# Symbolic Analysis of Assembly Traces: Lessons Learned and Perspectives

Raphaël Khoury
Université du Québec à Chicoutimi
Department of Computer Science and Mathematics, 555, boulevard de l'Université, Chicoutimi, QC, Canada G7H 2B1
raphael.khoury@uqac.ca

***Abstract*—In this study, we have developed a software to implement a symbolic analyzer for assembly traces. The software receives as input traces of assembly instructions. It then builds a symbolic expression characterizing the possible range of values for each variable and feeds this value to the Yices STM solver. The Yices solver returns possible concrete values that respect the symbolic expressions associated with each variable. This software has several potential applications including software testing and fuzzing and vulnerability detection. To verify the validity of our approach, we have tested our software with real-life traces and investigated its potential use for malware detection. For instance, that the software automatically detects the input values that would cause a buffer overflow in some cases. To conclude, we reflect on the lessons learned during the development of this software, which can help guide the future development of symbolic analyzers.**

## I. Introduction

Symbolic evaluation (or symbolic execution) is a method of software analysis that allows us to reason about the possible values of program variables at each program point and about the inputs that lead the program into these different states. This paper presents research performed on behalf of the Canadian armed forces at the *Research and Development for Defence Canada* research Center (RDDC), which consisted in the development of a symbolic interpreter for assembly traces, as well as studies aimed at exploring the use of this interpreter to improve the software security. The software takes as input a trace file of assembly instructions, corresponding to a trace of the execution of a target. This trace is generated using an in-house tracer previously developed at DRDC. Using symbolic analyses techniques, the interpreter produces logical expression describing the sets possible input values that allow the could lead the program along the current execution path. Manipulating these expression allow us to explore alternative execution paths in an efficient manner. We use the interpreter to detect potential vulnerabilities in software.

Consider the fragment of code in figure 1. An input value is read on line 1, is then modified and, depending on the resulting value, the program may or may not enter an error state. However, only one possible input value out of $2^{32}$ possible values will cause the program to reach this error state. If we rely exclusively on randomly generated inputs or black-box fuzzing to test this program, the probability of ever encountering the error state would be very limited with each randomly generated input giving the tester one additional chance out of $2^{32}$ to detect the error state. Date triggered malware is a real-life example of this issue.

```
1 void _me (int x) {
2       int z = double(x);
3       if (z==10)
4          abort(); /* error */
5       }
```

Fig. 1: Example 1

In contrast, symbolic execution allows for the program to be executed with a symbolic, rather than a concrete value. In this context, a concrete value refers to the actual value of a program variable during the execution while a symbolic value refers to a range or set of possible concrete values. Each program variable is assigned a symbolic value, which abstracts its concrete value. The initial value of each program variable is normally the entire range of possible values for that variable type (i.e. $0$–$2^{32}$ for integers or $\{\text{true} \cup \text{false}\}$ for booleans) unless some information is provided that constraints the initial value of the variable. The symbolic value then evolves with each program instruction that manipulates it, and is constrained when a conditional control flow instruction is encountered.

Furthermore, whenever a conditional control flow instruction is encountered, a special constraint, called a path condition, is created. The path condition is a boolean constraint which must hold for a given path be possible. In the example of figure 1, two path constraints are created on line 3, namely 2*x==10 if the condition hold and 2*x!=10 if it does not hold. These path conditions are what allow us to explore the execution space of the program in a more effective and directed manner than with random testing. A valuation of the variables in the path condition corresponds to concrete initial values to the program variables which would lead the program to take a specific execution path. If no such valuation is possible (i.e. the path condition is unsatisfiable) then there are no possible input values for which the execution of the program reaches the program point where the path condition is created and then selects the branch of the conditional associated with the path condition. Figure 2 shows how the symbolic constraints on the program variables and the path constraints evolve during the analysis of the code.

In the example in figure 1, the entire state space of possible execution paths can be explored with only two input values. In practice, the set of possible may be infinite. Nonetheless, symbolic execution is a useful tool target input generation for testing and fuzzing in a manner that maximizes the number of different paths explored. Symbolic analysis can

978-1-4673-6917-6/15/$31.00 © 2015 IEEE

```
                                        Symbolic State          Path Constraint
1 void _me (int x, int y) {             x, y = anything
2       int z = double(x);                   z= 2*x
3       if (z==y)
4          abort(); /* error */                                    2*x==y
5       }
```

Fig. 2: Example 2

be used on source code, on assembly level code or in our case, on execution traces. The use of execution traces allows us to circumvent several problems that arise when symbolic execution is performed directly on the code. For example, certain methods or fragments of code may be unavailable or uncomputable. It would be very difficult, for example, to assign a path constraint to a segment of code that computes the hash function of an input, check it against a remote value and takes one of two control flow path according to the result of this check. Malware, whose discovery and understanding is an important target of analysis, frequently relies on self-modifying code as a means to dissimulate itself, making symbolic analysis of the code impossible.

The rest of this paper is organized as follows. Section 2 presents our software. Section 3 discusses some lessons learned during the development process, which we believe may guide further research. Section 4 presents a review of the literature. Concluding remarks are given in section 5.

## II. Overview of the Software

The program receives as input a trace file of assembly instructions. It then builds a symbolic expression characterizing the possible range of values for each variable and feeds this value to the Yices SMT solver [1]. Yices returns possible concrete values that respect the symbolic expressions associated with each variables. These new input values allow the target software to be run and monitored along a new execution path. This is particularly interesting whenever a specific conditional jump or another instruction is pinpointed as being potentially useful as part of a vulnerability. Such determinations can occur as the result of other static analyses. In this case, the symbolic analysis suggest inputs values that will follow the same execution up to that specific instruction and then diverge on evaluation of that specific condition. The interpreter is thus a type of concolic execution, in which the symbolic analysis draws upon the results of the dynamic monitor for its input, and in turn, the dynamic monitor draws upon the result of the symbolic analysis to generate new interesting traces. This synergy is illustrated in figure 3.

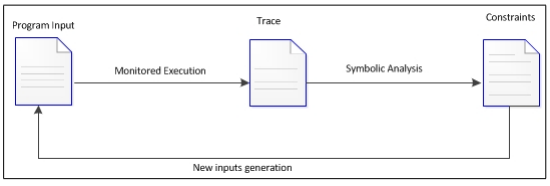


Fig. 3: The static-dynamic synergy of the symbolic interpreter

Figure 4 gives a top-level view of the interpreter's architecture. The input file is first preprocessed and parsed. The interpreter then processes the lines of the input file one at a time, as each line corresponds to the execution of one assembly instruction or one system call. The interpreter maintains two models of the program's current state in parallel, a concrete model and an abstract model, and updates both states each time a new line of the trace file is read and processed.

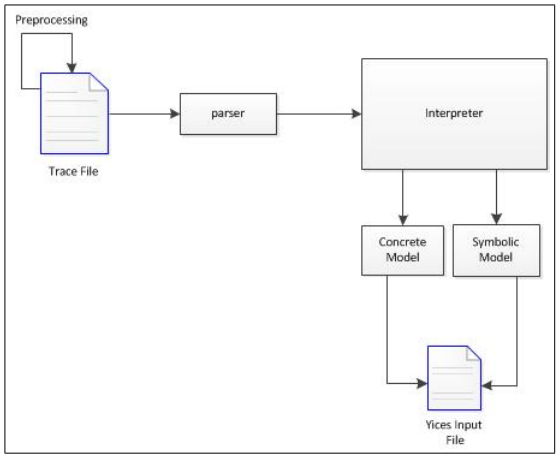


Fig. 4: Overview of the software

The concrete state maps each register with the exact 32-bit value it contains, and each memory address with the 8-bit value it contains. The flag register is represented by a string indicating which flags are set. The first line of the trace file contains initial values for each register, while the memory is initially blank until instructions that read or write to it are encountered. Likewise, when the execution returns from a system call, the values contained in the registers are updated. With few exceptions, changes to the memory performed by system calls are not generally modeled. The abstract state (or symbolic state) maps registers and memory locations to symbolic expressions, which can be translated into Yices's input language.

The symbolic model maps the registers, flags and memory locations to a symbolic expression that expresses a range or set of possible concrete values. Both registers and memory locations are modeled by 32-bit vectors. Each flag is modeled by an abstract representation of a boolean value. Only the carry (C), zero (Z), sign (S) and overflow(O) flags are modeled. Finally, each time a conditional jump is encountered, an additional abstract value, called a path condition, is created. This path condition also maps to a boolean value.

When a value is read in memory, it is initially associated with an unconstrained 32-bit vector. In practice this is represented simply by a variable name, with no associated

restrictions. The values associated with flags range over True, False and Top, the latter being used when the value of the flag is undefined, which occurs after the execution of certain instructions. Path constraints always evaluate to true or false.

Note that since the memory is actually indexed at 8-bit intervals, it is possible that the symbolic values representing the values in memory become interleaved, with multiple variables referring to the same memory locations. For example, the snapshot in figure 5 contains symbolic values both for address 4f77e0 and for address 4f77e1. However, the upper 24 bits of the value at 4f77e0 are also the lower 24 bits of the value at 4f77e1. These bits are thus modeled by two different symbolic values (in this case, two unconstrained bit-vectors).

Figure 5 gives a snapshot of the content of a pair of concrete and abstract states at a given moment of the program's execution.

The generation of the concrete model is optional. Its main purpose is to provide a validation during the development process to ensure the symbolic expressions correctly model the semantics of the assembly instructions.

### A. Examples

The inputs for the interpreter are text files containing traces of assembly instructions generated by a tracer developed at Defence Research and Development Canada(DRDC) for Intel 64 architectures. Each line of these files corresponds to a single assembly-level instruction performed by the system being monitored or to a system call. It records a variety of information including the name of the instruction, its parameters (registers, literals or memory locations) and the value associated with each parameter.

A short fragment of a trace is shown in figure 6. Each line begins with a sequential number, and then contains three sets of information, separated by —. The first is the assembly instruction that occurred, with its parameters. This is followed by the initial values present in each relevant register or memory location before this instruction is executed. Finally, the tracer records the values of registers, memory locations or flgas that changed were modified by the instruction. In this example, the trace is that of a simple C program that reads an integer value from a file. Subsequently, a branching instruction checks whether or not the value that has been read equals 1000 (which is the case in our example) and branches accordingly. The first element of each line is a sequential line number. The relevant assembly instructions are on lines 06bb62 and 06bb63 of the trace file. The complete trace file contains about 470 000 lines.

While DRDC is guarded about the inner workings of its proprietary tracers, the symbolic analyser could easily be adapted to take as input traces from other tracers, with only the parser class necessitating modifications.

The analyzer generates the symbolic expressions that characterize each program variable at each program point. When translated into Yices's input language, these constraints have the following form:

```
(assert (= L0021f840 (bv-add V9 V2) ))
```

The constraint above states that the value at the location 0021f840 in memory is the sum of the initial values of two program variables (sequentially numbered V9 and V2). As the execution progresses, the constraints quickly become much more complex and constraints that comprise several thousand terms become common, even for short traces.

Note that the jump on line 06bb63 is not taken. The abstract interpreter generates the following path constraint:

```
(assert (= PC441187 (= (bv-sub (bv-concat
(bv-extract 31 8 V134) (bv-extract 7 0 V67))
(mk-bv 32 1000)) (mk-bv 32 0))))
```

Stated informally, this constraint, sequentially named PC441187, is built from two 32-bit values read in memory namely V67 and V134. It states that the first 8 bits of V67, concatenated with the last 24 bits of V134, when subtracted from 1000, equal 0. This constraint abstracts the semantic of the instructions leading to the conditional jump on line 06bb63.

These expressions are then fed to the Yices solver which assigns a possible concrete value to each variable, if such an assignation is possible, or returns that the constraint set is unsatisfiable otherwise.

```
(= V134 0b01110111011101111101100011010100)
(= V67 0b00000000001000011111100101000000)
```

The jump was not taken, thus the path constraint evaluates to false.

```
(= PC441187 false)
```

To generate initial values that reach this path, but take a different branch, the user must run Yices with the constraints generated by the interpreter up to that point, and add the following constraint:

```
(assert (= PC441187 true))
```

This constraint will force Yices to find initial values to all the variables in such a way that path constraint PC441187 evaluates to true. Conversely, if there does not exist any valuation of the program variables for which the execution reaches that program point and then diverges, Yices will indicate that the constraint set is unsatisfiable.

In another example, we examined the trace of a C program in which a variable-length string was copied to a fix-length buffer on the stack. This operation creates the condition for a stack overflow if the string is longer than the buffer to which it is being written. At the assembly level, when string copies occur, a register is initialized with the number of iterations of the string operation that must be performed. The value of this register is decremented iteratively until it reaches zero, while the string is being copied. The interpreter can build constraints on the initial value of the counter register in such manner that the Yices solver would suggest input values that lead to an overflow. This can be detected even if the execution under consideration does not contain a buffer overflow.

### B. Constraint Simplification

We found that a major difficulty in scaling the abstract analyzer to the size of the program traces generated by the monitoring of real program has been its considerable memory

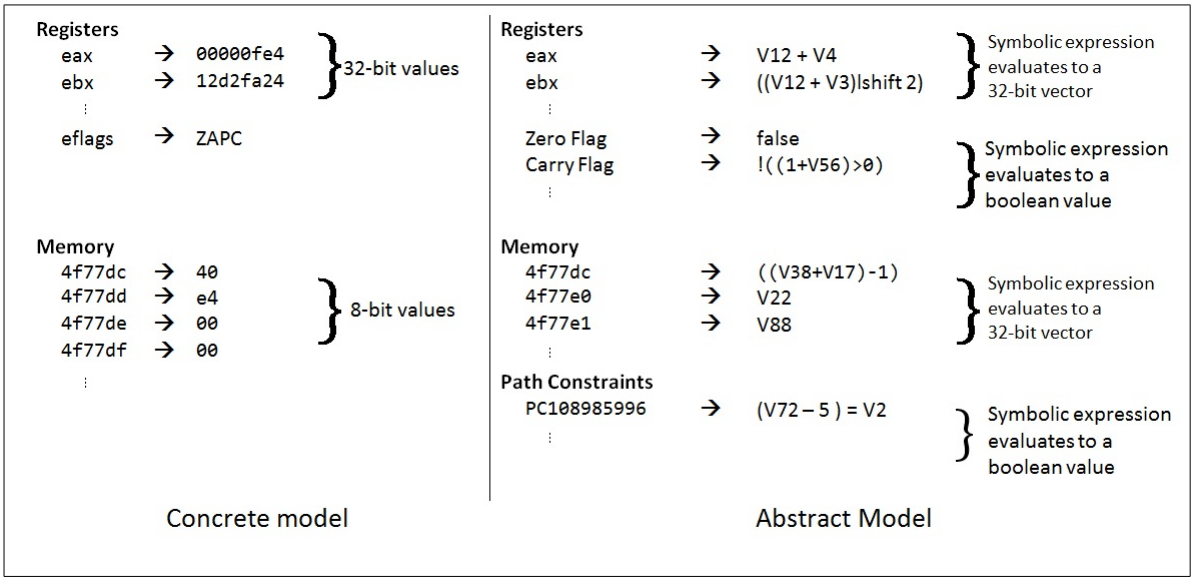


Fig. 5: Overview of the software

```
06bb5c mov esp, ebp | EBP=001bfbf4 | ESP=001bfbf4
06bb5d pop ebp | ESP=001bfbf4 [001bfbf4]=001bfc24 | EBP=001bfc24 ESP=001bfbf8
06bb5e push ecx | ECX=71f1a8b9 ESP=001bfbf8 | ESP=001bfbf4 [001bfbf4]=71f1a8b9
06bb5f ret | ESP=001bfbf4 [001bfbf4]=71f1a8b9 | ESP=001bfbf8
06bb60 ret | ESP=001bfbf8 [001bfbf8]=01391036 | ESP=001bfbfc
06bb61 add esp, 0x20 | ESP=001bfbfc | ESP=001bfc1c EFLAGS=
06bb62 cmp dword ptr [ebp-0x4], 0x3e8 | [001bfc20]=000003e8 EBP=001bfc24 | EFLAGS=ZP
06bb63 jnz 0x1391057 | EFLAGS=ZP |
06bb64 push 0x1392144 | ESP=001bfc1c | ESP=001bfc18 [001bfc18]=01392144
```

Fig. 6: A fragment of assembly trace

usage. Memory use increases monotonically with the number of trace lines that are processed, as new constraints are added to the model. Furthermore, since new constraints are usually built from existing ones through an aggregative process, the length of constraints also tends to increase in proportion with the number of lines that have been processed up to the point where each constraint is created.

Two strategies are employed to minimize the memory footprint of the program. The first is to systematically delete constraints once they no longer occur in the remainder of the target trace. This is possible since we operate symbolic analysis on a complete trace of the execution of interest, rather than on source code, which makes it a straightforward task to identify and record the last use of each memory location.

The second strategy we rely upon is to simplify the size of the constraints at the moment when they are created. A number of semantics-preserving simplifications can be applied to the constraints as they are generated. We list a few examples below.

- The trace contains of number of instances of the instruction `xor reg reg`, in which a register is xored with itself to clear it. When this occurs, the interpreter replaces the symbolic value associated with this register with the constant 0. Since the `xor reg reg` instruction frequently occurs at the beginning of a series of computations involving this register value, this simplification can yield a substantial reduction on the size of later constraints. The effect is compounded by the fact that having the constant 0 in a constraint allows a number of other simplifications, based upon arithmetic identities, to be applied with ease.
- Like the xor instruction, the test and sbb instructions can be applied by a register to itself as the result of compiler generated code. When this happens, the abstracts constraints representing the affected registers and flags can be simplified, compared with the general case of two distinct operands.
- Arithmetic and boolean identities are used to simplify expressions.
- When the cmpxchg and cmpxchg8b instructions occur, the interpreted performs a syntactic equality check between its two operands. If they are equal, which happens frequently in our subset of traces, the constraints representing the registers and flags affected by this instructions can be substantially reduced in size since they no longer contain a conditional expression.
- Other simplifications can be performed by reflecting upon both the semantics of the instruction and their operands. For instance, we have observed that the cmpxchg instruction (a conditional swap) is often applied to a destination operand that is itself a conditional expression. This later conditional expression will thus be present twice in the abstract expressions representing the destination operand and eax after the

execution of the cmpxchg. For instance, if the original value of eax is the constant 17 and the destination is the conditional `(if (V18 == 0)` then 17 else V18). The final value of eax is then : if((if (V18 == 0) then 17 else V18)== 17 then 17 else (if (V18 == 0) then 17 else V18)). Note that the ultimate `if` can never be evaluated to its first branch. The expression can then be simplified to if((if (V18 == 0) then 17 else V18)== 17 then 17 else V18).

- The most frequent types of bit-vector operations in the constraints are extractions of part of a bit vector, concatenations of two bitvectors, sign-extentions and zero-extentions. Indeed, a single assembly instruction may generate as many as half a dozen such instructions. A number of identities are applied to minimize the number of these constraints as soon as they are generated.

We found that the elaborate constraints (often numbering thousands of terms) that are generated when naively abstracting the content of an execution trace could be simplified by several orders of magnitude by applying the type of simplifications detailed above. Furthermore, we found that minimal semantics-preserving changes in constraint could have a substantial impact on the evaluation time when the constraint was fed to Yices.

### C. Benchmark

We ran the software on a i5-M540 2.53GHz dual core. On a sample file of 250 000 lines it requires 4.7 seconds to run, including the time to generate the constrains as well as the time needed by Yices to resolve them. It generates about 3700 symbolic variables, most of which have a very simple form, often less than 5 terms. The longest constraint extant at the end of the execution process has 430 terms.

## III. Lessons Learned

The development of this prototype allowed us to draw several insights about the use of abstract interpretation in the context of the security analysis of software.

**The interpreter is an effective tool for security analysis**
The first and most interesting insight that comes out of the prototype development is that symbolic interpretation is an effective tool for the security analysis of code. Symbolic interpretation allows us to guide fuzzing and detect potential vulnerabilities in code. Furthermore, when binaries are unavailable, symbolic interpretation can be performed directly on assembly level traces. One of my ongoing research projects is to state the conditions that can lead to an attack in the form of solvable constraints over sets of initial values. As discussed in section 2, our software is already capable of looking for certain types of buffer overflow. We continue to investigate this issue.

**Memory usage must be managed carefully**
Programmers often trade higher memory consumption for lower execution time. However we found that the symbolic interpreter had such an expensive memory consumption that care had to be taken to minimize it at each step of the development. Indeed, this is not surprising since many static analysis and model-checking methods similar to symbolic execution also tend to be memory-intensive. In our experience, we found that an important part of the development process was taken by optimizing the memory usage of the software, using methods described in section II-B. Indeed, initial versions, which naively implemented the symbolic analysis algorithm, could not scale beyond 2-3 thousand lines of assembly code. Only after steps were taken to improve memory usage at every step of the computation were we able to treat traces of a realistic length, containing upwards of several hundred thousand assembly instructions.

**The solver must be chosen judiciously**
We have experimented with several different solvers before settling on Yices (which itself can operate using one or more of several different logics, the choice of which affects both its capabilities and its efficiency). Each solver exhibits advantages and drawbacks, which will affect the interpreter's ability to find adequate solutions. In this respect, it is interesting to observe that in general, the problem of finding solutions to the constraints is an NP-complete problem, thus, regardless of the choice of the solver and of the efficiency with which the interpreter optimizes the constraints, some of the constraints will necessarily require considerable time to be resolved.

**The constraints should be tailored to the solver**
This insight was first stated by Thanassis et al. when discussing the experience of his team in the development of an symbolic interpreter at Carnegie Mellon University. They further observed that : "*it is more fruitful to view the SMT solver as a search procedure and use optimizations to guide the search*"[2]. In an ideal situation, the inner workings of the solver should be well documented, thus allowing the constraints to be constructed in such a manner to ensure (or at least maximize) their timely resolution. In our experience, slight semantic-preserving changes to the structure of constraints was often sufficient to improve the resolution time by an order of magnitude or more.

**We must reason about *exploitability***
A given string copy operation may not be liable to cause a buffer overflow in the executed program, no matter its input. When looking for vulnerabilities in a trace that may contain several thousand string copy instructions, it is necessary to narrow the search, focusing on the instructions most likely to be at the source of an intrusion. We are currently endeavoring to state the constraints which the solver will verify in such a way as to ensure that the analysis remains focused on potential vulnerabilities.

## IV. Review of the Literature

While symbolic execution was first suggested over thirty years ago in [3], the method has recently seen a renewed interest, due in part to recent developments in algorithmic and computational power that have made the procedure more tractable [4]. As discussed in section 1, symbolic execution proceeds by simulating the execution of a target program using symbolic values, rather than concrete values. Each program state then corresponds to a mapping of program variables to symbolic values, and a path constraint. A symbolic execution tree, whose nodes are program states, tracts the execution paths examined during the analysis. We find in the literature several examples of recent approaches that follow this method, such as [5], [6], [7].

Of particular interest in the context of this evolution was the work of Godfroid et al. in[8] who suggested that symbolic analysis be used in conjunction with concrete execution, a form of analysis termed concolic execution (for concrete and symbolic). The guiding principle of this type of analysis is to rely upon the symbolic program execution to generate new input values, and maximize code coverage. Multiple tools have since been developed to implement this approach, [9], [10], [11]. In addition to exploring new execution paths, symbolic execution has a number of applications in the context of software development and testing including data structure repair [12] and finding differences between implementations [13], amongst other things. In the context of this approach, the use of constraint solving techniques to generate input values was proposed in [14] and [15].

In contrast with the software presented so far in the literature, what sets apart the software developed in this study is that it is applied to execution traces, rather than source or assembly level code. The use of assembly trace was motivated by our desire to incorporate symbolic analysis into a suite of in-house code analysis tools that operate on such traces. This choice was meant to allow us to perform symbolic execution even in cases where both the code and the binaries are unavailable and allows us to circumvent certain difficulties inherent in the analysis of programs, such as the impossibility of assigning an abstract value to the result of some computations like hash functions. Conversely, applying symbolic analysis on traces raises its own difficulties, particularly with respect to memory management and the size optimization of the symbolic constraints, as detailed in section II-B.

## V. Conclusion and Future Work

The main limitation of the interpreter is that the process by which a given conditional jump or string manipulation instruction is selected for further analysis is not yet automated. The assembly-level trace of even a short program may contain several thousand instructions of interest. We currently select the instructions on which we focus the symbolic execution by an informal analysis of the underlying source-code. Observe that the assembly instructions that correspond to a single source code instruction may occur multiple times (indeed, infinitely often) in the corresponding trace.

As we discussed in section III, we believe that introducing a notion of exploitability is a promising solution to this problem. Intuitively, we aim to express the possibility of a vulnerability using symbolic constraints, which would be checked using the interpreter. The constraints that detect stack overflows discussed in section II are an initial example of this type of constraints. Once incorporated into the architecture, trace analysis will comprise two phases: an initial phase in which instructions of potential interest are identified and a second phase in which new initial program values are generated so that we may examine the behavior of interest in a real execution. We are currently pursuing this avenue of research, as well as continuing to extend and optimize the interpreter.


## Acknowledgements

I wish to thank Frederic Painchaud and Richard Khoury for insightful remarks during the development of this research.


## References


[1] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for dpll(t)," in *Proceedings of the 18th International Conference on Computer Aided Verification*, ser. CAV'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 81–94.

[2] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," in *Communications of the ACM*, Feb. 2014, pp. 74–84.

[3] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[4] C. S. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *STTT*, vol. 11, no. 4, pp. 339–353, 2009.

[5] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe, "Extended static checking," 1998.

[6] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé, "Using symbolic execution for verifying safety-critical systems," in *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-9. New York, NY, USA: ACM, 2001, pp. 142–151.

[7] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Softw. Pract. Exper.*, vol. 30, no. 7, pp. 775–802, Jun. 2000.

[8] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 213–223.

[9] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ESEC/SIGSOFT FSE*, 2005, pp. 263–272.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 322–335.

[11] N. Tillmann and J. de Halleux, "Pex - white box test generation for .net," in *Proc. of Tests and Proofs (TAP'08)*, ser. LNCS, vol. 4966. Prato, Italy: Springer Verlag, April 2008, p. 134153.

[12] S. Khurshid, I. García, and Y. L. Suen, "Repairing structurally complex data," in *SPIN*, ser. Lecture Notes in Computer Science, P. Godefroid, Ed., vol. 3639. Springer, 2005, pp. 123–138.

[13] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song, "Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation," in *In In Proceedings of the 16th USENIX Security Symposium (Security07*, 2007.

[14] A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques," in *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '98. New York, NY, USA: ACM, 1998, pp. 53–62.

[15] N. Gupta, A. P. Mathur, and M. L. Soffa, "Automated test data generation using an iterative relaxation method," in *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '98/FSE-6. New York, NY, USA: ACM, 1998, pp. 231–244.