Human-Centric Verification for Software Safety and Security

Payas Awadhutkar

04/15/2020

Background

- 2009-2013
  - B.Tech. in Computer Engineering, College of Engineering, Pune, India
  - Internship: Texas Instruments (2012), Bangalore, India
- 2013-2014
  - Design Verification Engineer, Texas Instruments, Bangalore, India
- 2014-2015
  - Teacher and Content Developer, Jnana Prabodhini, Pune, India
- 2015-
  - Ph.D. in Computer Engineering, Iowa State University

Research Activities

- Publications (8)
- Program Analysis Tools (3)
- Courses Taught (2)
- Invited Talks (1)

First Author/Lead Student Author Publications

- **Payas Awadhutkar**, Ganesh Ram Santhanam, Ben Holland, Suresh Kothari: **DISCOVER: Detecting Algorithmic Complexity Vulnerabilities,** *27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019), Tallinn, Estonia, August 2019*

- **Payas Awadhutkar**, Ganesh Ram Santhanam, Ben Holland, Suresh Kothari: **Intelligence Amplifying Loop Characterizations for Detecting Algorithmic Complexity Vulnerabilities,** *24th Asia-Pacific Software Engineering Conference, December 2017*

- Suresh Kothari, **Payas Awadhutkar**, Ahmed Tamrawi, Jon Mathews: **Modeling Lessons from Verifying Large Software Systems for Safety and Security**, *50th Winter Simulation Conference, December 2017*

- Suresh Kothari, **Payas Awadhutkar**, Ahmed Tamrawi: **Insights for Practicing Engineers from a Formal Verification Study of the Linux Kernel**, *27th International Symposium on Software Reliability Engineering, October 2016*

Co-Author Publications

- Ben Holland, **Payas Awadhutkar**, Ahmed Tamrawi, Jon Mathews, Suresh Kothari: **COMB: Computing Relevant Program Behaviors**, *40th International Conference on Software Engineering, May 2018*
- Ben Holland, Ganesh Ram Santhanam, **Payas Awadhutkar**, Suresh Kothari: **Statically-Informed Dynamic Analysis Tools to Detect Algorithmic Complexity Vulnerabilities**, *16th International Working Conference on Source Code Analysis and Manipulation, October 2016*
- Ahmed Tamrawi, Sharwan Ram, **Payas Awadhutkar**, Ben Holland, Ganesh Ram Santhanam, Suresh Kothari: **DynaDoc: Automated On-Demand Context-Specific Documentation**, *3rd International Workshop on Dynamic Software Documentation, September 2018*
- Suresh Kothari, Ganesh Santhanam, Benjamin Holland, **Payas Awadhutkar**, Jon Mathews, Ahmed Tamrawi: **Catastrophic Cyber-Physical Malware**, *book chapter in Versatile Cybersecurity, Advances in Information Security (ADIS, volume 72), April 2018*

Program Analysis Tools Developed

- Loop Comprehension Toolbox (Toolbox to aid human comprehension of loops)
  - *Open Source - https://ensoftcorp.github.io/loop-comprehension-toolbox/*
- RULER (Tool Suite to detect STAC vulnerabilities)
- Memchecker (Tool Suite to verify Memory Leaks)

Courses

- SE 421: Software Analysis and Verification for Safety and Security, Iowa State University
  - TA'd and Helped to develop with Benjamin Holland, Fall 2018.
  - TA'd and Co-taught with Suresh Kothari, Spring 2019 - present
- Learn to Understand, Analyze, and Verify Large Software, National Institute of Technology, Patna, India
  - Short Course about Verifying Large Software using Static Analysis
  - Co-Taught with Suresh Kothari and Akshay Deepak, March 2017

Invited Talks

- **Intelligence Amplifying Technology for Cybersecurity**, *Central Area Networking and Security Workshop, October 2019*

Recap - Young Researcher in Software Verification

- I began my Ph.D. with the DARPA Space/Time Analysis for Cybersecurity (STAC) program.
- Forms the part 1 of my dissertation (presented at Qualifying exam)
  - A human-on-the-loop approach was developed
  - We were one of the top performers, and eventually achieved 100% accuracy
- A major challenge was to prove absence of vulnerability.
  - A counter-example is sufficient to prove that there is a vulnerability.
  - How does one prove that there is no vulnerability?

Recap - Young Researcher in Software Verification

- Proof of Verification (Prelims)
  - More generally, how to prove that the verification result produced by the software is correct?
  - Human-Centric Proofs - What artifacts should be produced to enable a human to cross check the results of an automated tool?
  - Can we improve verification results using these proofs?

Recap: Prelims

- 2-event matching: Verify that an event e1(O) must be succeeded by an event e2(O) on every feasible execution path, where the two events are operations on the same object O.
  - e.g. Lock/Unlock Pairing (LUP), Memory Allocation/Deallocation pairing (MADP)
- Thesis 1: “There should be a general verification algorithm that can verify 2-event matching”
  - State-of-the-art[1] can accurately verify 99.4% of the LUP instances in the Linux kernel but only 35.8% of the MADP instances
  - Reason: MADP presents a different set of complexities in the pointer analysis
  - These complexities must be addressed to improve the results

[1] Evidence-enabled verification for the Linux kernel, Ahmed Tamrawi, Ph.D. Dissertation

Recap: Prelims

- Thesis 2: “There should be an evidence of the correctness of the verification that a human can cross-check”
  - Closest we could find was
    - Counter examples generation for bugs
    - Work of Tamrawi[1]. Tamrawi produces evidence for Control Flow paths but does not reveal how the data flow was captured.

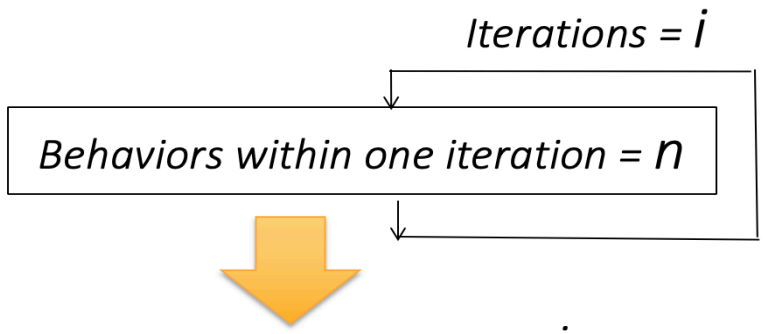
[1] Evidence-enabled verification for the Linux kernel, Ahmed Tamrawi, Ph.D. Dissertation

Summary of Contributions (since Prelims)

- Developed a new Data Flow algorithm for memory leaks verification
- Improvement of Memory Allocation/Deallocation Pairing (MADP) verification
  - Evaluation using Linux kernel
- Evidence Graph - A novel abstraction to prove the result of the verification

Fundamental Challenges in MADP Verification

- Control Flow Challenge
  - Path Explosion - Loops and branches combine to create an exponential number of paths even for a small program



Suppose there are b non-nested branch conditions inside the loop then $n = 2^b$ → *Total # Behaviors =* 2^{bi}

$i = 100$ and $b = 5$ → Total # Behaviors = 2^{5*100} = approx. 10^{150}

(The estimated number of atoms in the universe = 10^{78} - 10^{82})

Fundamental Challenges in MADP Verification

- Control Flow Challenge
  - Path Explosion - Loops and branches combine to create an exponential number of paths even for a small program
  - Solution: Path Explosion is handled using Projected Control Graphs (PCG) by Tamrawi[1] and Path Expressions[2]. Path Expressions are used to accurately capture the loop behaviors. Tarjan[2] developed an efficient algorithm to compute Path Expressions, which we have implemented.

[1] Projected Control Graph for Accurate and Efficient Analysis of Safety and Security Vulnerabilities, Ahmed Tamrawi and Suresh Kothari, APSEC, 2016
[2] Fast Algorithms for Solving Path Problems, Robert E. Tarjan, Journal of ACM, 1979

Illustration of the solution for Control Flow challenge

```
1   p = malloc();
2   while() {
3       free(p);
4       p = malloc();
5   }
6   free(p);
```

Notation:
A(m) - allocation on line m
D(n) - allocation on line n
* - expression repeated zero or more times

Path Expression as computed by Tarjan's algorithm - A(1) (D(3) A(4))* D(6)

This expression succinctly captures all the behaviors of the loop.

Fundamental Challenges in MADP Verification

- Data Flow Challenge
  - MADP verification needs to track pointer to the allocated memory across functions
  - There are three modes of interprocedural data flow
    - 1) Parameter
    - 2) Return
    - 3) Global Variables
  - The data flow algorithm to track the pointer to the allocated memory needs to handle all three of them

Fundamental Challenges in MADP Verification

- Data Flow Challenge
  - There is additional complexity in MADP caused by pointer chains
  - This complexity is not presented in LUP
  - The complexity is caused by pointer chains.
    - Pointer chains are formed using structures in C
    - If the pointer to the allocated memory is a field of a structure then, the pointer to the structure must also be tracked

Data Flow complexity of MADP - Pointer Chains

```
1  typedef struct st {
2      int *p;
3  } st;
4
5  st* foo() {
6      st *s;
7      s->p = alloc();
8      return s;
9  }
```

- An example of the complexity caused by Pointer chains
- Normally, we would only track pointer to the allocated memory p.
- Now, we also need to track s, pointer to the structure containing p

Related Work for tracking pointers for MADP

- SVF - Sparse Value Flow, pointer analysis tool for memory leaks
  - Implements a field-sensitive Andersen's analysis
  - Does not handle Global Variables
  - Not known to scale well
    - Especially for the software of the size of the Linux kernel
- BLAST and CPAChecker
  - State-of-the-art model checker for Linux kernel verification
  - Uses an algorithm similar to Ferrante, Ottenstein, and Warren (FOW)[1] to compute Data Flow. Uses heuristics which are not well documented.
  - Does not handle pointer chains
  - Selective support of Linux kernel (only certain versions supported)

Algorithm to analyze pointers

- We developed a new algorithm to address the complexity presented by MADP
- This algorithm is an improvement over the Data Dependence Graph of Ferrante, Ottenstein, and Warren[1] (FOW).
- We modify FOW algorithm to handle pointer chains, global variables, and interprocedural data flow.

[1] The program dependence graph and its use in optimization,
Jeanne Ferrante, Karl Ottenstein, Joe Warren,
ACM Transactions on Programming Languages, 1987

Algorithm to analyze pointers (I) : Compute Data Dependence Graph

1. Compute Data Dependence Graph (DDG) using FOW algorithm.
2. Following additional edges are added.
3. If the variable is a field in the structure, add an edge from the definition of the structure to the use of the field.
4. For every parameter p passed to a callsite of the function f, find the statements where p was used in f and add an edge from the callsite to each of those statements.

Algorithm to analyze pointers (II) : Compute Data Dependence Graph

5. Add an edge from every return statement in function f to every callsite of f.
6. FOW algorithm can handle global variables. Create a unique node for each global variable and maintain the set of nodes for global variables while computing DDG. FOW algorithm takes care of the rest.
7. Label every edge in DDG with the variable that flows along it.

Algorithm to analyze pointers (III): Track the pointer

1. Compute the DDG for every function in the software. This is a one-time cost.
2. Traverse the DDG in forward direction starting at the allocation along the edges associated with the pointer to the allocated memory.
3. If the pointer is a field of a structure, then also include edges associated with the pointer to the structure.
4. The subgraph captured by the forward traversal captures the relevant data flow

Evaluation

Kernel	#Allocations	Results		
		Paired	Partially Paired	Unpaired
5.0	180	**116**	7	57
Distribution	100%	**64.4%**	3.9%	31.7%

Paired - A is paired with D on all feasible execution paths
Unpaired - A is not paired with D on any of the feasible execution paths
Partially Paired - A is paired with D on some of the feasible execution paths, but there exist feasible execution paths without the pairing

Evaluation

Kernel	#Allocations	Results		
		Paired	Partially Paired	Unpaired
5.0	180	**116**	7	57
Distribution	100%	**64.4%**	3.9%	31.7%

Comparison with State-of-the art studies

Tool	Safe (Paired)	Unsafe (Unpaired)	Incomplete Verification
M-SAP (Tamrawi)	35.8%	1.7%	62.5%
LDV (BLAST)	12%	24.3%	63.7%

Visionary Paper - Need for Evidence

- Social Processes and Proofs of Theorems
  - Paper by Richard De Millo, Richard Lipton, and Alan Perlis[1] (DLP)
  - Published in Communications of the ACM, 1979
  - Our work on Verification Evidence was inspired by this paper

[1] First Recipient of the Turing Award

Visionary Paper - Need for Evidence

- A summary of the important points
  - Proof is a social process to increase the confidence of a human in the correctness of theorem. It is not the final word, but only a step in that direction.
  - A good proof makes us wiser. A formal, deductive argument can assert that the program is correct, but it does not tell, to what extent is the program reliable, dependable, and safe. It does not convey the limits within the program will work.
  - DLP argue that the verification must provide knowledge that improves the practice.
  - The verification should produce evidence of its correctness that also enhances our understanding of the program.

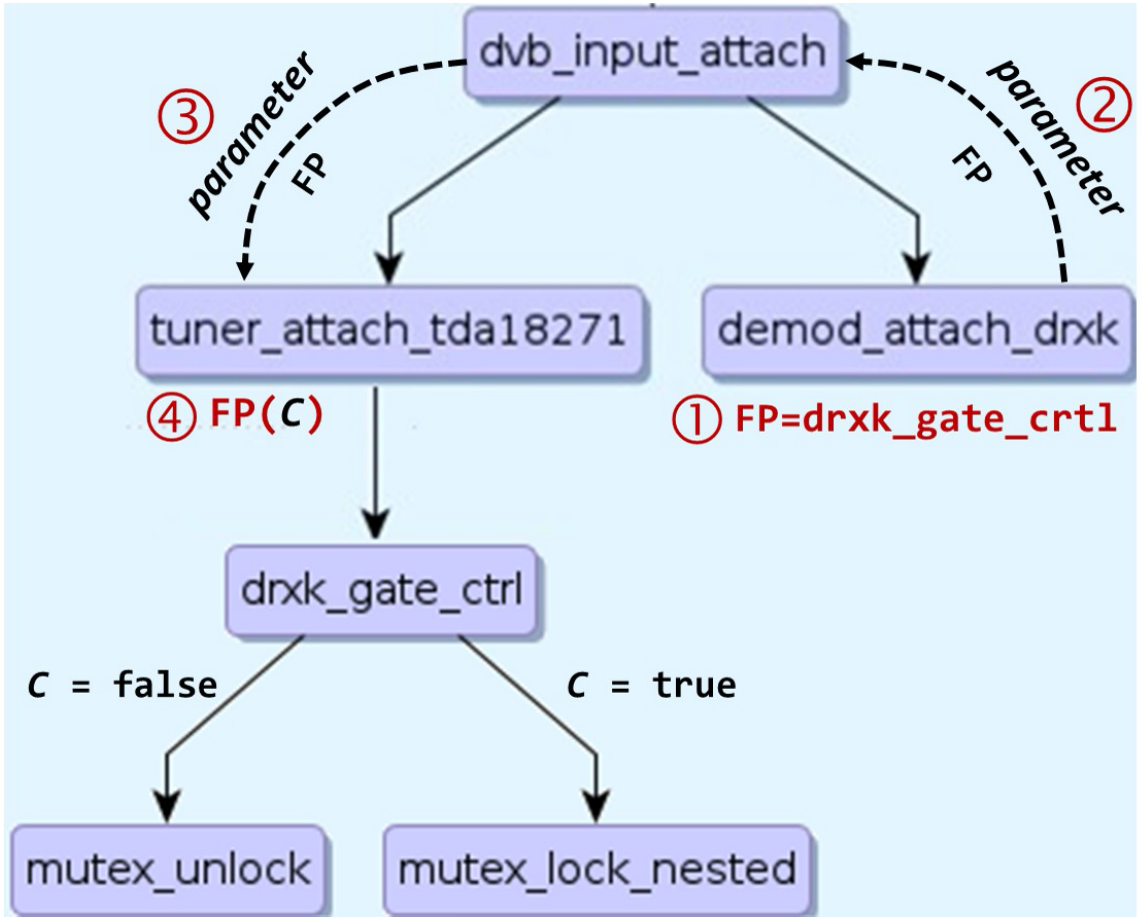
Need for Evidence - Status after 40 years

```
1  drxk_gate_ctrl(int enable) {
2      if (enable) {
3          lock(0);
4      } else {
5          unlock(0);
6      }
7  }
```

- This example is taken from the function `drxk_gate_ctrl` in the Linux kernel.
- BLAST, a state-of-the-art model checker declares this example to be safe.
- The source code shows that the function either locks or unlocks object O, depending upon the value of enable.

What exactly is going on?

Need for Evidence - Status after 40 years



- The function `tuner_attach_tda18271` (f2) calls the function drxk_gate_ctrl (f1) via a 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.

- f1 must be called twice. first to lock then to unlock. The function f2 has a feasible path on which there is a return before the second call to f1 and thus it is a bug.

Need for Evidence - Status after 40 years

- We do not know why BLAST thinks it is a SAFE instance.
  - For *safe* instances, BLAST does not produce any evidence of the correctness of the verification that a human can cross-check.
- Clearly, there is a need for such an evidence.
- Question: What should be captured by this evidence?

Verification Evidence

- We argue that the following information must be captured by the evidence
  - Relevant Functions
  - Data Flow across relevant functions
  - Control Flow Paths
  - Path feasibility

Relevant Functions

- The pointer to the allocated memory can be passed across many functions before it is deallocated.
- Missing even one of them can lead to incorrect verification.
- Thus, the evidence must capture all the functions analyzed by the verifier.

Data Flow across Relevant Functions

- It is necessary to know the mode of the data flow across functions
  - It tells that the relevant functions were correctly captured
  - It tells that the pointer was accurately tracked
  - It also reveals the complexity of the verification
- Evidence must capture the interprocedural data flow due to
  - Parameters
  - Returns
  - Global Variables

Control Flow Paths

- Evidence must capture the following control flow paths
  - paths with deallocation
  - paths on which a pointer to the allocated memory is passed to another function
  - paths without deallocation on which the pointer to the allocated memory is not passed to any function

Path Feasibility

- Evidence must include the set of conditions relevant for checking the feasibility of the control paths without deallocation.

```
1  foo() {
2      p = malloc();
3      if (!p) {
4          return;
5      }
6      free(p);
7  }
```

For example, in the given source code, if the condition on line 3 is true then the function foo returns without deallocating the memory allocated on line 2.

Thus, the condition on line 3 is relevant for path feasibility and should be captured in the evidence.

Evidence Graph

- We developed a novel abstraction, Evidence Graph, that succinctly captures all the information needed to prove the verification.
- Evidence Graph captures
  - Functions analyzed by the verifier
  - Data flow across functions analyzed by the verifier
  - Control flow paths without deallocations
  - Control flow paths on which pointer to the allocated memory is not passed to another function
- For functions with paths without deallocation, we compute Projected Control Flow Graph (PCG) which captures the conditions for path feasibility.

Demonstration of Evidence Graph

- We will now demonstrate the use of Evidence Graph using an example from XINU.
  - XINU is a small operating system developed for education purposes at the Purdue University.
  - Now, XINU is widely used in embedded devices such as routers.
  - This particular example involves
    - Data flow due to parameters
    - Function Pointers
    - Data flow due to Global Variable
    - Path feasibility

```
1   dgwrite(devsw *devptr, xgram *buff, int len) {
2       if(C1) { return; }
3       packet = (struct epacket *) getbuf();
4       if (C2) {
5           if (C3) {
6               ...
7           }
8       } else {
9           if (C4) {
10              freebuf(packet);
11              return(SYSERR);
12          }
13      }
14      return(udpsend(packet));
15  }
```

A

D

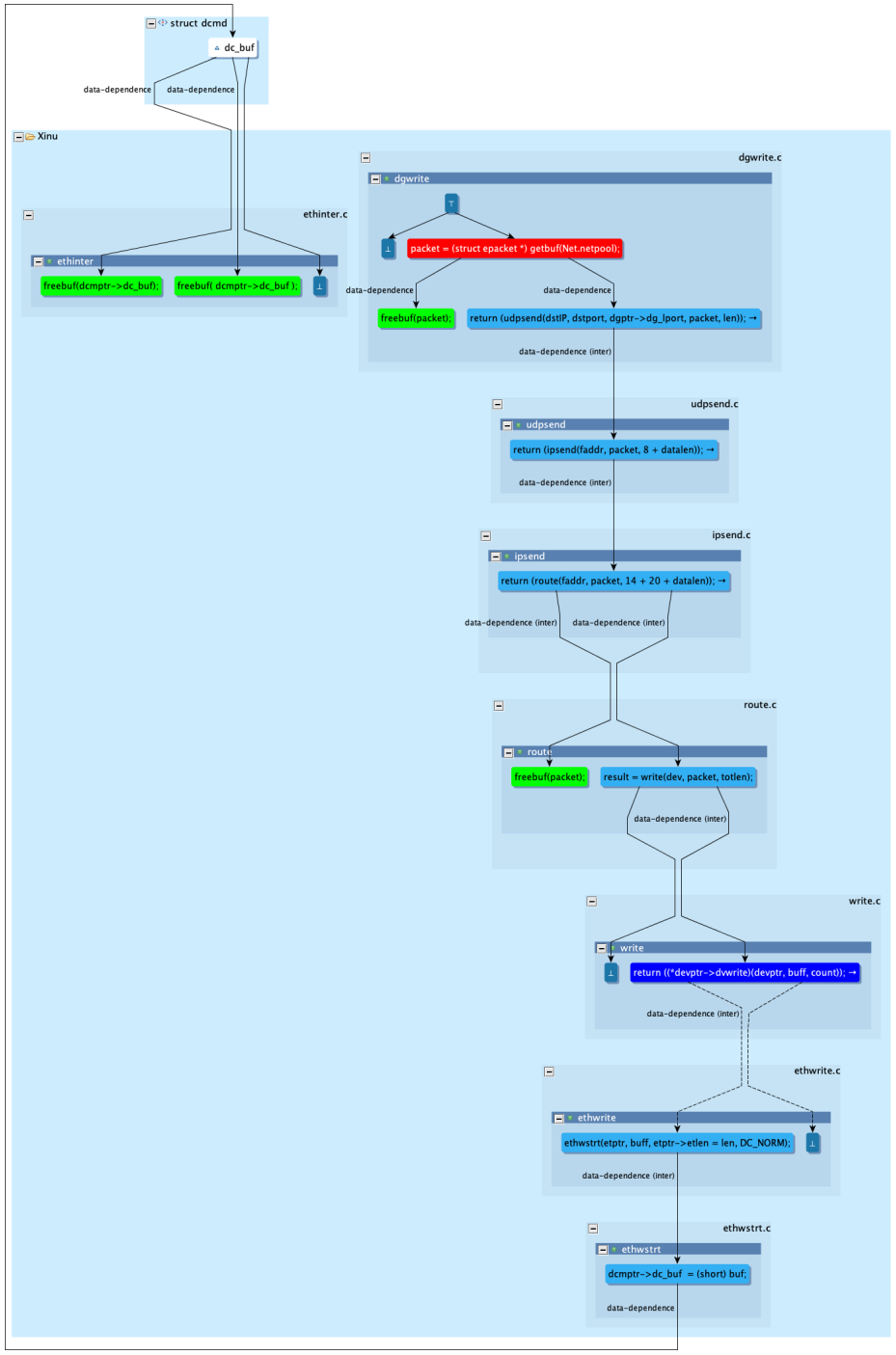
Source code for the example

Line 3 allocates memory and the pointer to the allocated memory is `packet`.

On one control flow path (C1 = false, C2 = false, C4 = true) Line 10 deallocates the memory.

Otherwise, `packet` is passed to the function `udpsend`.

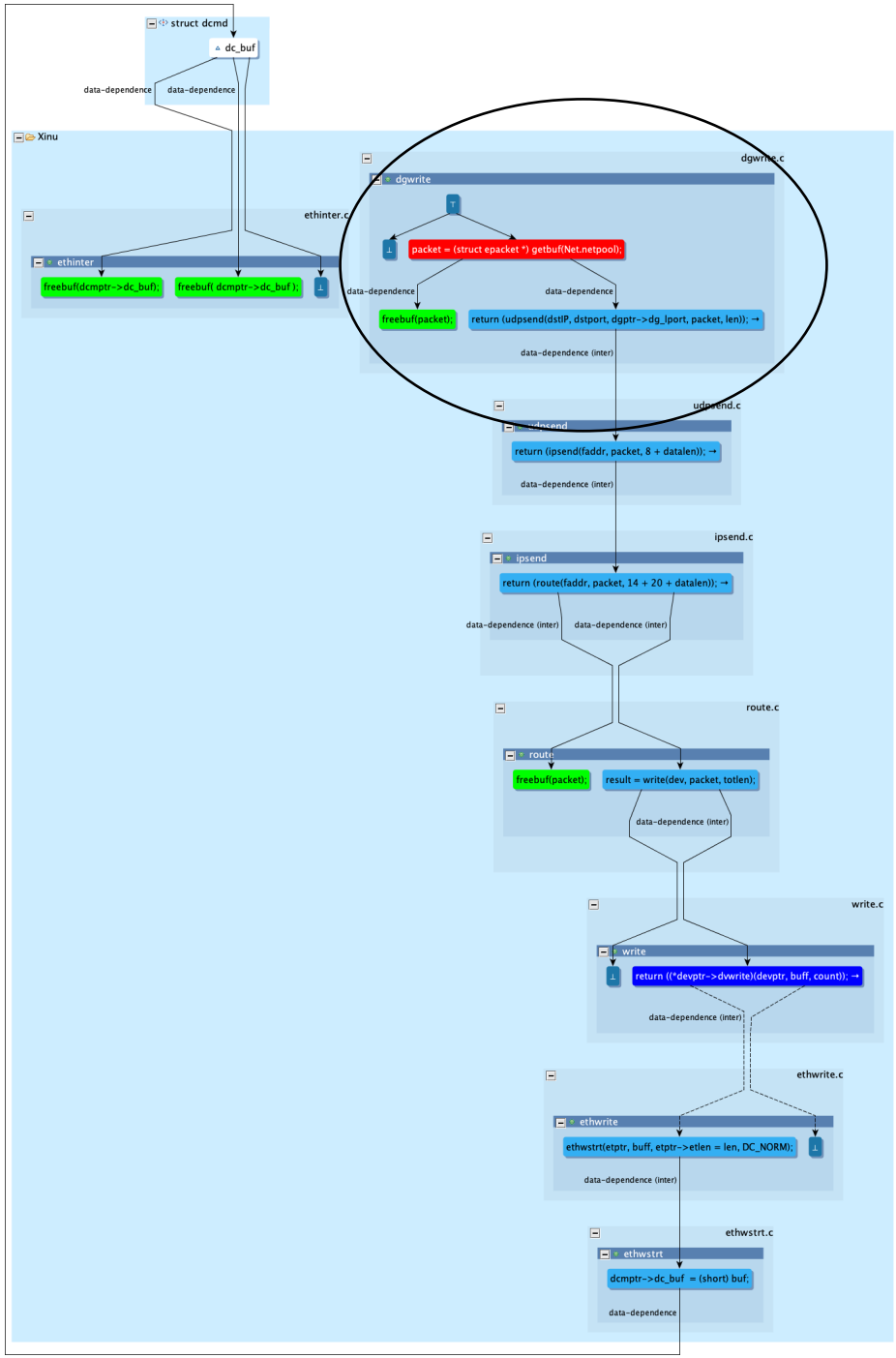
Let's see the Evidence Graph

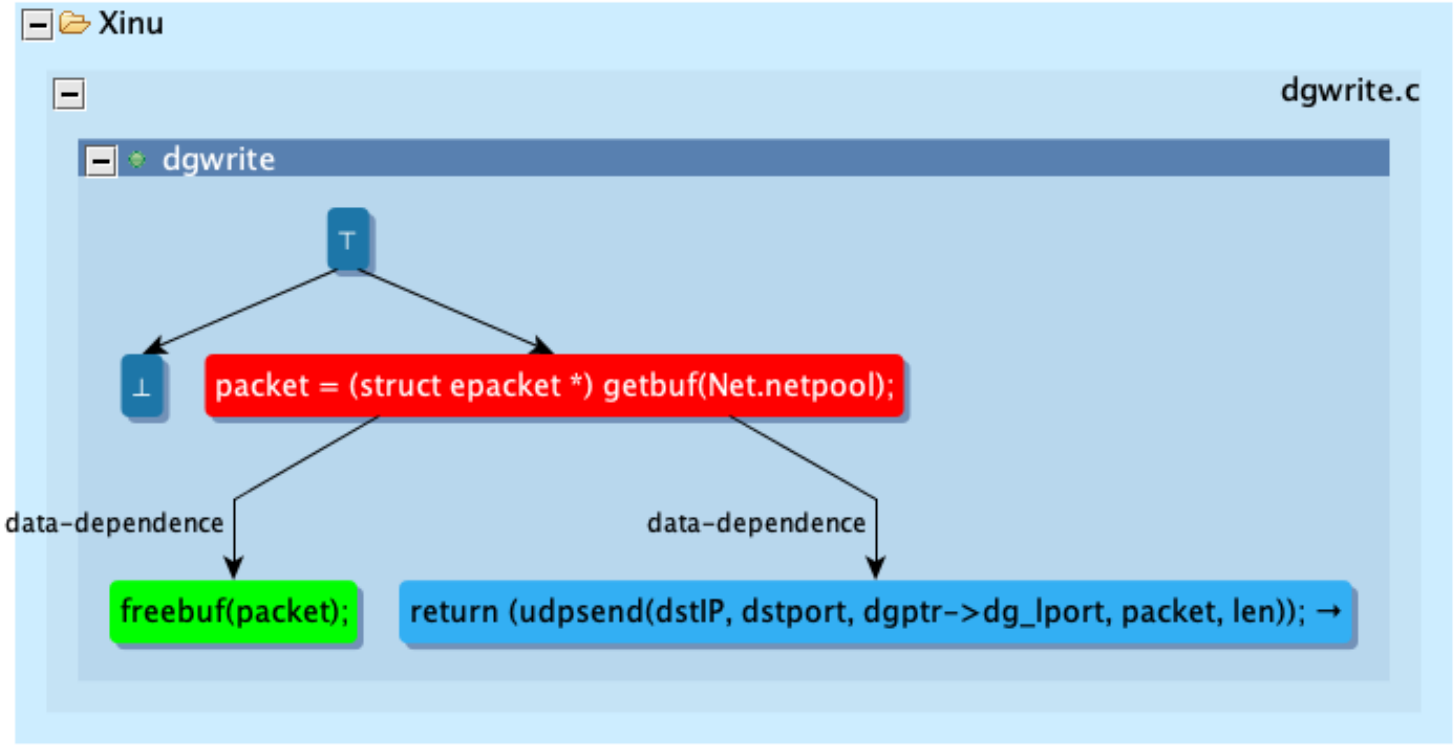


The evidence graph for the example

We will zoom in the individual parts of the graph but before that let's state the highlighting convention

1) Red Node - Allocation
2) Green Node - Deallocation
3) Blue Node - Function Pointer Callsite
4) White Node - Global Variable

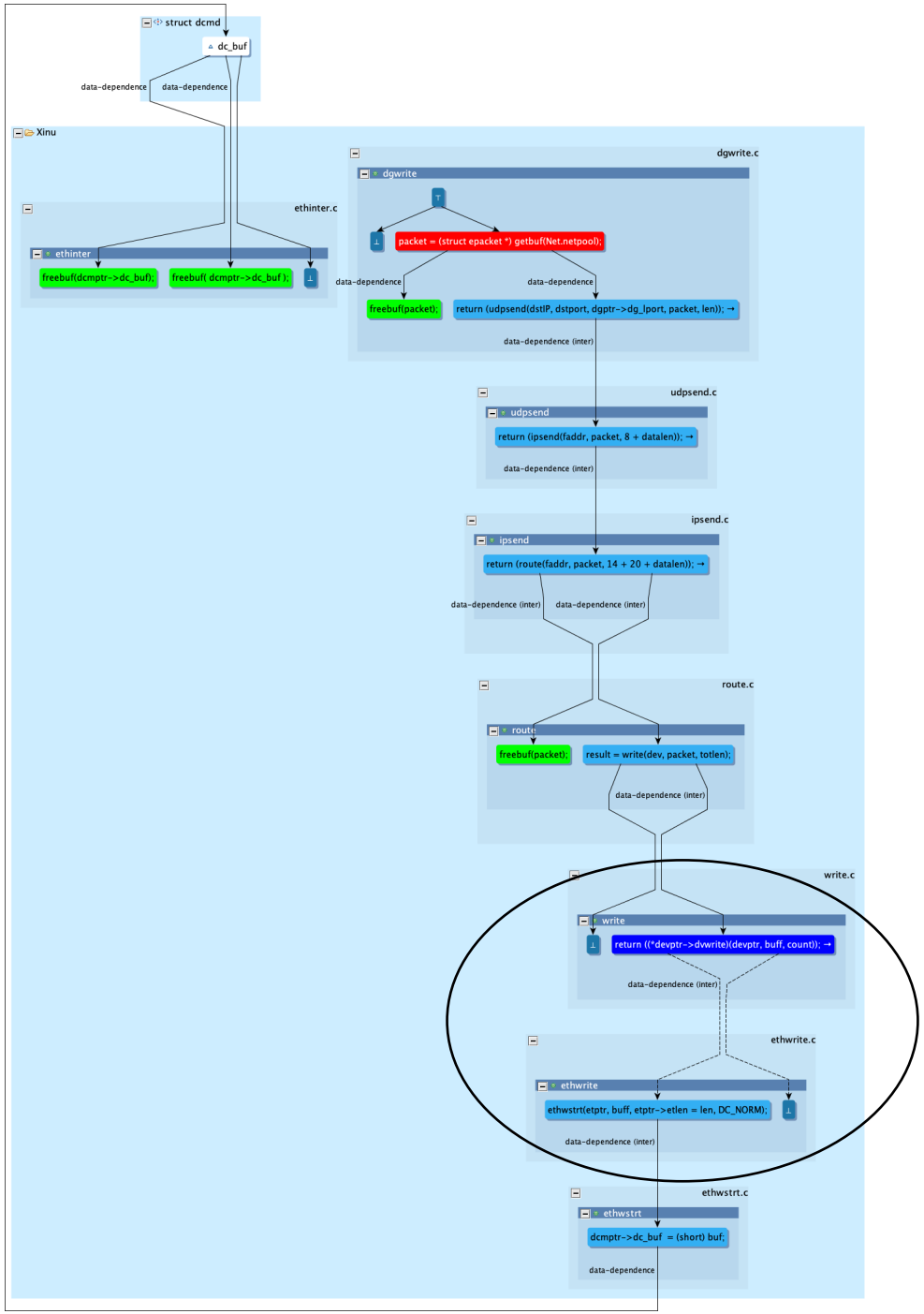

struct dcmd
dc_buf
data-dependence
data-dependence
Xinu
dgwrite.c
dgwrite
ethinter.c
ethinter
freebuf(dcmptr->dc_buf);
freebuf(dcmptr->dc_buf);
packet = (struct epacket *) getbuf(Net.netpool);
data-dependence
data-dependence
freebuf(packet);
return (udpsend(dstIP, dstport, dgptr->dg_lport, packet, len)); →
data-dependence (inter)
udpsend.c
return (ipsend(faddr, packet, 8 + datalen)); →
data-dependence (inter)
ipsend.c
ipsend
return (route(faddr, packet, 14 + 20 + datalen)); →
data-dependence (inter)
data-dependence (inter)
route.c
freebuf(packet);
result = write(dev, packet, totlen);
data-dependence (inter)
write.c
write
return ((*devptr->dvwrite)(devptr, buff, count)); →
data-dependence (inter)
ethwrite.c
ethwrite
ethwstrt(etptr, buff, etptr->etlen = len, DC_NORM);
data-dependence (inter)
ethwstrt.c
ethwstrt
dcmptr->dc_buf = (short) buf;
data-dependence

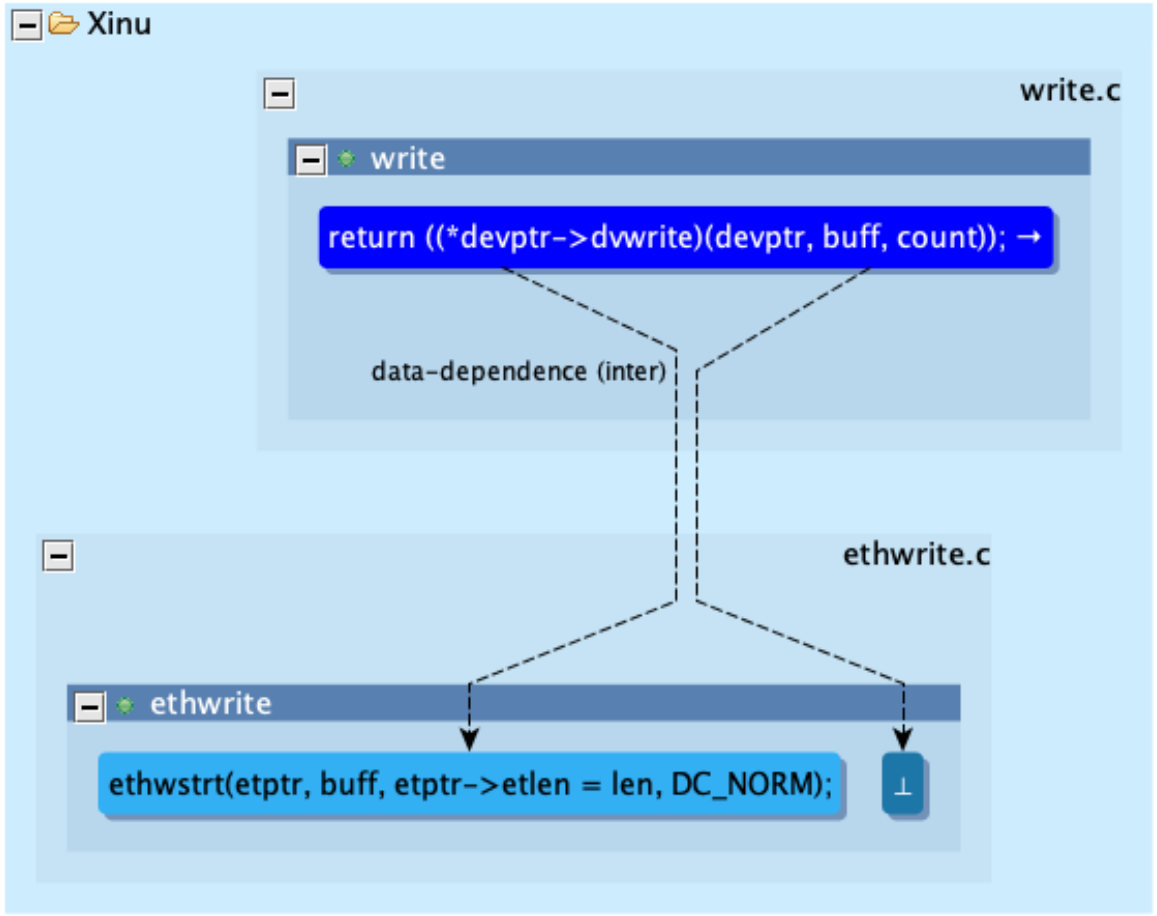


Evidence Graph shows what we observed in the source code for the function `dgwrite`

Memory is allocated and it is deallocated on a path and on others it is passed to `udpsend`.

It also shows that on one path (C1 = true), memory is not allocated

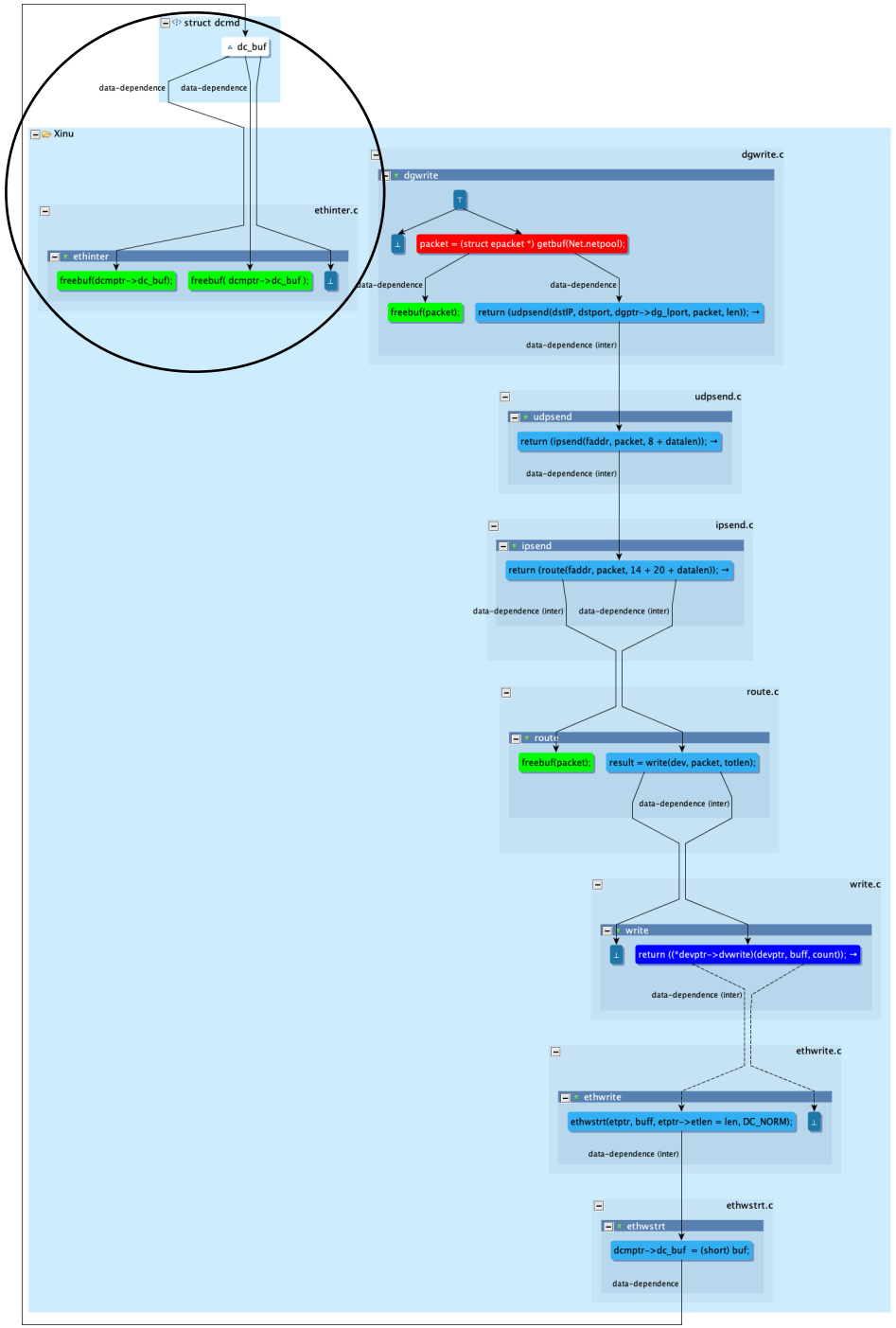

struct dcmd
dc_buf
data-dependence
data-dependence
Xinu
ethinter.c
ethinter
freebuf(dcmptr->dc_buf);
freebuf(dcmptr->dc_buf);
dgwrite.c
dgwrite
packet = (struct epacket *) getbuf(Net.netpool);
data-dependence
data-dependence
freebuf(packet);
return (udpsend(dstIP, dstport, dgptr->dg_lport, packet, len)); →
data-dependence (inter)
udpsend.c
udpsend
return (ipsend(faddr, packet, 8 + datalen)); →
data-dependence (inter)
ipsend.c
ipsend
return (route(faddr, packet, 14 + 20 + datalen)); →
data-dependence (inter)
data-dependence (inter)
route.c
route
freebuf(packet);
result = write(dev, packet, totlen);
data-dependence (inter)
write.c
write
return ((*devptr->dvwrite)(devptr, buff, count)); →
data-dependence (inter)
ethwrite.c
ethwrite
ethwstrt(etptr, buff, etptr->etlen = len, DC_NORM);
data-dependence (inter)
ethwstrt.c
ethwstrt
dcmptr->dc_buf = (short) buf;
data-dependence

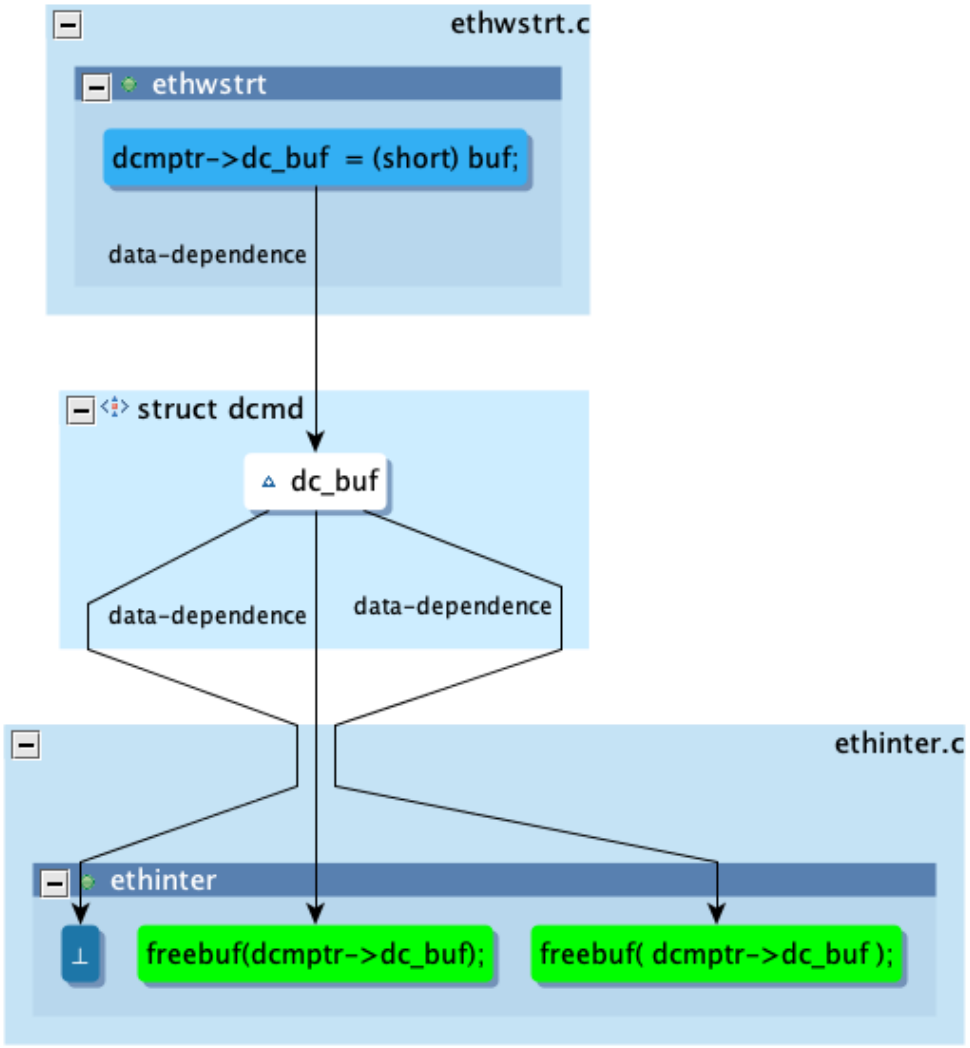


Evidence Graph shows that eventually `packet` is passed to the function `write`. write passes it to a function pointer.

Evidence Graph shows that the verifier resolved the function pointer to the function `ethwrite`. If needed, a human can check that the function pointer was correctly resolved.

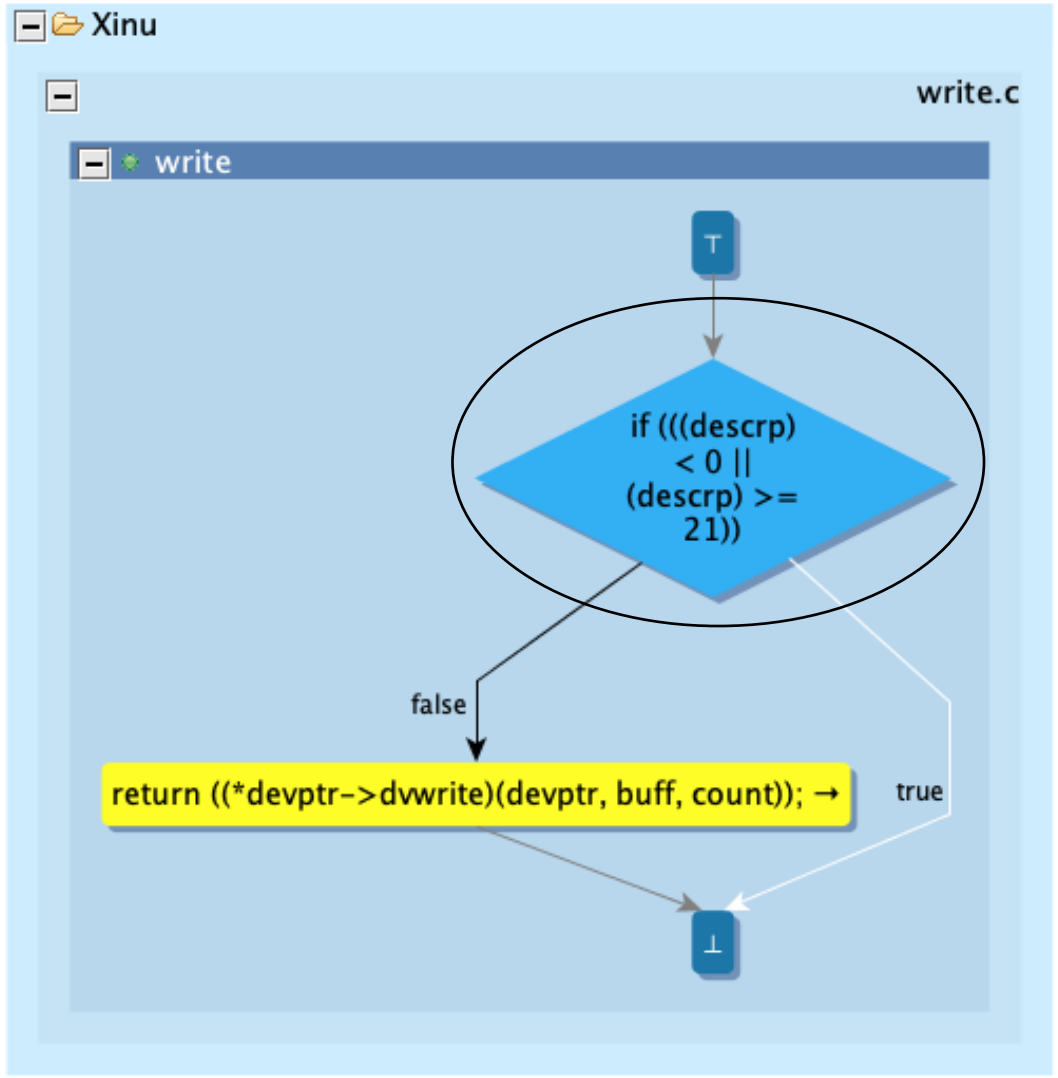
It also shows that there is a path on which memory is neither deallocated nor passed to another relevant function.


struct dcmd
dc_buf
data-dependence
data-dependence
Xinu
ethinter.c
ethinter
freebuf(dcmptr->dc_buf);
freebuf(dcmptr->dc_buf);
dgwrite.c
dgwrite
packet = (struct epacket *) getbuf(Net.netpool);
data-dependence
data-dependence
freebuf(packet);
return (udpsend(dstIP, dstport, dgptr->dg_lport, packet, len)); →
data-dependence (inter)
udpsend.c
udpsend
return (ipsend(faddr, packet, 8 + datalen)); →
data-dependence (inter)
ipsend.c
ipsend
return (route(faddr, packet, 14 + 20 + datalen)); →
data-dependence (inter)
data-dependence (inter)
route.c
route
freebuf(packet);
result = write(dev, packet, totlen);
data-dependence (inter)
write.c
write
return ((*devptr->dvwrite)(devptr, buff, count)); →
data-dependence (inter)
ethwrite.c
ethwrite
ethwstrt(etptr, buff, etptr->etlen = len, DC_NORM);
data-dependence (inter)
ethwstrt.c
ethwstrt
dcmptr->dc_buf = (short) buf;
data-dependence



Evidence Graph shows that eventually packet is passed to the function `ethwstrt`. `ethwstrt` stores packet in the field `dc_buf` of a global variable. The pointer to the global variable is `dcmptr`

Evidence Graph shows that the function `ethinter` accesses the global variable to deallocate the memory.



Lastly, we show PCG of the function `write`, a function which has a path on which the `packet` is neither deallocated nor passed to another function.

PCG shows that is indeed the case and depending upon the circled condition, either `packet` is passed to a function pointer or it returns without deallocating the memory.

The circled branch condition is the relevant condition for path feasibility and it is captured by the PCG.

Evaluation

Kernel	#Allocations	Results		
		Paired	Partially Paired	Unpaired
5.0	180	**116**	7	57
Distribution	100%	**64.4%**	3.9%	31.7%

- We verified the results of all the 180 examples.
- All the paired instances are correctly verified.
- Partially Paired instances are not bugs. Need a better Path Feasibility Check and then they will be verified as Paired.
- Unpaired instances reveal further complexities that need to be addressed.

Conclusions and Future work

- We developed a new Data flow algorithm to track pointers for MADP
- We report a significant improvement of MADP results for the Linux kernel.
- We developed a novel abstraction, Evidence Graph, to generate evidence of the correctness of the verification.
- Future work: MADP results can be further improved by modeling pointer arithmetic.

Questions?