A Hybrid Aproach for Control Flow Graph Construction from Binary Code

Minh Hai Nguyen[1], Thien Binh Nguyen[2] and Thanh Tho Quan[3]
Faculty of Computer Science and Engineering
Hochiminh City University of Technology
Hochiminh City, Vietnam
[1]551307910@hcmut.edu.vn,
[2]551105019@stu.hcmut.edu.vn, [3]qttho@cse.hcmut.edu.vn

Mizuhito Ogawa
School of Information Science,
Japan Advanced Institute of Science and Technology
Ishikawa, Japan
mizuhito@jaist.ac.jp

***Abstract*—Binary code analysis has attracted much attention. The difficulty lies in constructing a *Control Flow Graph* (CFG), which is dynamically generated and modified, such as mutations. Typical examples are handling dynamic jump instructions, in which destinations may be directly modified by rewriting loaded instructions on memory. In this paper, we describe a PhD project proposal on a hybrid approach that combines static analysis and dynamic testing to construct CFG from binary code. Our aim is to minimize false targets produced when processing indirect jumps during CFG construction. To evaluate the potential of our approach, we preliminarily compare results between our method and Jakstab, a state-of-the-art tool in this field.**

Keywords: binary code analysis, static analysis, dynamic analysis, SMT, symbolic execution, control flow graph construction

I. Introduction

There are several reasons to choose binary code as a program analysis target. First, once source codes are lost or unavailable, we need to directly analyze binary codes. Third party modules and computer virus are such examples. Second, a serious issue emerges from compiling from source codes to binary codes. A compiler may remove certain behaviors of programs, hence altering its contents or even its semantics [1].

Recently, there are a lot of tools and prototypes introduced for analyzing binary code. BINCOA [2] offered a framework for binary code analysis. Its core technology is a refinement-based static analysis [6] by abstract interpretation [7]. IDA Pro [3] is commercial software, which has been used in many binary analysis platforms. Remarkably, Jakstab [4][5] is a state-of-the-art tool in the field of binary code analysis. It translates binary codes to a low level intermediate language in an on-the-fly manner and performs further analysis accordingly.

Fig.1 shows four major steps. The first step translates binary codes to disassembly codes. The second step builds an *intermediate representation* (IR) from the disassembly codes. The third step constructs the *Control Flow Graph (*CFG), whose vertices represent basic blocks of instructions and directed edges represent jumps in control flow [10]. Based on the constructed CFG, other analysis utilities like malware detection or security checking will be further provided.

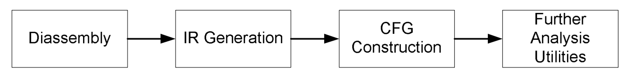

Fig. 1. Common steps for binary code analysis.

In Fig.1, the CFG construction step plays an essential role. Whereas CFG construction at an imperative language level is a classic work, that at the binary code level still remains a challenging task due to the following obstacles.

The first challenge lies in *Complex Instruction Set Computer* (CISC) [11] architectures, such as *x*86. They have very rich instruction sets, e.g., hundreds of instructions and thousands of operands combination in *x*86 architecture [12]. All of them must be interpreted properly to construct a CFG. The second challenge lies in the lack of desirable properties of *high level semantic structure*. For instance, there are no function abstraction and/or type at binary code level. Moreover, the issues of *code and data ambiguity*, *indirect branches* and *overlapping instructions* [13] are also burden. Most of existing tools use static analysis with an over-approximation, resulting in a CFG with more false targets.

Inspired by [29], this paper proposes a hybrid approach which combines static analysis and dynamic testing for generating CFG from binary codes. We apply standard intra-procedural CFG generation until indirect jumps and/or function calls occur. Then, test data are generated to decide their precise destinations. Different from [29], we apply symbolic execution to generate appropriate test data. This hybrid method is neither sound nor complete, but will give a practically more precise CFG (even with mutation), compared to abstract interpretation based static analysis.

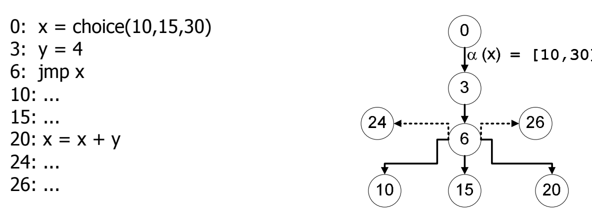

Fig. 2. CFG reconstructed by over-approximation abstraction

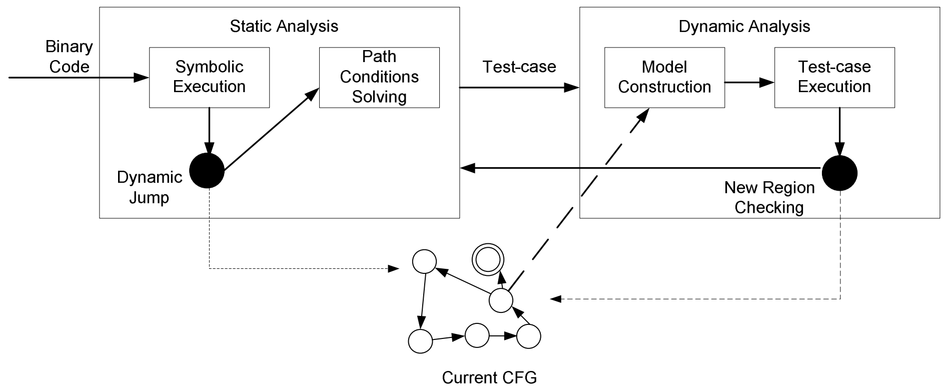

Fig. 3. The framework of combining static and dynamic analysis

The rest of the paper is organized as follows. Section II briefly describes a motivating example which shows problems of an over-approximation approach. Section III illustrates the overview of our hybrid framework. Section IV discusses in more detail our running examples to clarify the advantages of our method. Section V illustrates our research challenges in the subsequent PhD project. Section VI shows our preliminary evaluation. Related works are presented in Section VII, and Section VIII concludes the paper.

II. MOTIVATING EXAMPLE

Fig.2 presents an example illustrating the drawback of the over-approximation approach. We consider a code fragment starting at *Instruction* 0, where variable x is given a value randomly picked up from a set of {10,15,30}. When we convert this program into an abstract form, a typical approach is to use an interval to represent possible values of variables. In this case, the abstract value of x, denoted as $\alpha(x)$, is represented as an interval of [10, 30].

The major problem occurs when value of x is used as the target address of an *indirect jump* instruction at line 6. In the abstract program, since x can take any values in the interval of [10, 30], there are several other *false branches* which may be produced, illustrated as the dotted arrows in Fig.2. They come from an *over-approximation* based abstract interpretation.

There are many approaches to give a better abstraction. The false branch problem of an over-approximation is inevitable, and this is crucial for CFG generation of binary codes. For instance, mutation tries to lead such false branching. This issue motivates us to consider a new hybrid approach.

III. THE PROPOSED FRAMEWORK

Fig.3 describes our framework, which consists of two phases: *Static Analysis* and *Dynamic Analysis*. They are executed alternatively until the CFG converges.

In this framework, a program to be analyzed is divided into *regions.* Each region is a block of instructions which contains no dynamic jump instructions. In the *Static Analysis* phase, we apply *Symbolic Execution* (SE) [14] to reconstruct execution paths in one region and create the corresponding sub-CFG. This process of SE is performed in forward manner until encountering an indirect branch.

When encountering a dynamic jump, we execute *Path Condition Solving* to solve path conditions associated with an execution path in the current region. Then, test-cases are generated to cover all execution paths. In the meantime, the sub-CFG of the current region will be updated.

Subsequently, the *Dynamic Analysis* phase will be executed. In this phase, firstly the *Model Construction* converts the CFG into an intermediate labeled transition system (LTS). The test-cases are executed in this intermediate LTS as *Test-case Execution* step. It allows us to verify real targets of dynamic jumps, which update the current CFG. If they jump into new areas, which are not explored yet, the Static Analysis phase is invoked again. Such combination of static and dynamic analysis is repeated until no new areas are discovered.

IV. EXAMPLES

Example I: Handling dynamic jump

Fig.4 shows our first example, which starts at *start* and introduces an indirect jump at *Instruction* 8. By static analysis, two execution paths leading to this dynamic jump are easily determined, i.e., P_1 = (*star*→0→1→5→6→7→8) and P_2 = (*start*→0→1→2→3→7→8). For an initial value α of register *eax*, the path conditions of P_1 and P_2 are evaluated to ($\alpha < 0$) and ($\alpha >= 0$), respectively.

We apply a *prover* to generate two test-cases corresponding to these path conditions, say, α_1=-1 and α_2=2. By executing the program with them, sound targets of indirect branches are determined to be *start* and *Instruction* 6. By continuing the path execution from *Instruction* 6, *Instruction* 4 is discovered as a new target of the indirect jump at *Instruction* 8. The full CFG is illustrated in Fig.4, where the dotted arrows indicate new edges discovered by Dynamic Analysis phase.

Example II: Combination of multiple regions and handling dead code

In this example, we extend Example I to illustrate a more complex scenario. Table 1 presents the program to be analyzed, which contains two indirect jumps at *Instruction* 7 and *Instruction* 16. In addition, this code uses an *obfuscation* by inserting *dead code* from *Instruction* 17 to *Instruction* 19.

This example describes our idea that each executed dynamic jump creates a new *region* in the program. Figure 5 illustrates our construction of CFG complying with this strategy. First, the CFG of *Region* 1 (corresponding to the code from *Instruction* 0 to *Instruction* 7) is extracted by the method as described in Example I. By generating test-cases and executing the indirect jump at *Instruction* 7, we discover a new region starting at *Instruction* 8.

```
start: ;reset eax
  0: cmp eax, 0
  1: jl lthen
lelse:
  2: mov eax, offset start + 1
  3: jmp lcont
lhalt:
  4: halt

lthen:
  5: mov eax, offset l1 + 4
l1:
  6: sub eax, 3
lcont:
  7: sub eax, 1
  8: jmp eax
```

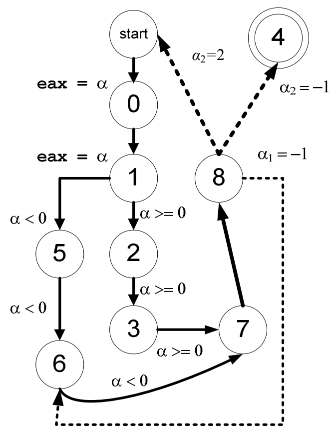

Fig. 4. CFG reconstructed by over-approximation abstraction.

It must be remarked that if we apply abstract interpretation, one typical method is to construct an interval covering all possible targets of *Instruction* 7, e.g., *start* and *Instruction 8*. Thus, it results the interval [*offset lstart*, *offset l1* + 12] as their possible addresses. Since this interval also includes addresses of dead code instructions, the analysis generates false jumps from *Instruction* 7 to this dead code, as described in Fig.5.

Table 1 – A binary code consisting of multiple dynamic jumps and dead code

```
start: ;Entry point
  0: cmp eax, 0
  1: jl lthen1
lelse1:
  2: mov ax, offset start – 1
  3: jmp lcont1

lthen1:
  4: mov ax, offset l1 – 12
l1:
  5: add eax, 11
lcont1:
  6: add eax, 1
  7: jmp eax

;;;;;; Dead code;;;;
 17: cmp ebx,eax
18: jz l4
19: jmp eax
;;;;;

 8: cmp ebx, 0
 9: jl lthen2
lelse2:
 10: mov eax, offset lstart + 1
 11: jmp lcont2
lhalt:
 12: ret

lthen2:
 13: mov eax, offset l2 + 6
l2:
 14: sub eax, 5
lcont2:
 15: sub eax, 1
 16: jmp eax
```

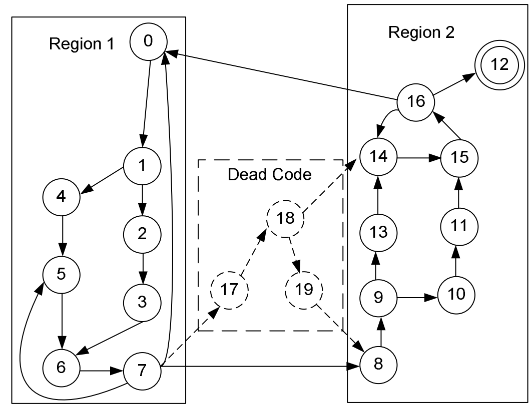

Fig. 5. Inter-region strategy of CFG construction

Using test-case as in our method, this dead code will never be explored since there are no real execution paths through it. Instead, the Static Analysis phase is invoked again to continue analyzing the source code from *Instruction* 8 to *Instruction* 16 and generates the corresponding CFG for *Region* 2. When performing Dynamic Analysis for the indirect branch at *Instruction* 16, there are two possible targets which are addresses of *Instruction* 0 and *Instruction* 12. Hence, we add a new edge from vertex 16 to vertex 0. Finally, since there are no newly-discovered regions, it stops and the final CFG is generated by combining Region 1 and Region 2.

V. RESEARCH CHALLENGES

To fully implement our suggested framework, we need to concern following research challenges.

The first challenge is to *handle path conditions* associated to each execution path of source binary code during symbolic execution. Provers (SMT) solve the path conditions for test-case generation. The challenge encountered here is the computational limitation of provers. Current provers mostly cover only linear constraints for arithmetic. At binary code level, the types are arithmetic and the challenge lies in non-linear constraints, such as *Z3.4.3* [20] and *raSAT* [18].

The next challenge is to *infer loop invariants*. This is a classic issue, and recently two methodologies (and their combinations) are popular. (1) *Loop invariants in arithmetic.* For a linear loop invariant, the technique based on *Farkas' lemma* [15] is common. For non-linear equational invariants, an algebraic method is known [33]. (2) *Loop invariants in first-order logic*. *Craig Interpolation* is known to be a good strategy to produce loop invariants [19].

The last challenge is to *simulate the program execution by* Dynamic Analysis *on the current CFG*. We intend to apply model checking, since the conversion from a CFG to a *Labeled Transition System* (LTS) is fairly straightforward. The key is, how to make model checking terminate, since test data are in an infinite domain, such as *Integers*. Currently, we use Promela of SPIN, which accepts arithmetic expressions and generates a model in an on-the-fly manner. We set the termination condition as to either reach to a final state or find a new target destination of an indirect jump. This does not guarantee termination, e.g., a loop that contains operations to increment. If it fails, we simply apply time-out. Alternatively, if an LTS gets stuck with a certain input value, i.e. it cannot determine the destination of a transition, the original CFG needs to be enlarged with that input value.

VI. SMALL EXPERIMENTS

Small experiments are performed to evaluate the feasibility of our method with 5 example programs[1] (some shown in Table 2) under the following constraints: (i) the code contains indirect jumps and (ii) the loop conditions are linear, which allowed us to handle them using Farkas' lemma. These experiments have been carried out in a Core i5-3340M computer with 4GB RAM. Our experiments are carried out in the following steps:

(1) From given assembly code, we produce an initial CFG that consists only program entry.

(2) We perform the intra-procedural CFG construction (or *static CFG construction* since we only process static jump instruction in this step) such that a CFG have program instructions as its vertices. There are 6 kinds of vertices defined, including *Start*, *Exit*, *Condition*, *Join*, *Loop* and *Other Instruction*. *Other Instruction* vertices cover the arithmetic instructions and move instructions of the assembly code.

We apply Jakstab to construct intra-procedural CFG. By default, Jakstab implements an on-the-fly method of static analysis on binary source code. Once encountering indirect branches, it applies abstract interpretation in order to resolve possible target addresses. Since Jakstab is an open source software, we automatically replace this step as to stop Jakstab when a dynamic jump is found, and apply step (3).

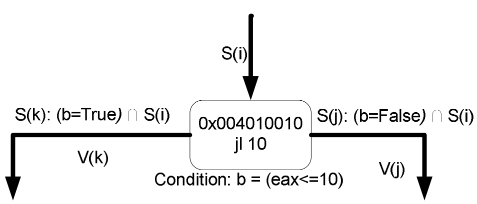

Fig. 6. Generating symbolic conditions for a *Condition* vertice

(3) We perform the symbolic execution on the current CFG. In order to do that, we build a simple symbolic execution framework to handle a subset of x86 instructions. For each edge in the CFG, we compute a symbolic condition, which is a necessary condition to have an execution path through this edge. For instance, Fig.6 illustrates the symbolic conditions generated when handling a *Condition* vertex of the CFG. Symbolic conditions for *Start*, *Exit*, *Condition*, *Join*, and *Other Instruction* vertices are straightforward. For *Loop* vertices, Farkas' lemma is used to infer a loop invariant.

(4) We apply *Z3.4.3* [20] to solve path conditions (given in step (3)) which are associated to paths reaching to the indirect jump vertices, and to generate test-cases to cover them.

(5) We use PAT [17] to generate a LTS from the current CFG and dynamically perform the test-cases on it. After PAT is performed, the outputs of PAT for estimating the targets of the indirect jumps. Then, the CFG is enlarged with the estimated targets.

(6) If the CFG is enlarged in step (5) with fresh vertices, return to step (2). Otherwise, the construction finished.

Fig.7 illustrates CFGs generated by Jakstab and our method in one testing program. The average runtime of Jakstab to process a program is less than one seconds. Although the test programs are just toy programs, Jakstab still fails to resolve the target addresses of dynamic jumps. For the program in Table 2, the CFG generated by Jakstab stopped at the indirect jump at location 21. Using our approach, the analysis process proceeds and achieves the full CFG.

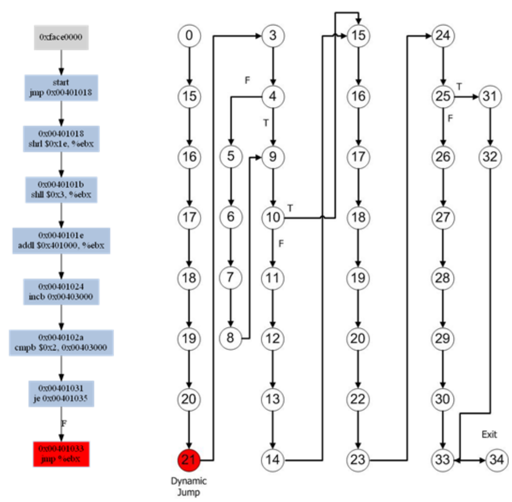

Fig. 7. The CFGs generated by Jakstab (left) and by our method (right)

Program				Jakstab			Hybrid Method		
Name	Src	Inst	NumDJ	J-Inst	Cvrg	Time	H-Inst	Cvrg	Time
demo1	asm	11	1	10	90%	0.1s	11	100%	2.1s
demo2	asm	38	2	8	21%	0.14s	23	60%	3.1s
demo3	asm	35	3	5	14%	0.12s	35	100%	1.4s
demo4	asm	48	3	11	22%	0.14s	12	25%	1.5s
demo5	asm	48	4	5	10%	0.12s	43	90%	3.5s

Table 3 - Experimental results

Table 3 gives the summary of our experiments. In this table *Inst* implies the number of instructions in the original program; *J-Inst* and *H-Inst* are the numbers of intructions actually reachable using Jakstab and our approach respectively; for reader convenience, we also compute the value of *Cvrg*, which implies the percentage of instruction coverage by the generated CFGs. One can observe that our

[1] Interested readers can download those 5 sample programs at http://cse.hcmut.edu.vn/~save/doku.php?id=project:start

approach can significantly improve the quality of the generated CFG, as compared to Jakstab. However, our approach suffers from higher computational cost, thus consuming more execution time.

VII. Related Works

1. Hybrid approaches for program analysis

The approach of using hybrid method by combining static analysis and dynamic testing to analyze imperative program has been considered in many related works. In the field of software testing, *concolic testing* [21][22][23][24] is a well-known technique, which combines symbolic execution and dynamic execution to generate test-case.

```
.data
        counter db 0h
   .code
start:    ; Entry point
  0:  jmp l2
  1:  inc edi          ;dead code
  2:  mov edi, 1       ;dead code
  3:  cmp al, 2
  4:  jle l1
  5:  nop
  6:  nop
  7:  nop
  8:  nop
l1:
  9:  cmp al, 2
 10:  jge l2
 11:  nop
 12:  nop
 13:  nop
 14:  nop
l2:
 15:  shr ebx, 30
 16:  shl ebx, 3
 17:  add ebx, 401000h
18:  inc counter

19:  cmp counter, 2
 20:  je l3
 21:  jmp ebx

l3:
 22:   shr eax,31
 23:   add eax, 401043h
 24:   cmp ebx,eax
 25:   jz l4
 26:   jmp eax

 27:   nop
 28:   add eax,ebx
 29:   sub ebx,eax
 30:   jmp l5
l4:
 31:  add ebx,eax
 32:  sub eax,ebx
l5:
33:   xor eax,eax
34:   invoke ExitProcess,0

end start
```

Table 2 – Source code of the experimental file

SLAM tool [25] is based on an automatic analysis of client code to validate a set of properties or find a counter-example showing a fail execution. DART [26] provides a new approach for completely automatic unit testing for software to avoid stubs that simulate the external environment of software. SYNERGY algorithm [27] presents a new approach to combine static and dynamic program analysis for property checking and test generation. DUALYZER is a dual static analysis tool [28], which is based on only over-approximation for both proving safety and finding real bugs.

OSMOSE [31] is a tool which also applies the concolic testing technique for automatic test-case generation from binary programs. This approach is quite close to our works, but it aims at generating test-cases, rather than CFG construction. Furthermore, OSMOSE involves solver to solve virtually all of path conditions of execution paths of the program. Meanwhile, our approach only invokes solver when handling execution paths leading to dynamic jumps, thus saving remarkable computational cost.

2. CFG construction from binary code

There are many methods of extracting CFG from binary source code. Gogul Balakrishnan and Thomas Reps introduced value-set analysis (VSA) [9]. By using numeric and pointer-analysis algorithm, VSA computes an over-approximation of the set of numeric values or addresses that every location may hold. This analysis technique was implemented in a tool called CodeSurfer [8][9], which is an extension from IDAPro [3].

Combining static and dynamics analysis for malwares is introduced in [29]. Regardless of over- or under-approximation, static analyses cannot resolve targets of indirect jumps when dynamic code modifications occur, e.g., mutations. Numerical abstract domains, such as intervals and *k*-sets, are used to handle targets of jumps, but hard to satisfy both accuracy and complexity. Recently, a refinement-based method is proposed based on *k*-sets [6]. Due to its cardinality bound, this method still remains certain limitations.

In BINCOA, a dynamic symbolic execution [14] and bit-vector constrain solving [30][31] are introduced. Meanwhile, IDA Pro relies on linear sweep decoding (the method of brute force decoding all addresses) and recursive traversal method [32] (decoding recursively until an indirect jump is found) for disassembly, which make it difficult to scale.

3. Binary analysis based on model-checking

Beside abstracting the memory addresses to reconstruct a CFG, another approach is to describe malicious behavior of functions using temporal logic. This reduces virus detection to model checking. An extension CTPL of CTL (Computation Tree Logic) is proposed for to specify certain obfuscation actions of a virus [36][37][38]. Further, Song and Touili extend CTPL to SCTPL for better description on stack-based actions of viral behaviors [34]. Recently, LTL (Linear Temporal Logic) is suggested to replace CTL, and SLTPL is introduced [35]. They mostly consider the situation that a reasonably precise CFG are statically computed, say, without mutations.

VIII. Conclusion

This paper preliminarily reports a proposal for PhD work. The initial goal of this work is to produce a more precise CFG from binary codes. The difficulty to decide the precise destinations of indirect jumps remains as a major problem in the field. We proposed a hybrid approach, which combines an over-approximation by static analysis and an under-approximation by dynamic testing to achieve practically more accurate CFGs. Initial results show that our method is quite promising. We expect that our approach not only resolves the issue of indirect jumps, but also improves efficiency of analyses on binary source codes.

Acknowledgement

This research is funded by Vietnam National University Hochiminh City (VNU-HCM) under grant number 01/BK/2013/ 911VNUHCM-JAIST.

REFERENCES

[1] G. Balakrishnan, T. Reps, D. Melski and T. Teitelbaum. What You See Is Not What You eXecute. In *Journal ACM Transactions on Programming Languages and Systems*. Lecture Notes in Computer Science, Springer, pp. 202–213. 2005.

[2] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary and A.Vincent. The BINCOA Framework for Binary Code Analysis, In *Proceedings of the 23rd International Conference of Computer Aided Verification (CAV 2011), pp.165-170*. 2011.

[3] IDAPro disassembler, http://www.datarescue.com/idabase/

[4] J. Kinder and H. Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*. Vol. 5123, Lecture Notes in Computer Science, Springer, pp. 423–427. 2008.

[5] J. Kinder, H. Veith and F. Zuleger. An Abstract Interpretation–Based Framework for Control Flow Reconstruction from Binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009*). Vol. 5403, Lecture Notes in Computer Science, Springer, pp. 214–228, 2009.

[6] S. Bardin, P. Herrmann and F. V´edrine. Refinement-based CFG Reconstruction from Unstructured Programs. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011).* 2011.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages. 1977.*

[8] G. Balakrishnan, R. Gruian, T. Reps and T. Teitelbaum. CodeSurfer/x86 – A Platform for Analyzing x86 Executables. In *Proceedings of the 14th International Conference on Compiler Construction (CC 2005)*. Vol. 3443. LNCS. Springer, pp. 250–254. 2005.

[9] G. Balakrishnan and T. Reps. Analyzing Memory Accesses in x86 Executables. In *Proceedings of the 13th International Conference on Compiler Construction (CC 2004)*. Vol. 2985. LNCS. Springer, pp. 5–23. 2004.

[10] F. Allen. Control flow analysis. SIGPLAN Notices 5 (7): 1–19. 1970.

[11] S. Andrew. Structured Computer Organization. Pearson Education, Inc. Upper Saddle River, NJ. 2006.

[12] Intel 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation. 2009.

[13] J. Kinder. Static Analysis of x86 Executables, Phd Thesis, Technische Universitat Darmstadt. 2010.

[14] J. King and T. Watson. Symbolic execution and program testing. In *Communications of the ACM Volume 19 Issue 7*, pp. 385–394. 1976.

[15] M. Colón, S. Sankaranarayanan and H. Sipma. Linear Invariant Generation Using Non–linear Constraint Solving. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*. LNCS 2725, Springer–Verlag, pp. 420–433. 2003.

[16] N. Nguyen, T. Quan, P. Nguyen and T. Bui. COMBINE: A Tool on Combined Formal Methods for Bindingly Verification. In *Proceedings of the 8th Internationl Symposium on Automated Technology for Verification and Analysis (ATVA 2010)*. Singapore, Springer Verlag, ISBN-10 3-642-15642-8, ISBN-13 978-3-642-15642-7. 2010.

[17] J. Sun, Y. Liu, J.S. Dong and J. Pang. PAT: Towards Flexible Verification under Fairness. In *Proceedings of the 21th International Conference on Computer Aided Verification (CAV 2009)*, Grenoble, France, June, 2009.

[18] K. To and M. Ogawa. SMT for Polynomial Constraints on Real Numbers, *Tools for Automatic Program Analysis TAPAS 2012* , Elsevier ENTCS vol.289, pp.27-40. 2012.

[19] J. Esparza, S. Kiefer and S. Schwoon. Abstraction Refinement with Craig Interpolation and Symbolic Pushdown Systems. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of System (TACAS 2006*). *2006.*

[20] Z3: An Efficient SMT Solver, http://z3.codeplex.com/

[21] N. Williams, B. Marre, P. Mouy and M. Roger. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In *Proceedings of the 5th European Dependable Computing Conference*, pp. 281–292. 2005.

[22] K. Sen and G. Agha. CUTE and jCUTE : Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*., pp. 419-423. 2006.

[23] P. Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd International Workshop on Random testing*, pp. 1-1. 2007.

[24] N. Beckman, A. Nori, K. Rajamani, R. Simmons, S. Tetali and A. Thakur. Proofs from Tests. *IEEE Transactions on Software Engineering*. 2012.

[25] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the SPIN 2001 Workshop on Model Checking of Software*, pp. 103-122. 2001.

[26] P. Godefroid, N. Klarlund and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation,* pp. 213-223. 2005.

[27] B. Gulavani, T. Henzinger, Y. Kannan, A. Nori and S. Rajamani, Synergy: A New Algorithm for Property Checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of software engineering.* 2006.

[28] C. Popeea and W. Chin. Dual analysis for proving safety and finding bugs. In *Proceedings of the 2010 ACM Symposium on Applied Computing,* pp. 2137-2143. 2010.

[29] T. Izumida, K. Futatsugi and A. Mori. A Generic Binary Analysis Method for Malware. In *Proceeding of the 5th International Workshop on Security.* 2010.

[30] S. Bardin and P. Herrmann. Structural Testing of Executables. In *IEEE ICST 2008. IEEE Computer Society, Los Alamitos.* 2008

[31] S. Bardin and P. Herrmann. OSMOSE: Automatic Structural Testing of Executables. In *International Journal of Software Testing, Verification and Reliability (STVR), 21(1).* 2011.

[32] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security (ACM 2003)*, pp. 290–299. 2003.

[33] S, Sabjarabaratababm, G,B, Suonam, and Z. Manna, Non-linear loop invariant generation using Grobner Bases, *ACM Princeples of Programming Languages (POPL 2004)*, pp.318-329, 2004.

[34] F. Song and T. Touili, Pushdown Model Checking for Malware Detection. In *Proceedings of TACAS. 2012*, 110-125.

[35] Fu Song, Tayssir Touili: LTL Model-Checking for Malware Detection. In *Proceedings of TACAS 2013*, 416-431.

[36] Holzer, A., Kinder, J., Veith, H.: Using Verification Technology to Specify and Detect Malware. In: *Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2007.*LNCS, vol. 4739, pp. 497–504. Springer, Heidelberg (2007)

[37] Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting Malicious Code by Model Checking. In: *Julisch, K., Krgel, C. (eds.) DIMVA 2005*. LNCS, vol. 3548, pp. 174–187. Springer, Heidelberg (2005)

[38] Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Proactive detection of computer worms using model checking. *IEEE Transactions on Dependable and Secure Computing 7(4) (2010)*