CBASS: Cross-platform Binary Automated Symbolic-execution System

Lixin Li, Xing Li and James E. Just
Cyber Innovation Unit, Battelle Memorial Institute
{lil,lix,justj}@battelle.org

***Abstract* – Research on Dynamic Taint Analysis (DTA) and symbolic execution have made great strides in the last ten years, particularly in analyzing x86 binary code. Unfortunately these impressive results are disconnected from the security demands of increasingly popular mobile and embedded systems that typically use non-x86 instruction sets. To support cross-platform security analysis of such devices, we designed and implemented a prototype DTA symbolic execution system that operates directly on an Intermediate Representation (IR) rather than on native instructions. Our initial experiments demonstrated great results on both x86 and ARM architectures. Our system provides the basis for continuing evaluations on other instruction sets.**

I. Introduction

Dynamic Taint analysis (DTA) and symbolic execution has been extensively researched and applied to a variety of security areas ranging from vulnerability discovery in benign software to analyzing malware [1]. While DTA and symbolic execution are complex and tricky to implement, cross-platform DTA design does not seem to have received serious attention.

In analyzing this topic, we identified three key issues that seem to be overlooked in general: (1) Existing DTA implementations are tightly integrated with their instrumentation or emulation environments, usually PIN or QEMU. This tight integration makes it harder to apply DTA directly on a platform-independent Intermediate Representation (IR). (2) DTAs propagate tags directly on one particular target Instruction Set Architecture (ISA) - x86 almost all the cases. Because of the great complexity and differences among ISAs, this approach means that porting an existing DTA implementation to a different ISA requires a total rewrite of existing code, even for the same algorithm. (3) Some DTA systems already used IR in certain phases, but in a somewhat ad-hoc and inconsistent manner. For example, IR is used for symbolic execution just after taint analysis is performed over native instructions. Furthermore, IR is supposed to support cross-platform analyses, but practically most IRs in existing systems don't have translators readily available for ISAs other than x86. Combine all three factors and it is clear that both research and design in current DTA systems lack cross-platform support. Therefore, specific DTA and performance improvement techniques can only be effective on a specific ISA and do not automatically benefit a different platform.

Thus our fundamental question: Is it possible to design a DTA system that works well on different ISAs with no or minimum adjustment and is cost-effective to implement?

II. Our Solution

This paper reports on our design and prototype for an IR-based systemic approach to support cross-platform DTA and symbolic execution (section II.A). Our architecture combines static and dynamic binary analysis to provide core functionality to different application needs (Section II.B). We implemented our Cross-platform Binary Automated Symbolic-execution System (CBASS) and performed several end-to-end security applications and evaluations for x86 and ARM binary (Section III).

A. *Cross-platform DTA and symbolic execution on IR*

In contrast to existing DTA approaches, CBASS separates native trace collection from analysis, translates native instructions to its IR equivalent first and performs DTA over the IR, rather than the native instructions. Hence, there is only one implementation for key algorithms, such as taint tracking, symbolic execution and constraint generation. Enhancements to those algorithms benefit all the platforms that can translate into this IR.

We identified fundamental requirements for this IR: (1) Availability of translators for the most popular ISAs. (2) Simple to analyze and low level enough (high level information like type is not easy to recover from binary). (3) IR semantics is faithful to native instructions, and in a form similar to bit-vector operations to facilitate symbolic execution and formula translation to Satisfiability Modulo Theories (SMT) solvers. Fortunately, rather than having to invent yet another IR, we found that REIL [2] meets these basic requirements, despite some of its limitations.

B. *Architecture to combine static and dynamic analysis:*

Our CBASS IR-based analysis engine is shown in Fig. 1.

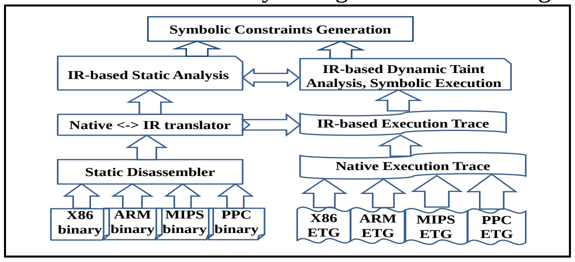


Figure 1. CBASS IR-based Binary Analysis Engine

1) *Static Binary Analysis:*

Static analysis in CBASS starts with the native assembly instructions from a binary, creates basic block, control flow graph and call graph; performs translation from native instruction into REIL instructions on-demand and keeps an efficient IR mapping cache for reuse. Each native instruction maps to a sequence of REIL instructions. Columns 1 and 2 in Table I. illustrate the mapping between native ARM instructions and IR instructions. To conserve space some IR instruction lists are incomplete.

CBASS can perform static (pure) symbolic execution. While we have performed some experiments on searching for ROP [3] gadgets using static symbolic execution, we mainly used static analysis to support dynamic analysis.

2) *Dynamic Binary Analysis:*

CBASS dynamic analysis is done offline, starting with an instruction trace generated by the Execution Trace Generator (ETG) shown in Fig. 1. Dynamic analysis involves the following key steps:

a) Mark input bytes as taint sources

In the execution trace, an input entry records the starting base memory address of the input, the size of the input and the concrete value of each byte. We assign a symbolic variable name to each input byte. In Table 1, input is passed in to a function through register *R0* and the symbolic variable *IN_R0* is created.

b) Symbolic execution on IR instructions

A symbolic expression is a tree structure, whose root node is a bit-vector operation; each of the children is a symbolic variable, another symbolic expression or a bit-vector constant. CBASS utilizes a combination of symbolic execution and concrete execution to emulate each instruction in the trace: creates a new symbolic expression for the destination operands when any source operands are symbolic expressions; uses concrete data from trace when the instruction does not involve any symbolic expressions. Third column in Table I. shows the symbolic expression of the condition (*t3*) in the last row, represented as the input symbolic variable (*IN_R0*) after it propagates through the IR instructions in second column.

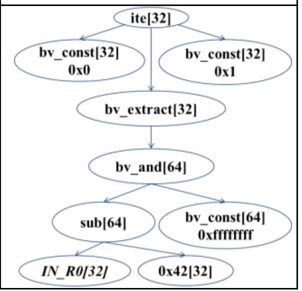
c) Check taint sink to construct constraints

Depending on the goals of the security analysis, taint sinks and constraints are selected differently. In exploit generation, the taint sink may be register(s) such as EIP, while constraints are created to steer program execution into a control-hijacked state. For a white-box fuzzing application like SAGE [4], one might choose a path condition as a sink to construct a formula that would force the code to take a different execution path.

III. Implementation and Evaluation

We implemented execution trace generators for x86 and ARM ISAs. Each execution trace generator produces trace in the same format independent of platform. We used the REIL Java libraries to translate native instructions to IR and implemented taint analysis and symbolic execution algorithms for this IR using Jython.

TABLE I. Symbolic Execution on IR instructions

Native ARM Instruction	**IR REIL Instructions**	**Path Condition Expression after Symbolic Execution on IR Instructions**
SUB R11, R12, #4	sub R12, 4, t0	 ite[32] bv_const[32] 0x0 bv_const[32] 0x1 bv_extract[32] bv_and[64] sub[64] bv_const[64] 0xffffffff IN_R0[32] 0x42[32]
	and t0, 0xffffffff, R11	
CMP R0, #0x42	sub R0, 66, t0 and t0, 0xffffffff, t1 bisz t1, , Z	
MOVNE R0, #0	str Z, , t3 jcc t3, , 32	

For both platforms, we successfully performed several initial analyses, including finding ROP gadgets, white-box fuzzing and automated exploitation generation. We also translated symbolic expressions for tainted sinks to a standard SMT formula and invoked an SMT solver (Z3 [5]) to return the input assignment that satisfies our constraints or determine that no such input exists.

Given the faithfulness of the IR we adopted, we achieved identical results using taint analysis on native instruction and on IR, for the experiments we conducted. Performance-wise, some factors, for example, translation to IR is required and almost always there are more IR instructions than native, expect to increase IR-based analysis overhead; while other facts such as IR translation only needs to be done once, can be preprocessed and reused and IR instruction is much simpler to analyze, all improve performance. Given the continuing development status of CBASS, further experiments and measurements on performance are needed in the future.

IV. Conclusion

We have designed an IR-based system to support cross-platform binary analysis, implemented the core binary analysis algorithms independent of specific ISA, and performed initial evaluations on x86 and ARM binaries. We are working to perform similar evaluations using traces from other ISAs and conduct empirical study measuring performance and scalability across platforms.

References

[1] D. Song, et al., BitBlaze: A New Approach to Computer Security via Binary Analysis, Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper, Hyderabad, India, Dec 2008.

[2] T. Dullien, and S. Porst, REIL: A platform-independent intermediate representation of disassembled code for static code analysis, CanSecWest, 2009.

[3] H. Shacham, The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86), Proceedings of the 14th ACM conference on Computer and communications security, October, 2007.

[4] P. Godefroid, M.Y. Levin, and D. Molnar, Automated whitebox fuzz testing, NDSS, 2008

[5] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In TACAS'08, Mar-Apr 2008.