Refinement-based CFG Reconstruction from Executables ⋆,⋆⋆

Sébastien Bardin, Philippe Herrmann, and Franck Védrine

CEA, LIST,
Gif-sur-Yvette CEDEX, 91191 France
first.name@cea.fr

Abstract. We address the issue of recovering a both safe and precise approximation of the Control Flow Graph (CFG) of a program given as an executable file. The problem is tackled in an original way, with a refinement-based static analysis working over finite sets of constant values. Requirement propagation allows the analysis to automatically adjust the domain precision only where it is needed, resulting in precise CFG recovery at moderate cost. First experiments, including an industrial case study, show that the method outperforms standard analyses in terms of precision, efficiency or robustness.

Motivation. Automatic analysis of programs from their executable files has many potential applications in safety and security, for example: automatic analysis of mobile code and malware, security testing or worst case execution time estimation. We address the problem of (safe) CFG reconstruction, i.e. constructing a both safe and precise approximation of the Control Flow Graph (CFG) of a program given as an executable file. CFG reconstruction is a cornerstone of safe binary-level analysis: if the recovery is unsafe, subsequent analyses will be unsafe too; if it is too rough, they will be blurred by too many unfeasible branches and instructions.

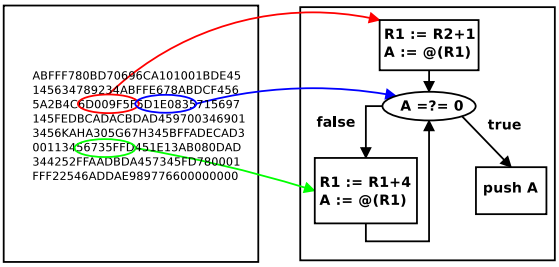

Fig. 1. CFG reconstruction from an executable file

Challenges. Such an approximation is difficult to obtain mainly because of dynamic jumps, i.e. jump instructions whose target expression is resolved at run-time and may

⋆ Work partially funded by ANR (grants ANR-05-RNTL-02606 and ANR-08-SEGI-006).
⋆⋆ The material presented here is taken from a preliminary version of the VMCAI'11 paper [3].

vary from one execution to the other. Dynamic jumps are very sensitive instructions and a small loss in precision on target expressions may affect dramatically the quality of the subsequent analysis, leading to vicious circles between value analysis and CFG reconstruction. Moreover, there is no reason why all valid targets of a dynamic jump should follow a nice regular pattern. Indeed they are just addresses in the executable code, often arbitrarily assigned by a compiler. Hence any analysis based on popular domains (i.e. convex domains possibly enhanced with congruence information) will introduce many false targets. For example, consider an instruction `cgoto(x)` with `x` $\in \{1355, 1356, 2126\}$: such an analysis cannot recover better than `x` $\in [1355..2126]$, reporting 99% of false targets.

Note that, unfortunately, dynamic jumps are ubiquitous in native code programs: they are introduced at compile-time either for efficiency (`switch` in C) or by necessity (return statements, function pointers in C, virtual methods in C++, etc.).

Related approaches. Industrial tools like IDA PRO [10] or AIT [9] usually rely on linear sweep decoding (brute force decoding of all code addresses) or recursive traversal (recursive decoding until a dynamic jump is encountered), enhanced with limited constant propagation, pattern matching techniques based on the knowledge of the compiling chain process and user annotations. These techniques are unsafe on general programs, missing many legal targets and branches. The only safe techniques are those by Reps *et al.* [4, 5] - based mainly on stride intervals propagation, and by Kinder and Veith [7, 8] - based on k-set (sets of bounded cardinality) propagation. Experiments reported by the authors show that while each approach performs much better than current industrial tools, both techniques still recover many false targets. Especially, stride intervals cannot capture precisely sets of jump targets, and k-sets are too sensitive to their cardinality bound, potentially leading to either imprecise or expensive analyses.

Our approach. We propose an original refinement-based procedure to solve CFG reconstruction [3]. The procedure is built on two main steps: a forward k-set propagation with local cardinality bounds (ranging from 0 up to a given parameter $Kmax$), and a refinement step controlling these cardinality bounds.

The forward propagation is mostly a standard one, enhanced with a few original mechanisms: (1) abstract values are downcast according to local cardinality bounds, permitting to lose information and increase efficiency; (2) $\top$ values (i.e. abstract values denoting the whole domain) are tagged with additional information recording their origin (for example $\top_{\langle 1,3,12\rangle}$ denotes the abstraction to $\top$ of the k-set $\{1, 3, 12\}$), allowing to pinpoint the *initial sources of precision loss* (ispl) and give clue for correction (cf. refinement); (3) alias, jump targets and branches that have been fired during propagation are recorded into a *journal* (cf. refinement).

Refinement is lazy and on-demand. When a jump expression evaluates to $\top$, the refinement mechanism takes place, trying to find out ispls responsible for the violation (guided by backward data dependencies and journal information) and to correct them by locally improving the domain precisions (using $\top$-flags).

Results. From a theoretical point of view, the procedure is sound and runs in polynomial-time. Moreover it is as precise as standard k-set propagation on a class of non-trivial programs, including dynamic jumps and alias [3]. From a practical point of view, the

procedure has been implemented and evaluated on an industrial safety-critical program (32 kloc) and on small handcrafted programs. It appears to be reasonably efficient (taking less than 5 minutes for the industrial case study), very precise (only 7% of false targets, beating standard approaches based on convex domains by several orders of magnitude), and very robust: the procedure does need an initial parameter, but its exact value does not seem to matter.

References

1. Balakrishnan, G., Gruian, R., Reps, T. W., Teitelbaum, T.: CodeSurfer/x86-A Platform for Analyzing x86 Executables. In: CC 2005. Springer, Heidelberg (2005)
2. Bardin, S., Herrmann, P.: Structural Testing of Executables. In: IEEE ICST 2008. IEEE Computer Society, Los Alamitos (2008)
3. Bardin, S., Herrmann, P., Védrine, F.: Refinement-based CFG reconstruction from Unstructured Programs. In: VMCAI 2011. Springer, Heidelberg (2011)
4. Balakrishnan, G., Reps, T. W.: Analyzing memory accesses in x86 executables. In: CC 2004. Springer, Heidelberg (2004)
5. Balakrishnan, G., Reps, T. W.: Analyzing Stripped Device-Driver Executables. In: TACAS 2008. Springer, Heidelberg (2008)
6. Godefroid, P., Levin, M. Y., Molnar, D.: Automated Whitebox Fuzz Testing. In: NDSS 2008. The Internet Society (2008)
7. Kinder, J., Veith, H.: Jakstab: A Static Analysis Platform for Binaries. In: CAV 2008. Springer, Heidelberg (2008)
8. Kinder, J., Zuleger, F., Veith, H.: An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In: VMCAI 2009. Springer, Heidelberg (2009)
9. Absint homepage `http://www.absint.com/`
10. IDA Pro homepage `http://www.hex-rays.com/idapro`