A Hybrid Aproach for Control Flow Graph Construction from Binary Code

Minh Hai Nguyen
Department of Comp. Sci. and Eng.
Hochiminh City Uni. of Industry
Hochiminh City, Vietnam
nguyenmhai1984@gmail.com

Thien Binh Nguyen[1] and Thanh Tho Quan[2]
Department of Comp. Sci. and Eng.
Hochiminh City Uni. of Tech.
Hochiminh City, Vietnam
[1]binhnguyenthien@gmail.com,
[2]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*—Nowadays, binary code analysis has been attracting much attention. There are many frameworks and tools introduced to perform the analysis of binary code. 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 which discusses a hybrid approach of combining static analysis and dynamic testing to construct CFG from binary code. By using this method, we aim at minimizing false targets produced when processing indirect jumps in during CFG construction process. To verify the potential of our approach, we have preliminarily compared our resultant CFGs generated after analyzing dynamic jumps with those by Jakstab, a state-of-the-art tool in the field of binary code analysis.**

Keywords: binary code analysis, static analysis, dynamic analysis, hybrid approach, control flow graph construction

I. INTRODUCTION

Nowadays, software is being used all over the world in most of aspects of daily life. The number of new programs has been increased dramatically, which raises a growing need for tools to check whether those programs are correct or not, to find out how long they terminate or to understand whether they compromise our system security. This program analysis can performed from human-readable source code of high level languages or from low level binary code.

There are several reasons for choosing binary code as an attractive target. First, once source codes are missing or unavailable, we need to directly analyze them. For instance, third party modules and computer virus are such examples. Second, a serious issue emerges from the compiling from source code to binary code. During compilation process, a compiler can remove some behaviors of programs, hence altering its contents or even its semantics [1].

In recent years, there are a lot of tools and prototypes introduced for analyzing binary code. BINCOA [2] offered a framework for binary code analysis. The core technology of this framework is the refinement-based static analysis [6] which performs the technique of abstract interpretation [7]. IDA Pro [3] is a closed-source 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 which translates binary code to a low level intermediate language in an on-the-fly manner and performs further analysis accordingly.

Even though introducing various approaches, most of the prominent tools dedicated for binary code analysis more or less adopt a similar framework as presented in Figure 1. There are four major steps in this framework. The first step is to translate the binary code to disassembly code. The second step is to build an *intermediate representation* (IR) from the disassembly code. The next step is to construct the *Control Flow Graph (*CFG) of the program. Basically, CFG is a graph whose vertices represent basic block 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 can be further provided.

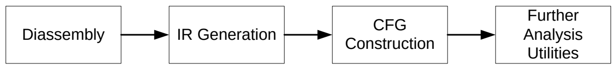

Fig. 1. Common steps for binary code analysis.

In Figure 1, the CFG construction step plays an essential role in any binary code analysis framework. However, whereas CFG construction from source code of imperative languages has become a classic work, performing this task at binary code level still remains a challenging task due to the following obstacles.

The first challenge comes from *Complex Instruction Set Computer* (CISC) [11] architectures, such as *x*86, which supply a very rich instruction sets including a large number of instructions. There are hundreds of instructions and thousands of possible operands combination in *x*86 architecture [12]. All of them should be interpreted properly to construct a precise CFG of programs. The second challenge comes from the lack of many properties of *high level semantic structure*. 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 a big hurdle preventing researchers from approaching the method of analyzing binary files.

To overcome this problem, most of existing tools use static analysis with an over-approximation technique, resulting in an even more over-complex CFG. In this paper, we propose a hybrid approach which combines static analysis and dynamic testing for generating CFG from binary code, inspired by ideas in [29]. We apply standard intra-procedural CFG generation until indirect jumps and/or function calls occur. Then, test data will be generated to decide precise locations of the destinations of indirect jumps and function calls. Different from [29], we apply symbolic execution to generate appropriate test data when indirect jumps and calls occur. This method is neither sound nor complete, but we can expect practically more precise CFG (even with mutation), compared to abstract interpretation based static analysis.

The remainder of the paper is organized as follows. In Section II, we briefly describe our motivating example which analyzes the problem of over-approximation approach. In Section III, we illustrate 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 which are needed to be addressed for the subsequent PhD project. Section VI explains our initial small experiments. Our related works are presented in Section VII. Finally, Section VIII is the conclusion of the paper.

II. Motivating Example

Figure 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 *α(x)*, is represented as an interval of [10, 30].

```
0:  x = choice(10,15,30)
3:  y = 4
6:  jmp x
10: ...
15: ...
20: x = x + y
24: ...
26: ...
```

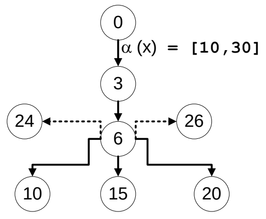

Fig. 2. CFG reconstructed by over-approximation abstraction

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* may possibly be produced, illustrated as the dotted arrows in Figure 2. It is because the abstraction technique is using an *over-approximation* approach, i.e. the abstract value will produce more unnecessary concrete values when the *re-concretization* process is applied.

There are many approaches proposed to give a better abstraction technique. However, most of them still rely on over-approximation, i.e. suffering from the false branch problem. This issue motivates us to consider a new approach, which is subsequently presented.

III. The Proposed Framework

Figure 3 describes our suggested framework. This framework composes of two main phases: *Static Analysis* and *Dynamic Analysis*. These two phases will be executed respectively until the full CFG of this program is constructed.

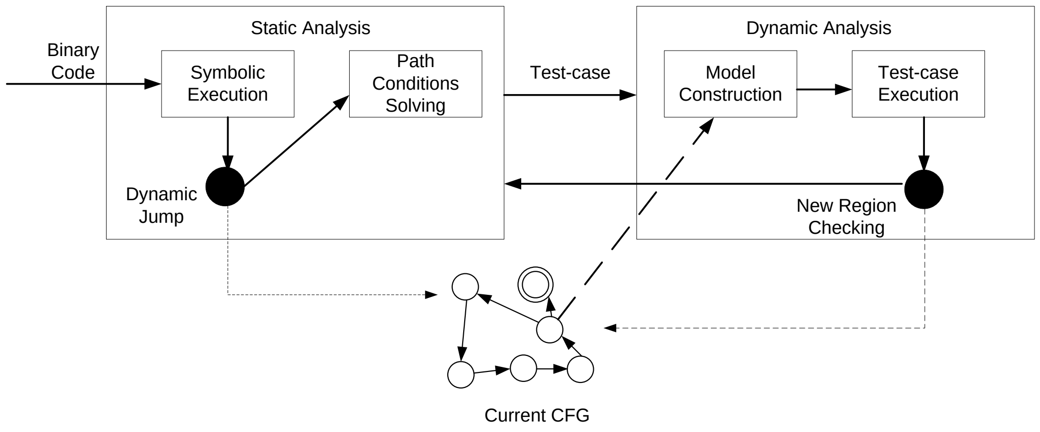

Fig. 3. The framework of combining static and dynamic analysis

In this framework, the program to be analysis will be divided into many *regions.* Each region is a block of instructions which contains no dynamic jump instruction. In *Static Analysis* phase, we use the technique of *Symbolic*

Execution (SE) [14] to reconstruct execution paths in one region and create the corresponding sub-CFG of this region. This process of SE will stop when encountering an indirect branch.

When encountering a dynamic jump, we execute *Path Condition Solving* to solve path conditions corresponding to execution path of program in current region and generate test-cases which cover all the execution paths. In the meantime, the sub-CFG of the current region will be updated into the current CFG.

Subsequently, the *Dynamic Analysis* phase will be executed. In this phase, firstly the *Model Construction* will represent the current CFG in the form of a mathematical *model* which allows simulating execution in CFG. Test-cases previously generated will be simulated in this model in *Test-case Execution* step. It allows us to verify real targets of dynamic jumps, which are updated into the current CFG. If this indirect branch jumps to a new area which has not yet been explored, the Static Analysis phase will be invoked again with this newly-discovered region. Such combination of static and dynamic analysis is then repeated until there is no new area discovered.

IV. EXAMPLES

In this section, we introduce two examples to show our approaches.

Example I: Handling dynamic jump

Figure 4 presents our first example where the analyzed program starts at address *start* and introduces an indirect jump at *Instruction* 8. Using the technique of static analysis, we can easily determine that there is two execution paths leading to this dynamic jump, which include $P_1 = (star \rightarrow 0 \rightarrow 1 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8)$ and $P_2 = (start \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8)$. Given that the initial value of register *eax* is α, the path conditions of P_1 and P_2 can be evaluated are ($\alpha < 0$) and ($\alpha >= 0$) respectively.

We use *prover* to generate two test-cases corresponding to these path conditions, which exemplarily are $\alpha_1 = -1$ and $\alpha_2 = 2$. Executing the program which these two test-cases, we practically calculate the real targets of indirect branches for those two cases are *start* and *Instruction* 6. When dynamically following the path execution to *Instruction* 6, we continue discovering *Instruction* 4 as a new target for the indirect jump. Hence, the full CFG of the program is constructed as illustrated in Figure 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. Listing 1 presents the program to be analyzed, which contains two indirect jumps at *Instruction* 7 and *Instruction* 16. In addition, the author of this code also uses an *obfuscation method* 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
```

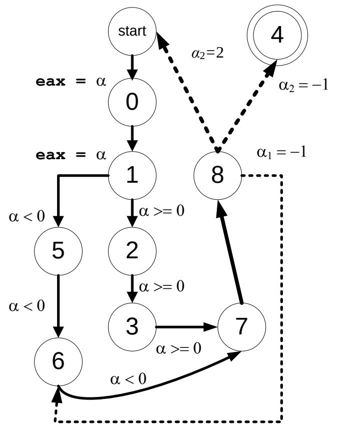

Fig. 4. CFG reconstructed by over-approximation abstraction.

```
start: ;Entry point           8: cmp ebx, 0
  0: cmp eax, 0               9: jl lthen2
  1: jl lthen1              lelse2:
lelse1:                      10: mov eax, offset lstart + 1
  2: mov ax, offset start – 1 11: jmp lcont2
  3: jmp lcont1             lhalt:
                             12: ret
lthen1:
  4: mov ax, offset l1 – 12 lthen2:
l1:                          13: mov eax, offset l2 + 6
  5: add eax, 11            l2:
lcont1:                      14: sub eax, 5
  6: add eax, 1             lcont2:
  7: jmp eax                 15: sub eax, 1
                             16: jmp eax
;;;;;; Dead code;;;;
 17: cmp ebx,eax
 18: jz l4
 19: jmp eax
;;;;;
```

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

It should be remarked that if we apply static abstract interpretation in this case, one typical method is constructing a set of interval covering all possible targets of *Instruction* 7. This technique can be refined to just consider significant values in this set of interval. Hence, the static analysis can determine that the maximum possible value of the register *eax* in *Instruction* 7 when executed is (*offset l1* + 12) (the address of *Instruction* 8) and the minimum value is (*offset lstart*) (the address of *Instruction* 0) (we assume that each construction will take two memory units). Therefore, the abstract interpretation will extract an interval of [*offset lstart*, *offset l1* + 12] as a potential target. Since this interval also includes addresses of dead code instructions, the analysis may generate some false jumps from *Instruction* 7 to this dead code, as described in Figure 5.

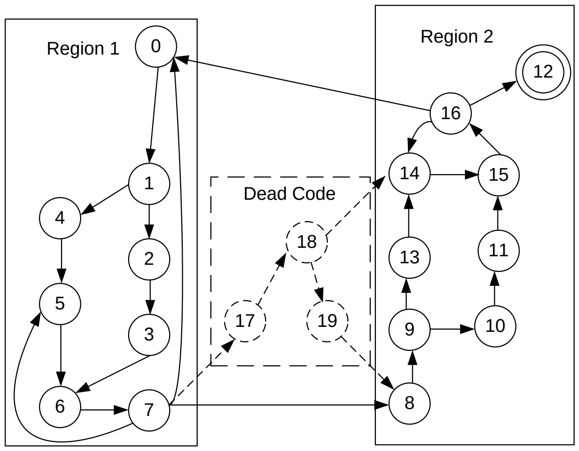

Fig. 5. Inter-region strategy of CFG construction

Using test-case as suggested in our method, this dead code will never be explored since there is no real execution path which goes 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 is no newly-discovered region, the overall analysis stops and the final CFG is eventually 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* corresponding to each of execution path in static analysis of source binary code. First, we must implement a symbolic execution method for binary code. Then, we apply provers (SMT) to 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 solve the problem of *inferring loop invariant in static analysis*. This is a classic issue, and recently two methodologies (and their combinations) are popular as follow: (1) *Loop invariants in arithmetic.* For the linear loop invariant, the technique based on *Farkas' lemma* [15] is common. For non-linear invariants (restricted to equality), an algebraic method is known [33]. (2) *Loop invariants in first-order logic*. It is known that *Craig Interpolation* is a good strategy to produce loop invariants [19].

The last challenge is to *simulate the program execution from the current CFG* in Dynamic Analysis. We are considering model checking for this issue, since once a CFG is obtained, conversion to a *Label Transition System* (LTS) is straightforward. Moreover, the program model generated by model checking can be used for further checking of other properties once the CFG is complete generated.

VI. SMALL EXPERIMENTS

We performed some experiments to evaluate the feasibility of proposed method. We generated 5 testing programs which have 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. The average length of these programs was approximately 35 lines of code. Listing 2 gives an example of these programs.

We have built a simple symbolic execution framework to symbolically handle x86 instructions. The prover used to solve the generated path conditions is *Z3.4.3* [20]. We use PAT [17] to generate the model and dynamically perform the test-case on that. The outputs generated from by Z3 and PAT are combined manually as follows. First, we use symbolic execution to perform the process of constructing CFG from the code until a dynamic jump encountered. Then, this CFG will be converted manually into input of PAT. After that, PAT will be executed to calculate the simulation of this CFG. The output of PAT was logged, from which the targets of the indirect jumps will be estimated manually.

Figure 6 illustrates CFGs generated by Jakstab and our method in one testing program. The average runtime of Jakstab to process a program was less than one seconds. However, even though the testing data are just toy programs, Jakstab still has failed to resolve the target address of dynamic jump. For the program in Listing 2, the CFG generated by Jakstab stopped at the indirect jump at location 21. However, by using our approach of combining static analysis and dynamic testing, we can continue the analysis process and achieve the full final CFG. Our initial experiments show that our method can successfully generate CFGs for all 5 testing programs while Jakstab all failed due to the dynamic jumps.

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.

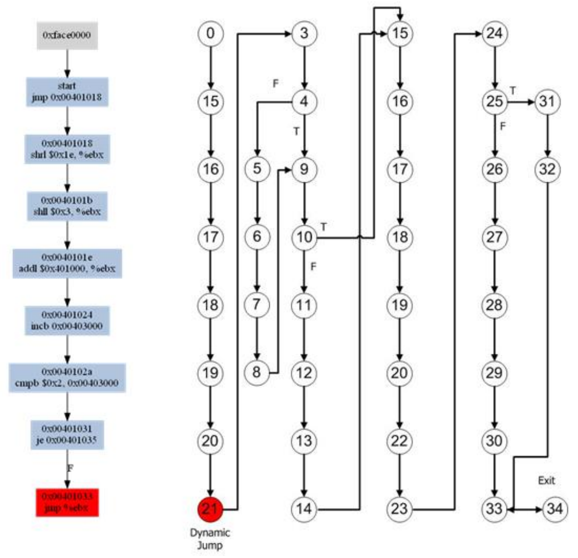

Fig. 6. The CFGs generated by Jakstab (left) and by our method (right)

```
.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
```

Listing 2 – Source code of the experimental file

Moreover, in the field of program verification, there are many frameworks which use the same approaches. 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 where writing test driver and harness code to simulate the external environment of software is unnecessary. SYNERGY algorithm [27] presents a new approach to combine static and dynamic program analysis for property checking and test generation. Alongside this track, DUALYZER is a dual static analysis tool [28] which has a new approach that is to base on only over-approximation for both proving safety and finding real bugs.

2. CFG construction from binary code

There are many method 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].

Combination of static analysis and dynamics analysis is a noticeable approach introduced in [29]. As static analysis techniques are over-approximation or under-approximation, these techniques cannot resolve all targets of indirect jumps. Some numerical abstract domains as interval, *k*-sets, etc. are used handle jump targets, but none of them meet both criteria of the accuracy and the complexity. Recently, there is a refinement-based method [6] that helps to reconstruct CFG more precise. However, using *k*-sets with cardinality bound, this method still fails in many cases.

In the aforementioned tool of BINCOA, a technique based on dynamic symbolic execution [14] and bit-vector constrain solving [30][31] is 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 approached) for disassembly.

3. Binary analysis based on model-checking

Beside the approach of abstracting the memory value to reconstruct the CFG, another approach is to describe malicious functions possibly caused by virus using temporal logic, thus enabling virus detection using model checking. CTL (Computation Tree Logic) provides a solid foundation to formally present system behaviors, based on which CTPL is proposed for to specify some certain obfuscation actions of virus [36][37][38]. Further, Song and Touili extend CTPL as SCTPL to better describe stack-based actions of viral behaviors [34]. Recently, LTL (Linear Temporal Logic) is suggested to replace CTL since LTL can be more precise to capture behaviors of malwares. SLTPL is then introduced to reflect this idea [35]. It can be considered the latest result in this direction achieved so far

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 the binary source code. As the unidentified target of indirect branch remains a major problem in the field of analyzing executable programs, we suggest combining the over-approximation of static analysis with under-approximation of dynamic testing to achieve more accurate results. Initial results showed that our proposed method is quite promising. We expect that our approach does not only help on the issue of indirect branches but also be later improved as a new efficient method for analyzing binary source code.

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)*