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 [41].

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. First they consider integer arithmetic instead of bit-vector arithmetic. Second, these tools discover dynamically the program, having at each step only a local view of the CFG, while OSMOSE relies on a (abstract) global view, allowing some specific optimisations [11]. Third, 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.

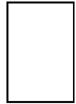
Constraint Programming-based test data generation. INKA [20, 21] performs structural test data generation on C programs through Constraint Programming. In this approach the whole program is translated into an equivalent CP problem, while the techniques presented so far translate only one path at a time. Notably, INKA includes a solver for floating-point arithmetic constraints [7]. Other CP-based testing tools have been developped for programs and models [5]. OSMOSE is the only one targeting executable files, and one of the very few CP-based tools dealing with bit-vector constraints.

Bit-vector solving. Many solvers for bit-vector theory have emerged recently [4, 6, 22], taking advantages of the recent dramatic increase in performances of SAT solvers. These solvers are based on bit-blasting: the original problem is encoded into a SAT problem, each bit of every bit-vector being represented by a boolean variable and each bit-vector constraint being represented by its logical circuit implementation, which is then solved by a state-of-the-art SAT solver. While very effective on bitwise-oriented problems, bit-blasting is well known to be less efficient on more high-level constraints, such as (non-linear) arithmetic [32]. Current SAT-based bit-vector solvers combine bit-blasting with heavy preprocessing to partly remedy this issue [22].

The approach proposed here stays at the exact opposite: bit-vectors are seen as integer variables and constraints are encoded as integer constraints. It turns out that this method performs better on arithmetic-oriented problems than on bitwise-oriented problems. Some work have already been conducted in this direction in the VHDL verification community [18, 53]. However, these approaches show some drawbacks: (1) they use the CP solver in a black box manner, preventing any "deep" optimisation, and (2) they handle bitwise constraints with a bit-blasting technique, which is very inefficient in a CP setting. The OSMOSE solver avoids bit-blasting through delayed computation and specific optimisations. Moreover, it relies on a bounded arithmetic solver optimised for constraints over very large domains, incorporating state-of-the-art technologies such as congruence domains [35] and

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr



global constraints to detect quickly unsatisfiable parts of the search space [19]. It would be interesting to conduct an in-depth comparison between SAT-based and CP-based approaches for solving constraints over bit-vectors.

10. Conclusion and future works

Verification at the machine code level is 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. This paper shows how to perform automatic structural test data generation on a standalone executable. The approach followed is to rely on existing robust technologies, namely concolic execution and Constraint Programming, and to adapt them to the specific issues appearing in binary-level analysis. Innovative techniques for IR recovery have also been developed. 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, the test data generation technology needs to be improved in order to scale up to larger programs, and especially to handle functions with many nested calls or very long paths. There are different possibilities, from modular generation [1, 27, 38] to hybrid generation [41] or dedicated constraint solving techniques [6, 22]. Second, the user interface of the tool must be improved 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 [14, 15, 33, 34] could give assurance about the quality of both the abstract control-flow graph and the coverage measure returned by OSMOSE. This is not yet done in the tool, but would be useful for safety applications where preciseness of the coverage measure is essential.

Acknowledgements. The authors wish to thank the three anonymous referees for their helpful comments, contributing to a significant improvement of the paper.

REFERENCES

1. S. Anand, P. Godefroid and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *TACAS 2008*. Springer.
2. K. R. Apt. Principles of Constraint Programming. Cambridge University Press, 2003.
3. K. R. Apt and M. Wallace. Constraint Logic Programming using Eclipse. Cambridge University Press, 2007.
4. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *TACAS 2009*. Springer.
5. S. Bardin, B. Botella, F. Dadeau, F. Charreteur, A. Gotlieb, B. Marre, C. Michel, M. Rueher and N. Williams. Constraint-based software testing. In GDR-GPL meeting, 2009.
6. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti and R. Sebastiani. A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems. In *CAV 2007*. Springer.
7. B. Botella, A. Gotlieb and C. Michel. Symbolic execution of floating-point computations. In *STVR* vol. 16, 2006.
8. G. Balakrishnan, R. Gruian, T. W. Reps and T. Teitelbaum. CodeSurfer/x86-A Platform for Analyzing x86 Executables. In *CC 2005*. Springer.
9. S. Bardin and P. Herrmann. Structural Testing of Executables. In *ICST 2008*. IEEE Computer Society.
10. D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking, In *ICSE 2008*. ACM.
11. S. Bardin and P. Herrmann. Pruning the search space in path-based test generation. In *IEEE ICST 2009*. IEEE.

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using stvrauth.cls

DOI: 10.1002/stvr

12. D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam and D. Song. BitScope: Automatically Dissecting Malicious Binaries. Technical report CS-07-133, CMU, 2007.
13. S. Blazy, X. Leroy. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785, pages 280-299, 2005.
14. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC 2004*. Springer.
15. 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.
16. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler. EXE: automatically generating inputs of death. In *CCS 2006*. ACM.
17. J. Esparza and S. Schwoon. A BDD-based Model Checker for Recursive Programs. In *CAV 2001*, Springer.
18. F. Ferrandi, M. Rendine and D. Sciuto. Functional verification for SystemC descriptions using constraint solving. In *DATE 2002*.
19. T. Feydy, A. Schutt and P. J. Stuckey. Global difference constraint propagation for finite domain solvers. In *PPDP 2008*. ACM.
20. A. Gotlieb, B. Botella and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *ISSTA 1998*. ACM.
21. A. Gotlieb, B. Botella and M. Watel. Inka: Ten years after the first ideas. In *ICSSEA 2006*.
22. V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *CAV 2007*. Springer.
23. P. Godefroid, N. Klarlund and K. Sen. DART: Directed Automated Random Testing. In *PLDI'2005*. ACM.
24. P. Godefroid, N. Klarlund, M. Y. Levin and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS 2008*.
25. N. Gupta, A. P. Mathur and M. L. Soffa. Automated Test Data Generation Using an Iterative Relaxation Method. In *FSE 1998*.
26. N. Gupta, A. P. Mathur and M. L. Soffa. UNA Based Iterative Test Data Generation and its Evaluation. In *ASE 1999*.
27. P. Godefroid. Compositional dynamic test generation. In *POPL 2007*. ACM.
28. `http://www.hispano-suiza-sa.com/`
29. J. C. King. Symbolic execution and program testing. Communications of the ACM, 19(7), july 1976.
30. B. Korel. Automated Software Test Data Generation. In *IEEE TSE*. IEEE, 1990.
31. B. Korel. Automated Test Data Generation for Programs with Procedures. In *ISSTA 1996*.
32. D. Kroening and O. Strichman. Decision Procedures: An Algorithmic Point of View. Springer, 2008.
33. J. Kinder and H. Veith Jakstab: A Static Analysis Platform for Binaries In *CAV 2008*. Springer.
34. J. Kinder, F. Zuleger and H. Veith An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *VMCAI 2008*. Springer.
35. M. Leconte and B. Berstel. Extending a CP Solver With Congruences as Domains for Software Verification. In *Workshop on Constraints in Software Testing, Verification and Analysis. CP 2006*. Springer
36. J. Lim and T. W. Reps. A System for Generating Static Analyzers for Machine Instructions. In *CC 2008*. Springer.
37. B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions: GATeL. In *ASE 2000*. IEEE.
38. P. Mouy, B. Marre, N.Williams and P. Le Gall. Generation of All-Paths Unit Test with Function Calls. In *ICST2008*.
39. J. Offutt, Z. jin and J. Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. In *Software Practice and Experience*, 29 (2), January 1999.
40. X. Rival. Invariant Translation-Based Certification of Assembly Code. In *STTT*, 6(1), July 2004.
41. R. Majumdar and K. Sen. Hybrid Concolic Testing. In *ICSE 2007*. IEEE.
42. F. Rossi, P. Van Beek and T. Walsh, editors. Handbook of Constraint Programming. Elsevier, 2006.
43. T. Reps, S. Schwoon, S. Jha and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. In SCP, october 2005.
44. D. Suwimonteerabuth, F. Berger, S. Schwoon and J. Esparza. jMoped: A Test Environment for Java programs. In *CAV 2007*, Springer.
45. K. Sen, D. Marinov and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE 2005*. ACM.
46. A. S. Tanenbaum. Structured Computer Organization. Prentice Hall, 6th edition, 2005.
47. N. Tillmann and J. de Halleux. Pex-White Box Test Generation for .NET. In *TAP 2008*. Springer.
48. N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/SIGSOFT FSE 2005*. ACM.
49. `http://www.absint.com/`
50. `http://www.datarescue.com/`
51. N. Williams, B. Marre and P. Mouy. On-the-Fly Generation of K-Path Tests for C Functions. In *ASE 2004*. IEEE.
52. T. Xie, N. Tillmann, P. de Halleux and W. Schulte. Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In *DSN 2009*. IEEE.
53. Z. Zeng, M. Ciesielski and B. Rouzeyre. Functional test generation using Constraint Logic Programming. In *VLSI-SOC 2001*.

Copyright © 2009 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr