On the Static Analysis of Indirect Control Transfers in Binaries

B. De Sutter* B. De Bus K. De Bosschere
Department of Electronics and Information Systems
Ghent University, Belgium

P. Keyngnaert B. Demoen
Department of Computer Science
Katholieke Universiteit Leuven, Belgium

Abstract *In this paper, we describe a method to handle uncertainty caused by indirect control transfers when reconstructing a control flow graph from a binary program. We have implemented our method on binaries for the Digital Alpha architecture, and we show that all but a few of the indirect jumps and more than 90% of the indirect procedure calls can be resolved automatically. A new analysis of relocation information almost halves the number of procedures conservatively assumed to be a possible callee of indirect calls. This new analysis has been incorporated in the Alto link-time optimizer and evaluated on the SPEC95 benchmarks. The obtained code size reduction with the new analysis is 30% on average, where it is only 23% without it.*

Keywords: binaries, static analysis, control transfers, optimization, compilers

1 Introduction

Binary modification has become a major research field during the last decade. Three major applications of binary modification are: (i) binary translation, becoming increasingly popular with Just-In-Time compilation for Java [1], (ii) instrumentation of binaries [2] for debugging purposes, and (iii) whole-program optimization at link-time. Link-time optimization systems are used for different purposes such as code optimization [3, 4] and code compression [5] which becomes increasingly important for mobile/embedded systems where large amounts of memory are not available because of power consumption/high cost.

These applications require a very detailed analysis of the binary program. A binary optimizer is however handicapped by the lack of semantic information about the program. Much of the information that is available for free in the compiler must be painstakingly reconstructed by the whole-program optimizer. In order to speed up or simplify the analysis, one may be tempted to make only conservative assumptions, but these assumptions often prohibit several of the more powerful optimizations.

This paper focuses on only one aspect of the analysis of a binary program, namely the reconstruction of the interprocedural control flow graph (ICFG, the combined CFGs of all procedures). The reason why we do this is that it is the basis for many other analyses and program optimizations. The reconstruction of an ICFG is straightforward for the vast majority of the code, but very hard when indirect control transfers occur (control transfers through a memory or register operand as used when calling a procedure through a procedure pointer, or when transferring control through an address table).

*The work described in this paper is funded by the Fund for Scientific Research - Flanders under grant 3G001998.

For the ease of processing, such indirect control transfers are often modeled by means of a control transfer to a fictitious *hell* node, which is the abstraction of an unknown target [3] to guarantee that only conservative program transformations can be applied.

The use of a hell node is a very elegant solution to handle unknown control flow behavior: instead of having to adapt all analyses and optimizations to deal with control flow anomalies for conservativeness, the conservativeness is incorporated in the behavior of the hell node. Some properties of the hell node are that (1) it is always reachable, (2) all registers are live at its entry-point, (3) all registers are defined with unknown contents (i.e. not constants).

Without going into too many details about the analyses that are performed on binaries, it is important to note that for (1) reachability analysis, (2) liveness analysis and (3) constant propagation, that are all fix-point calculation algorithms, it suffices to initialize the properties of the hell node before the fix-point algorithm is applied to guarantee conservativeness. From that moment on, the fix-point algorithms are not aware of the existence of the hell node.

For 64-bit architectures however, the careless use of hell edges is dramatic because almost all subroutine calls are memory indirect as we will show in section 3.1.

Hence, in order to reconstruct the ICFG, we need to carry out a careful analysis to find out what the targets of the indirect control transfers are. However, the necessary analyses, including constant propagation, require an ICFG themselves, which leads to a phase ordering problem. The solution we propose is to start with a super-conservative ICFG, where every indirect control transfer instruction is replaced by a control transfer to the hell node. After constant propagation, some of the edges to hell nodes (hereafter called *hell edges*) can be replaced by edges to regular nodes, allowing for a more detailed constant propagation, which in turn might eliminate additional hell edges, etc. After a few iterations, this process converges to a stable ICFG with fewer hell edges. The more information that is used during this process (the code itself, but also relocation information, symbol table information and the read-only data sections), the more hell edges we are able to remove.

2 Building the ICFG

The ICFG consists of the CFGs of all procedures in a program. Its vertices are basic blocks and the edges are the intraprocedural control flow paths between basic blocks and additional call and return edges to model procedure calls.

Detecting the basic blocks of a binary program for a RISC-architecture is straightforward [6]: control flow transfer instructions mark the end of a basic block, targets from direct control flow transfers and the instructions directly following control flow transfer instructions mark the entry-points (leaders) of basic blocks, as do relocations for indirect jumps since possible target addresses are marked for relocation. Procedures are detected in the same way.

The edges of the control flow graph are also straightforward if they are manifest in the binary (either absolute or relative). This is not the case for indirect control flow transfers, as the target of the control transfer is then stored either in a memory location or in a register. Information about the set of possible targets can only be obtained by means of constant propagation over the program, a simple inspection of the code does not suffice.

We will illustrate this for an Alpha binary generated from the C-code of Figure 1. The CFG for the procedure f() is depicted in Figure 2. Note that the direct calls to g0() and g1() in the source code are implemented as indirect calls since they were implemented in another source code file. Solid line edges are found on constructing the ICFG, since they come from direct control flow transfers. Dotted edges result from indirect control flow transfer, and cannot be resolved until after some more analysis. Therefor, the indirect control transfers will in a first approximation be mod-

```
extern float g0(float x);
extern float g1(float x);

float f(int fun, float x)
{
    float y;
    switch (fun)
    {
        case 0:
            y = g0(x);
            break;
        case 1:
            y = g1(x);
            break;
    }
    return y;
}
```

Figure 1: Example C-code source file that is just part of a whole program.

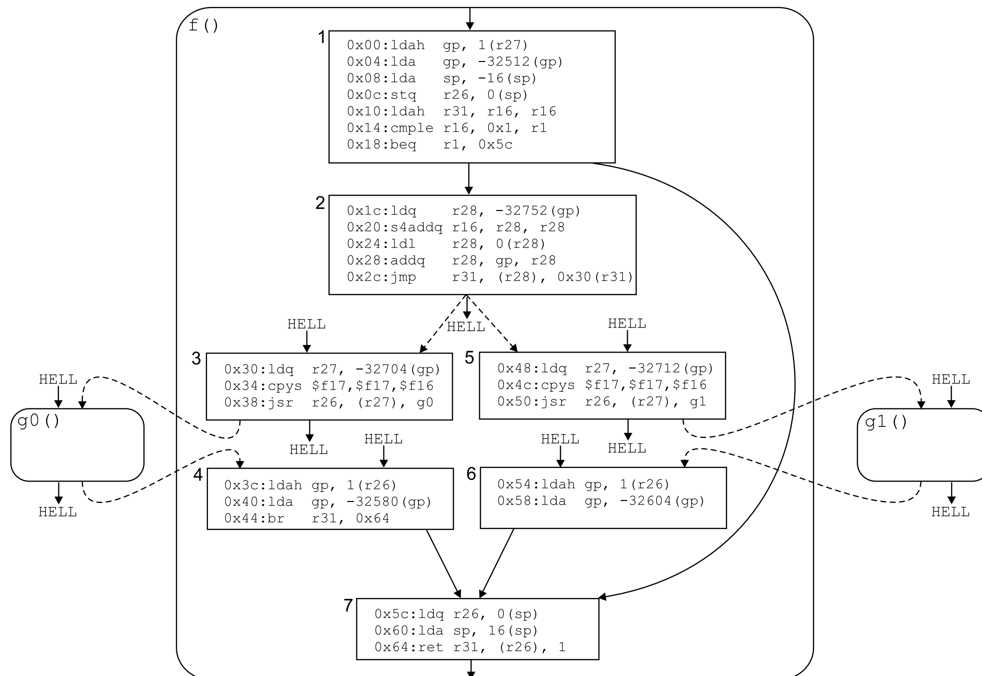

Figure 2: Part of the ICFG for the C-code in Figure 1: basic block 1 mainly tests the index in the address table, basic block 2 implements the actual table lookup, basic blocks 3/4 implement the procedure call/return to/from g0() (basic blocks 5/6 do the same for g1()) and basic block 7 implements the return. The third argument of the `jsr` and `jmp` instructions are nothing but a hint for branch prediction, they cannot be used to model control flow precisely.

eled conservatively, i.e. with edges to the hell node. All possible targets of indirect control flow transfers (relocatable addresses) become reachable from the hell node. For the sake of clarity, the hell node is not drawn as one single node in Figure 2.

3 Refining the ICFG with static analysis

The aim of the static analysis is to remove as many superfluous hell edges as possible from the conservative interprocedural control flow graph and replace them by more precise (and real) control flow paths.

There are two kinds of hell edges: hell edges that point to the hell node (*outgoing* hell edges), and hell edges that point from the hell node to a regular node (*incoming* hell edges). They will be treated in a different way.

3.1 Outgoing hell edges

Outgoing hell edges result from unknown outgoing control behavior at specific program points. These are the result of indirect control transfers, which have five major sources: (i) switch-statements that are implemented with a table lookup and an indirect jump, (ii) exception handling, where control is transferred to a handler indirectly on raising an exception, (iii) goto-statements to pointer-assigned labels, (iv) implicit or explicit use of procedure pointers, (v) dynamic method invocation in OOP languages.

The former three are implemented with indirect jumps, the latter with indirect calls. Dynamic method invocation is very hard to handle statically at the binary code level, since the type information that is needed to resolve the indirect calls will most of the time be stored in dynamically allocated memory. Resolving these calls is out of the scope of this paper.

The implementation of switch-statements by means of an address table is the major source of hell edges leaving from jump instructions.[1] To resolve jump targets resulting from a switch-statement it is necessary to relate the jump instruction to a table lookup. A table lookup that is compiler generated is most of the time implemented with a recognizable sequence of instructions existing of index normalizations, an index bound test, loading the target address using the table and the index and the jump itself.

Recognizing the sequence is sometimes tricky because the scheduler might have mixed other instructions in the sequence. Furthermore, the normalizing instructions differ with the data type of the operand in the switch-statement, and may even be duplicated if, again for scheduling reasons, a duplicate of the index is made somewhere in the sequence. The solution we have implemented is to look for the program slice producing the target address of the jump, and to match this slice with a set of known slices. After unification, the number of elements in the address table can be calculated based on the upper-bound test value and the values used for normalization of the index. Using this scheme in combination with target addresses stored in a read-only section of the binary, we are able to resolve all indirect jumps coming from switch-statements.

On raising exceptions, control is mostly transferred to a handler by means of a computed jump. If there is only one target handler at the jumping program point, which we found to be true for the SPEC95 benchmarks compiled on the Alpha architecture, it suffices to find its address, which is normally stored in a read-only section of the binary. If there is more than one possible target, the analysis required to find all possible targets is much harder. So far, we have not found a safe algorithm to resolve this.

Of all indirect procedure calls found in binaries compiled for the Alpha, the vast majority are indirect implementations of actual direct calls. The reason is that the displacement between a call and its target can potentially be

[1] In the whole SPEC95 benchmark suite compiled for the Alpha architecture, there are only two jumps not related to a switch-statement or exception handling.

a 64-bit integer that cannot be encoded in a 32-bit instruction. This means that all calls for which the compiler does not know the displacement (i.e. all intermodular calls) have to be implemented with a load from a literal address pool and an indirect procedure call.

To resolve these indirect calls, the value that is loaded in the register used for the call has to be known. Since the address of the callee is stored in the read-only literal address pool, constant propagation suffices to find the address at which the target address is stored.

What is left after these refinements are the true indirect procedure calls (through procedure pointers). These are much harder to analyze. The main reason is that the addresses that are loaded do not come from a read-only section.

3.2 Incoming hell edges

Some incoming hell edges result indirectly from the outgoing hell edges discussed in the previous paragraphs (e.g. a call edge to hell always has a corresponding return edge from hell). For those, refinement is easy because these edges can be replaced together with the outgoing edges.

Other incoming hell edges are created because a basic block's entry instruction is relocatable: its relocatable address is stored in the data sections.

This is the case for indirect jumps, where 32-bit displacements are stored in address tables. These displacements have 32-bit relocations associated with them. As long as there are indirect jumps in a procedure that are not resolved we must assume the jumps can reach all the 32-bit-relocatable blocks in the procedure, and therefor include a hell edge in the ICFG to model the unknown predecessors. Once all indirect jumps in a procedure are resolved however, we know all possible predecessors of these basic blocks and can safely remove the hell edges.

This is also the case for the entry-points of all procedures that are accessible from outside their own object file, since they can only be called by an indirect call, as indicated in the previous section. Potentially, any procedure of which the entry-point address is stored in a data section can have its (64-bit relocatable) address loaded somewhere in the program and be the target of an indirect procedure call (e.g. using a procedure pointer). As a consequence, as long as there are indirect calls in the program of which the possible callees are not known, all the aforementioned procedures can be their callees and hence must be assumed to be reachable by a hell edge to guarantee correctness. This scheme works very well, but unfortunately it is overly conservative and is therefor extended with an analysis of the program code and the associated relocations.

On the Alpha, all instructions loading addresses from the literal address pool have an associated relocation, since the location where the call target address is loaded, is itself relocatable. This allows these load instructions to be identified. Further more, the instructions using the loaded address have a relocation indicating the purpose of the loaded value (either an indirect call or something else). Using this information, we delete the call edge from hell to a procedure if loads of it's entry-point address come from the literal address pool only and are not used for anything but indirect calls and if all those indirect calls have been resolved.

Note that we know prior to building the original ICFG for which procedures the removal of the hell edges will be viable. Still we have to add them to the ICFG at first, since at that point, not all calling contexts are linked to the called procedures and thus we need to approximate the joined calling contexts conservatively, with a call from hell. This also keeps the called procedure live.

4 Evaluation

We have shown how a hell node can be used to model unknown control flow behavior in a conservative approximation of the ICFG and how the ICFG can be refined during the analysis and transformation of the binary.

benchmark	Before optimization & hell edge removal						After optimization & hell edge removal					
	#procs	#ins	calls to hell	jumps to hell	calls from hell	jumps from hell	#procs	#ins	calls to hell	jumps to hell	calls from hell	jumps from hell
compress95	334	27103	1023	38	230	108	143	18422	131	1	51	2
gcc	2795	365031	15073	260	1382	2439	1445	277453	377	1	284	2
go	997	90200	2308	45	321	195	271	71741	160	1	51	2
ijpeg	819	75164	2366	43	518	149	409	51580	744	1	236	2
li	726	43415	2047	45	540	157	441	26092	147	1	245	2
m88ksim	674	61007	2508	57	463	272	334	42078	277	1	112	2
perl	788	111671	5098	94	549	1065	402	81811	461	2	59	4
vortex	1559	174517	9650	72	904	366	539	100824	205	2	100	4
applu	1123	150212	5095	187	800	881	499	106016	421	2	142	4
apsi	1275	169495	5364	186	819	903	542	118630	423	2	143	4
fpppp	1152	151105	5011	182	816	884	498	103076	421	2	142	4
hydro2d	1174	150885	5173	194	804	933	527	107184	421	2	142	4
mgrid	1115	142616	5117	193	801	927	503	98717	421	2	142	4
su2cor	1158	152308	5168	195	810	944	524	108108	421	2	142	4
swim	1105	139714	5043	186	800	876	496	96048	421	2	142	4
tomcatv	1064	136113	4964	189	776	896	481	95443	406	1	142	3
turb3d	1139	147961	5132	192	802	922	514	104364	421	2	142	4
wave5	1272	171319	5180	177	811	840	555	124689	421	2	142	4
geom. avg.	1038	118442	4312	118	670	753	462	82345	341	2	129	3
after/before							**45%**	**70%**	**8%**	**1%**	**19%**	**1%**

Table 1: Results of the optimization and hell edges removal.

To evaluate this method for real-life applications, we have analyzed the hell edges for the SPEC95 benchmark suite when being optimized with Alto, A Link-Time Optimizer for the Alpha architecture. The binaries were created using the DEC C compiler (version 5.6-071) on a Digital UNIX V4.0 system. The compiler flags used were `-O4, -Wl,-d -Wl,-z -non_shared`. This means full compiler optimization and static linking, including relocation and symbol information in the binary [2]. Table 1 shows a summary of the results.

As can be seen from the table, the number of indirect calls in the binaries drops with 92%. This is obtained by the strength-reduction in Alto that converts indirect calls to direct calls if the displacement between call-site and callee is small enough to be encoded in the call instruction. The number of procedures in the binaries drops by 55%, due to the combined effect of the removal of dead procedures and the inlining of procedures that have only one call site. Note that without the hell edge removal based on the relocation information indicating the use of loaded addresses, this could only have been 35% since 65% of the procedures was originally considered as a callee from hell. 89% of those are no longer considered as a callee from hell because of the new analysis.

The number of instructions in the final optimized binary drops with about 30%. Without using the relocation information indicating the use of loaded addresses, this was only 23%.

5 Related Work

The `setjump` and `longjump` procedures in C are typical examples of control flow anomalies with interprocedural branches. These are examples of hard to model behavior, not of unknown behavior. For how to model these in the same ICFG, we refer to [4]. The basic idea is to add additional edges to the ICFG that guarantee conservativeness. Again, the analyses and optimization algorithms themselves need not be aware of the anomalies.

Alto [4] is a Link-Time Optimizer developed at the University of Arizona and Ghent Uni-

[2] The Alpha compiler refuses to include relocation information if the binaries are not statically linked.

versity. By optimizing binaries, it now speeds them up with e.g. 16.5% on average for the SPECint95 benchmarks. Alto does a much better job than OM [3], a DEC link-time optimizer. OM uses a hell function, but there is no literature on the removal of hell edges in OM.

Spike [7] optimizes Alpha/NT-executables. If indirect control flow transfers is unresolved, conservative approximations (following calling conventions) are used for the different analysis and optimizations. These approximations are not modeled by a hell node but are incorporated in the Program Summary Graph [8] nodes. In this graph each node describes properties of the corresponding program point, such as liveness information and may-use information.

EEL, the Executable Editing Library[9], uses program slices to resolve control flow in more or less the same way we do. For program tracing, when static analysis is unable to resolve indirect calls or jumps, they insert code that translates target addresses at runtime. The instruction rescheduler implemented using EEL [10] is a local scheduler, so there is no need for accurate CFG information.

6 Conclusion

This paper describes how hell nodes and hell edges can be used to model the uncertainty caused by indirect control transfers. We describe a new analysis to greatly reduce the number of hell edges in the ICFG using relocation information indicating the use of loaded addresses . The method has been implemented for the Digital Alpha in Alto, A Link-Time optimizer. We are able to convert 92% of all indirect procedure calls to direct calls and the new analysis results in an additional 7% average code size reduction for the SPEC95 benchmarks.

References

[1] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V.M. Parikh, and J.M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *ACM Sigplan Notices*, volume 33, pages 280–290, 1998.

[2] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[3] A. Srivastava and D.W. Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Languages*, pages 1–18, March 1993.

[4] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. alto : A link-time optimizer for the dec alpha. Technical Report 98-14, The University of Arizona, 1999.

[5] S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compression. *ACM Transactions on Programming Languages and Systems*, 2000. Accepted for publication.

[6] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[7] R. Cohn, D. Goodwin, P.G. Lowney, and N. Rubin. Spike: An optimizer for alpha/nt executables. In *USENIX Windows NT Workshop*, August 1997.

[8] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 47–56, 1988.

[9] J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[10] E. Schnarr and J.L. Larus. Instruction scheduling and executable linking. In *Workshop on Compiler Support for System Software*, February 1996.