Dillon Prendergast

CAP5137: Research Paper

Professor Liu

28 November 2018

**Chosen Paper:** Practical Analysis of Stripped Binary Code- Laune Harris and Barton Miller

**Summary of Paper:**

The paper addresses the use of static analysis as a method to extract structural information from stripped binaries. It is vital to the analysis of binaries to have information on the flow and structures within a binary, but it is common to see stripped binaries when looking at commercial software and especially when analyzing malicious code.

The paper focuses on the efforts to build upon LEEL and RAD analysis techniques. LEEL was said to be inefficient at reaching functions that were only reachable through indirect control transfers. This technique works through the statics call flow graph to create a set of call targets for the function starting points. For functions that are only reachable through indirect control transfers, there will be a gap in the code which LEEL handles through two ways: not analyzing the gap or assume the gap to be the starting address of a function. The former option is very inefficient for code where most of the function are reached indirectly. RAD performs this by matching byte sequences in the gaps to known function prologues, which will only work if the function prologue is previously known. to have two and compares the results with the capabilities of IDAPro.

The method designed seeks to improve the code coverage of the LEEL and RAD methods, and properly handle unconventional function structures. The proposed design is broken up into four pieces: a file format reader, instruction decoder, generic assembly language interface, and parser.

The file format reader functions by extraction information from the file header to find addresses and sizes of text, data, symbol tables and the entry point. The reader then uses heuristics to determine the address of *main()*, which aids tools expecting a *main()* and finding the program's call graph.

The instruction decoder works both as a disassembler and to extract semantic information from the structures which will be implemented through architecture dependent routines. This method has capabilities for Intel IA32, AMD 64, and IBM Power architecture.

The generic assembly language interface exports a uniform assembly language to the parser which is independent from the architecture to allow for generic operations. There operations are given to each instruction as follows: isCondBranch, isUncondBranch, isIndirBranch, isCall, isReturn, getBranchTarget, getMultiBranchTargets, isPrologue, isIllegal, --, ++, and size.

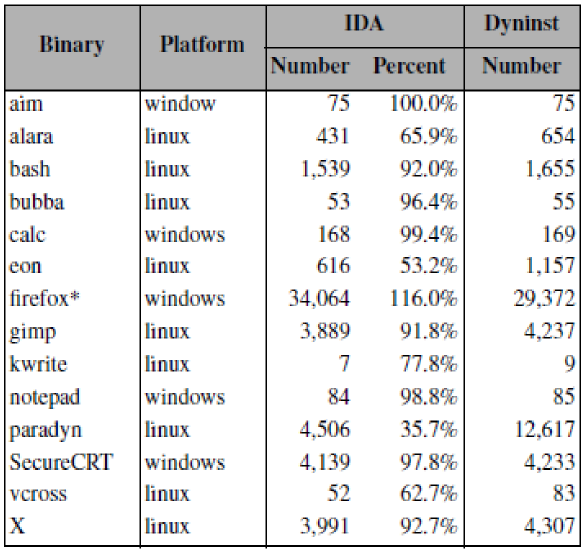
The parser uses the initial starting addresses for its algorithm as when as breadth-first call graph traversal and recursive disassembly. Valid targets get named as functions and disassembled for control flow graphs, while CFG conflicts and illegal instructions are rejected. In the case of gaps in the text due to data, padding, or unreferenced functions there are two gap completion phases.

In the first phase, gaps are checked for known function prologues and isPrologue will be called. A return of true means that the identified area is the start of a function. After the first phase if gaps persist then Orso's speculative completion method will be used. This entails build a control flow graph at each possible code address in the gap, then using CFG conflicts to prune the graphs. In addition, the CFGs must contain one non-terminal control transfer instruction in order to reduce false positives.

Two issues that were pointed out were the analysis of indirect branches and identifying exit points. When encountering jumps with targets that are determined at run time, the method used is to backtrack the control path in order to find the instructions which set up the jump table access. From there the base location and number of entries in the jump table is known.

To identify exit points, the method used involves return instruction detection accompanied by tail-cell detection in the instruction decoder. This will label non-returning call instructions with exit points, and transfers not identified with an exit are assumed to be intra-procedural.

To test their created method, they tested 519 programs found in the *bin* directory. Using *objdump* to display unique function addresses, their method provided 95.6% of the resulting addresses. The method was also tested for function recovery rate against IDAPro with the following results:



| Binary | Platform | IDA | | Dyninst |
|---|---|---|---|---|
| | | Number | Percent | Number |
| aim | window | 75 | 100.0% | 75 |
| alara | linux | 431 | 65.9% | 654 |
| bash | linux | 1,539 | 92.0% | 1,655 |
| bubba | linux | 53 | 96.4% | 55 |
| calc | windows | 168 | 99.4% | 169 |
| eon | linux | 616 | 53.2% | 1,157 |
| firefox* | windows | 34,064 | 116.0% | 29,372 |
| gimp | linux | 3,889 | 91.8% | 4,237 |
| kwrite | linux | 7 | 77.8% | 9 |
| notepad | windows | 84 | 98.8% | 85 |
| paradyn | linux | 4,506 | 35.7% | 12,617 |
| SecureCRT | windows | 4,139 | 97.8% | 4,233 |
| vcross | linux | 52 | 62.7% | 83 |
| X | linux | 3,991 | 92.7% | 4,307 |

**Application**

The application of this paper is substanstial in our ability to identify functions in stripped malware binaries and better understand them. As the paper stated, there were methods already capable of doing this in some form, but none of them have been perfected and this paper attempted to achieved greater function analysis than the other capable methods. By expanding on the known areas of difficulty in this paper even further, one can hope to achieve even further efficiency in function identification.

**Improvement**

One issue occurring in the method devised by the paper is false positive function. The example given is data bytes being interpreted as:

```
mov reg, mem
mov reg, mem
ret
```

and being identified as a basic function. To counteract false positives, it is a requirement for the identified function to have two or more control flow instructions within. This excludes legitimate functions that do not have the two requirement control flow instructions. I believe that this can be relieved by accepting the above example as a function and doing additional analysis on the instructions before and after the function call to check for the corresponding registers being used. If the registers being used in the function are again used or their values used immediately after the return of the function, then it is likely that the function is legitimately modifying the registers for future use. This will slow the analysis of the binary depending on the window in which the register usage is looked for, but I believe it will provide better insight into identifying functions, while maintaining a low false positive rate.

**References**

Laune C. Harris and Barton P. Miller, "Practical Analysis of Stripped Binary Code," ACM SIGARCH Computer Architecture News - Special issue on the 2005 workshop on binary instrumentation and application, Volume 33 Issue 5, Pages 63 – 68, December 2005.