Human-Machine Resolution of Invisible Control Flow[⋆]

Suresh Kothari*, Ahmed Tamrawi* and Jon Mathews†
*Iowa State University, †EnSoft Corp.
Ames, Iowa
*{kothari,atamrawi}@iastate.edu, †jmathews@ensoftcorp.com

***Abstract*—Invisible Control Flow (ICF) results from dynamic binding and asynchronous processing. For modern software replete with ICF, the ability to analyze and resolve ICF is crucial for verifying software. A fully automated analysis to resolve ICF suffers from imprecision and high computational complexity. As a practical alternative, we present a novel solution of interactive human-machine collaboration to resolve ICF.**

Our approach is comprised of interactive program analysis and comprehension to systematically capture and link the clues crucial for resolving ICF. We present the tool support we have developed using the query language and visualization capabilities of the Atlas Platform. We illustrate the approach using examples where resolving ICF is crucial to verify software and show a complex bug in the Linux kernel discovered by resolving ICF.

I. Introduction

Automated verification of software has been the holy grail of software engineering research [1]. A fully automated analysis encounters NP hard problems with ICF as a major source of difficulty. A recent review [2] points out the issues static analysis and formal verification tools have in tackling ICF. The ICF due to dynamic binding mechanisms such as function pointers can be resolved by automated pointer analysis. However, due to imprecision of the analysis, it can lead to false results. More expensive pointer analysis increases the cost significantly without corresponding precision gains [3], [4]. The ICF due to aysnchronous processing cannot be discovered by pointer analysis.

For even the fully automated analysis as implemented by tools such as [5]–[9], human effort is necessary to cross-check the analysis results. We advocate a different approach by targeting automation to produce evidence for the human to ensure correct resolution of ICF.

II. Dynamic Binding ICF

The human-machine collaboration methodology for resolving ICF due to dynamic binding is illustrated with an example drawn from the XINU operating system kernel.

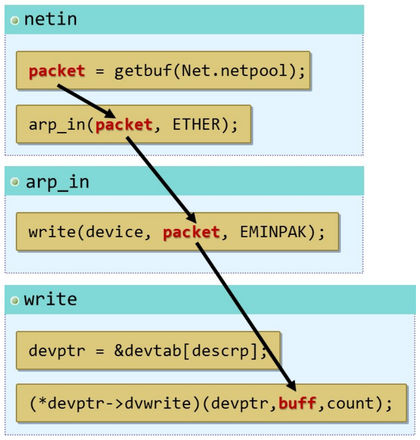
XINU uses a pattern similar to a Dynamic Dispatch Table (DDT) to organize function pointers for device drivers. Function pointers are organized in an array that acts as a demultiplexer with the array index as the selector. Developers use function pointers to call specialized methods for different customizations of a single logical task. The unique ID for a specialized method, passed as a parameter to the function pointer call, is used as the array index into DDT to select the target function. We illustrate the specific pattern in XINU, but a similar pattern is used in other systems.

A representative case shown below is that of device drivers with `OPEN`, `READ`, `WRITE`, `CLOSE` and other logical tasks to be performed for different types of devices. A DDT pattern is used to map each logical task to a specialized method for a particular device type. For example, `WRITE` maps to `dskwrite` or `ttywrite` for the disk or the terminal device type.

A. A Dynamic Binding Example

As shown in Figure 1, the situation is as follows:

1) Memory is allocated by invoking `getbuf` inside the function `netin`. The pointer `packet` to the allocated memory is passed as the first parameter to the function `arp_in`.
2) Inside `arp_in`, the pointer (`packet`) is passed as the second parameter to the function `write`.
3) Inside `write`, the pointer (`packet`) is passed as the second parameter to a function called using a function pointer.



```
packet = getbuf(Net.netpool);
arp_in(packet, ETHER);
```

```
write(device, packet, EMINPAK);
```

```
devptr = &devtab[descrp];
(*devptr->dvwrite)(devptr,buff,count);
```

Fig. 1. Pointer to allocated memory passed to a function pointer call

A Memory Leak Verification Problem: Decide whether the memory allocated in `netin` gets deallocated through the function pointer call. A precise resolution of the function pointer call is crucial to get the correct answer.

⋆This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0080 and FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. We thank our colleagues from Iowa State University and EnSoft for their help with this paper. Dr. Kothari is the founder President and a financial stakeholder in EnSoft.

B. Human-Machine Collaboration for Dynamic Binding ICF

The machine performs automated analysis to produce results that the human observes to decide the next analysis step.

1) The machine analyzes inter-procedural flow of the memory pointer starting from the allocation of memory inside `netin` to the point in the function `write` where the memory pointer gets passed as a parameter to a function called using a pointer.
2) The human observes that the function pointer comes from an array of pointers `devtab[]`, indicating the DDT pattern. The human observes that `descrp`, a parameter to the function `write` is used as the array index to select the function pointer. The human uses the machine to perform reverse inter-procedural dataflow to get the value of `descrp`.
3) The machine shows that the value of `descrp` is the constant ETHER equal to 5.
4) The human uses the machine to finde the code that initializes the array of function pointers `devtab[]` and using the index 5 resolves the function pointer call to the function `ethwrite`.

A completely automated pointer analysis to trace through individual array elements is expensive, it does not work in all cases, and its answer cannot be guaranteed to be correct. On the contrary, the proposed human-machine collaboration does not require much manual effort and the human can guarantee its correctness by breaking down the problem into pieces that are easy to understand and reason about.

Besides the DDT pattern, the human-machine collaboration can be adapted to use other clues and the *type signature* strategy [4]. In the above example, a clue for locating the target function could be to look for functions that receive as the second parameter a pointer to the structure `epacket`. Another clue could be that the target function is named after the logical task and thus it is likely to include `write` in its name. The Atlas query language (Section IV-A) has the flexibility to express a variety of such clues.

III. Asynchronous ICF

The human-machine collaboration methodology for resolving ICF due to asynchronous processing is illustrated with an example drawn from the XINU [10] oprating system kernel.

Asynchronous processing presents an even more difficult case of ICF. Dynamic binding has at least a control flow artifact, i.e. the function pointer call, as the point in program where the control flow goes to a new method. Asynchronous processing does not have any explicit control flow artifact; the control flow changes are caused by external events such as interrupts. These events can happen at any point in the program. This makes it intractable for automated analyzers to resolve ICF due to asynchronous control flows.

We present a human-machine collaboration strategy to handle asynchronous ICF for the Matching Pair Verification (MPV) problems which involve verifying the correct pairing of two events on all possible execution paths. Specific examples of such events can be: *allocation* and *deallocation* of memory, *locking* and *unlocking* of mutex, or *sensitive source* and *malicious sink* for a confidentiality breach.

We illustrate the strategy using an example of memory leak verification problem from the XINU kernel. It involves the classic *producer-consumer* pattern as follows. The *producer* allocates the memory, the pointer to the allocated memory is inserted in a globally shared linked list. The *consumer* gets the pointer to the allocated memory from the linked list and deallocates the memory. The *producer* and the *consumer* work asynchronously. The consumer is interrupt-driven in the following example of XINU disk driver.

A. An Asynchronous Processing Example

The situation is as follows:

1) Memory is dynamically allocated by invoking `getbuf` inside the function `dswrite`. The memory is allocated for a structure of type `dreq`. The `drptr` pointer to the allocated memory is passed to other functions and eventually it is inserted in a globally shared linked list with the code `dsptr->dreqlst = drptr`.
2) The function `dsinter` gets `drptr`, and deallocates memory with the code `drptr = dsptr->dreqlst` followed by `freebuf(drptr)`.

A Memory Leak Verification Problem: Decide whether the memory allocated in `dswrite` gets deallocated.

The verification is intractable using just pointer and control flow analysis. Since the function `dsinter` is driven by interrupts and not called anywhere in the code, control flow analysis cannot reach `dsinter`. In fact, `dsinter` would appear to be dead code. The pointer to the allocated memory is inserted in a linked list, so it cannot be tracked individually by pointer analysis. Automated memory leak analyzers declare such asynchronous cases as memory leaks. Without tool support, it becomes difficult and time consuming for the human to determine whether it is a false positive.

B. Human-Machine Collaboration for Asynchronous ICF

The real power of automation is the ability to build models to solve complex problems. An automated tool can enable modeling that is impossible to do by hand, because of the enormous size of software. A model can be iteratively refined to a point where the human determines it to be good enough to reason about a given problem.

We illustrate here an iterative model refinement to reason about the asynchronous ICF problem. The model is based on the *matching pair* requirement. The starting point is the function `dswrite` which calls `getbuf`. The goal is to build a model to find asynchronous functions that have the corresponding `freebuf` call(s).

1) The machine analyzes the inter-procedural flow of the memory pointer starting from `dswrite` to the point in the function `dskenq` where the memory pointer is inserted in a globally shared linked list with the code `dsptr->dreqlst = drptr`. At this point, the human takes over.
2) The human reasons that the asynchronous functions must call the deallocation function `freebuf` if it is not a

memory leak. To locate the asynchronous function, the human decides to compute the *Reverse Call Graph* (RCG) with `getbuf` and `freebuf` as the leaves.

3) The machine computes the RCG. The human notes that the RCG is too big as it includes all allocations and deallocations besides the type `dreq` for which memory is allocated in `dswrite`. The human refines the search model by restricting the RCG to functions that reference structures of the type `dreq`. The resulting search model is shown in Figure 2.

A complete verification involves examining the totality of execution paths, which can be divided into two types: the paths on which deallocation happens asynchronously and the other paths where the pointer to the allocated memory is passed through a call chain to the function that deallocates memory. The model in Figure 2 reflects both types of paths. It shows a call chain to `dskopt` which in turn calls `freebuf` to deallocate the memory. The human can infer the asynchronous deallocation from the model in Figure 2 by observing that the function `dsinter` is not connected to `dswrite` by a forward or a reverse call chain. Moreover, `dsinter` calls `freebuf` but *not* `getbuf`. Based on the observation, the human can hypothesize that `freebuf` in `dsinter` pairs with the `getbuf` in `dswrite`. The human can validate the hypothesis by checking that inside `dsinter`, the pointers are withdrawn from the global linked list, the memory addressed by each pointer is deallocated, and this happens until the linked list becomes empty.

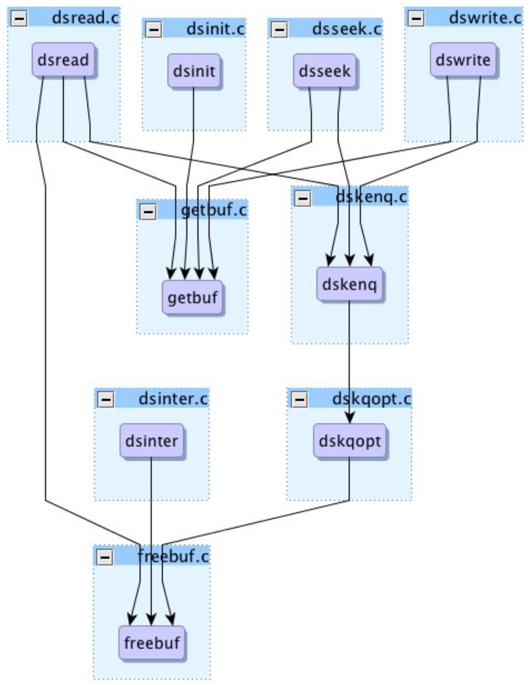


Fig. 2. A model to locate asynchronous functions

.

IV. Tool Support

We use the query language and the visualization capabilities of Atlas [11], [12]. The Atlas queries are composable. The result of a query can be stored in a variable and then passed as an input to another query. Atlas uses the eXtensible Common Software Graph (XCSG) as a common schema to write language independent analyzers. Relationships (edges) can be selected by keywords such as `XCSG.Call` in the *Call Graph* (`cg`) function (Section IV-A). Readers can get more information about XCSG and Atlas from [11], [12].

A. Query Language

We illustrate below the *Call Graph* (`cg`) and the *Reverse Call Graph* (`rcg`) queries using the Atlas query language.

Call Graph Function

```
public Q cg(Q function){
  return edges(XCSG.Call).forward(function);
}
```

The *Reverse Call Graph* (`rcg`) function is implemented as above but change `forward` to `reverse`.

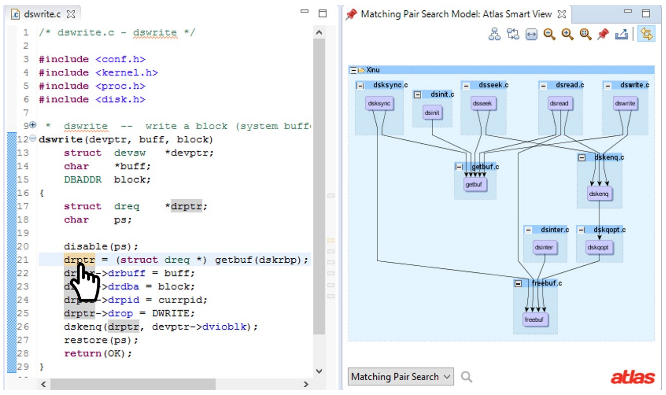
The search model, described in Section III-B, can be built with the following sequence of queries:

```
1. var a1 = functions("getbuf");
2. var a2 = functions("freebuf");
3. var leaves = a1.union(a2);
4. var fullrcg = rcg(leaves);
5. show(fullrcg);
6. var dreq = Type("dreq");
7. var dreqrefs = ref(dreq);
8. var dreqroots = dreqrefs.root();
9. var restrictedrcg = graph(dreqroots,leaves);
10. show(restrictedrcg);
```

The `graph` query 9 induces a call graph with `leaves` as the leaves and `dreqroots` as the roots as defined by the prior queries 4 and 8 respectively.

B. SmartView Capability

In Atlas, a SmartView offers a way to interactively apply a sequence of queries to a selection in source code or on a graph produced by a previous query. Figure 3 shows the result of applying the *search model* SmartView we implemented above on a reference to `drptr`, a pointer to the type `dreq`.



```
/* dswrite.c - dswrite */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <disk.h>

 *  dswrite  --  write a block (system buff
dswrite(devptr, buff, block)
    struct  devsw   *devptr;
    char    *buff;
    DBADDR  block;
{
    struct  dreq    *drptr;
    char    ps;

    disable(ps);
    drptr = (struct dreq *) getbuf(dskrbp);
    drptr->drbuff = buff;
    drptr->drdba = block;
    drptr->drpid = currpid;
    drptr->drop = DWRITE;
    dskenq(drptr, devptr->dvioblk);
    restore(ps);
    return(OK);
}
```

Fig. 3. The *search model* invoked as a SmartView

As another example, we show the built-in *dataflow* SmartView in Atlas. As noted earlier, the pointer to the memory allocated in `netin` is passed as the second parameter to `ethwrite`, a function called using a function pointer. Using the *dataflow* SmartView, Figure 4 shows the pointer to allocated memory is passed from `ethwrite` to `ethstrt`, then it is accessed by `ethinter` to deallocate memory. The pointer to the allocated memory is received as a generic character pointer by `ethwrite` and thus the type-based *search model* would not locate `ethinter`. However, the *dataflow* model provides an alternative. This examples shows that the flexibility to choose an appropriate mode is crucial to circumvent difficulties encountered in fully automated analysis.

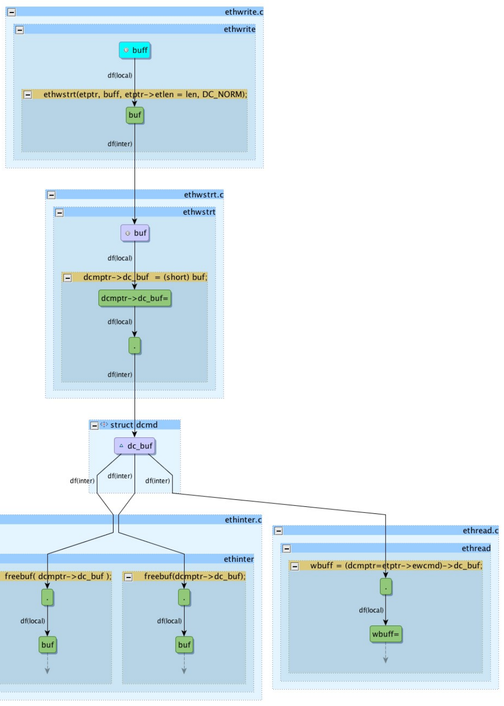


Fig. 4. The dataflow SmartView

.

V. A Linux Bug

We present an example of a complex Linux bug we discovered by resolving ICF. The lock and unlock are on disjoint paths in the function `drxk_gate_crtl` (`f1`) and if $C =$ `true`, the lock occurs, otherwise, the unlock occurs. The lock and unlock can match if `f1` is called twice, first with $C =$ `true` and then with $C =$ `false`. A quick query shows that `f1` is not called directly anywhere. Thus, it is either dead code or `f1` is called using a function pointer.

Resolving ICF due to function pointers as described earlier, we find the situation shown in Figure 5. The function `tuner_attach_tda18271` (`f2`) calls the function `f1` via function pointer. `demo_attach_drxk` sets the function pointer to `f1`, the pointer is communicated by parameter passing to `dvb_input_attach`, then to `f2`.

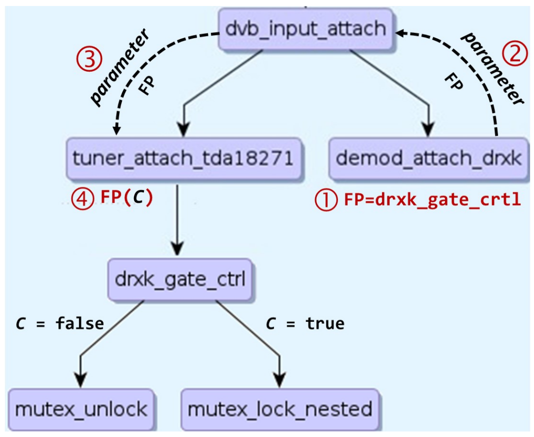


Fig. 5. Search model for `drxk_gate_crtl` after resolving calls via function pointers

Recall that `f1` must be called twice. The function `f2` has a path on which there is a return before the second call to `f1` and thus it is a bug.

VI. Conclusion

The paper describes program comprehension techniques and tool-support for cases of control behavior which are difficult to predict automatically. The paper presents a part of our human-in-loop program analysis research on DARPA APAC [13] and STAC [14] projects. Besides the two ICF categories presented in this paper, our ongoing research encompases other variants of ICF such as the flows hidden due to APIs [15].

References

[1] B. Gates, “Bill Gates Keynote: Microsoft Tech-Ed,” 2008. [Online]. Available: http://news.microsoft.com/speeches/bill-gates-keynote-microsoft-tech\%E2\%80\%A2ed-2008-developers/
[2] Scheibner, “Control flow analysis of event-driven program,” https://gnunet.org/node/2590.
[3] A. Milanova, A. Rountev, and B. G. Ryder, “Precise call graphs for c programs with function pointers,” Jan. 2004, vol. 11, no. 1, pp. 7–26.
[4] D. Atkinson, “Accurate call graph extraction of programs with function pointers using type signatures,” in *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC-2004)*. IEEE, 2004.
[5] Y. Xie and A. Aiken, “Saturn: A scalable framework for error detection using boolean satisfiability,” *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 3, May 2007.
[6] “Linux driver verification tool,” http://linuxtesting.org/ldv.
[7] “Clang static analyzer,” http://clang-analyzer.llvm.org/.
[8] Y. Sui, D. Ye, and J. Xue, “Detecting memory leaks statically with full-sparse value-flow analysis,” *Software Engineering, IEEE Transactions on*, vol. 40, no. 2, pp. 107–122, 2014.
[9] “Coverity static analysis,” http://www.coverity.com.
[10] “XINU,” http://en.wikipedia.org/wiki/XNU.
[11] T. Deering, S. Kothari, J. Sauceda, and J. Mathews, “Atlas: a new way to explore software, build analysis tools,” in *Companion Proceedings of the 36th ICSE*, 2014.
[12] “Ensoft corp.” http://www.ensoftcorp.com.
[13] “Automated Program Analysis for Cybersecurity (APAC),” https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-11-63/listing.html.
[14] “Space/Time Analysis for Cybersecurity (STAC),” https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-14-60/listing.html.
[15] T. Deering, G. R. Santhanam, and S. Kothari, “Flowminer: Automatic summarization of library data-flow for malware analysis,” in *Information Systems Security - 11th International Conference, ICISS 2015, Kolkata, India, December 16-20, 2015, Proceedings*, 2015, pp. 171–191.