Let’s Verify Linux: Accelerated Learning of Analytical Reasoning through Automation and Collaboration[*]

Suresh Kothari and Ahmed Tamrawi
Iowa State University
Ames, Iowa
{kothari, atamrawi}@iastate.edu

Jeremías Sauceda and Jon Mathews
EnSoft Corp.
Ames, Iowa
{pi, jmathews}@ensoftcorp.com

ABSTRACT

We describe our experiences in the classroom using the internet to collaboratively verify a significant safety and security property across the entire Linux kernel. With 66,609 instances to check across three versions of Linux, the naive approach of simply dividing up the code and assigning it to students does not scale, and does little to educate. However, by teaching and applying analytical reasoning, the instances can be categorized effectively, the problems of scale can be managed, and students can collaborate and compete with one another to achieve an unprecedented level of verification.

We refer to our approach as Evidence-Enabled Collaborative Verification (EECV). A key aspect of this approach is the use of *visual software models*, which provide mathematically rigorous and critical evidence for verification. The visual models make analytical reasoning interactive, interesting and applicable to large software.

Visual models are generated automatically using a tool we have developed called $\mathcal{L}$-SAP [14]. This tool generates an Instance Verification Kit (IVK) for each instance, which contains all of the verification evidence for the instance. The $\mathcal{L}$-SAP tool is implemented on a software graph database platform called Atlas [6]. This platform comes with a powerful query language and interactive visualization to build and apply visual models for software verification.

The course project is based on three recent versions of the Linux operating system with altogether 37 MLOC and 66,609 verification instances. The instances are accessible through a website [2] for students to collaborate and compete. The Atlas platform, the $\mathcal{L}$-SAP tool, the structured labs for the project, and the lecture slides are available upon request for academic use.

*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.

ICSE ’16 Companion, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4205-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2889160.2889192

1. INTRODUCTION

With so much critical infrastructure depending on software, the time has come to teach students the analytical reasoning skills required to find software safety and security problems.

The challenges of verifying our software infrastructure are daunting, in part because of the complexity of the software, but also due to the sheer volume of it. The Linux kernel alone, which provides the basis for so many devices (web servers, routers, smart phones, desktops), is over 12 MLOC. How can anyone hope to verify this mountain of code?

And yet we must teach students to work with large software. Teaching software engineering with toy examples is akin to teaching skyscraper construction with bird houses: the basic points can be illustrated clearly, but issues inherent in large scale software are sorely missed.

With this in mind, we have created advanced algorithms and tools for model-based reasoning in our DARPA research on cybersecurity of large software. The technology from our research is offered here to enable education to solve software problems of size and complexity that are otherwise impossible to tackle in a classroom.

This paper presents a case study of a verification problem characterized as Matching Pair Verification (MPV). MPV is broadly applicable to memory leak, confidentiality leak, and other safety and security vulnerabilities. The specific course project we describe is to verify correct pairing, on all execution paths, of `mutex` lock and `spin` lock with corresponding unlocks.

With 66,609 verification instances to check inside 37 MLOC, we cannot simply divide up the cases among the students. Instead, we take an approach we call Evidence-Enabled Collaborative Verification (EECV). A key aspect of this approach is the use of *visual software models*, which provide mathematically rigorous and critical evidence for verification. These models are generated automatically, and always include evidence which aids a human in verification. The models are categorized in a way that causes irregular cases to stand out, while making the code dependencies necessary for further investigation immediately accessible. This accomplishes several goals: it teaches the students what normal patterns of software architecture are, it allows them to focus their auditing efforts on irregular instances, and, most importantly, teaches them the analytical reasoning skills required to decide whether an instance is a bug or a feature.

As a direct result of this approach, we have discovered 8 synchronization bugs which have been reported to, and confirmed by, the Linux community. By using this approach in the classroom, dozens of students sharpened their analytical reasoning skills.

The paper is organized as follows. Section 2 provides an overview of Evidence-Enabled Collaborative Verification (EECV) and an EECV class project. Section 3 provides examples of structured labs that teach fundamentals of visual models for intra-procedural and inter-procedural evidence. Section 4 provides results from our case study reviewing the Linux kernel. Section 5 provides an overview of the enabling technology of the graph database platform, the graph schema and its use for creating visual models, and examples of how visual models are created. Sections 6 and 7 describe related work and the conclusion.

2. APPROACH OVERVIEW

This section provides an overview of the EECV approach: visual models as verification-critical evidence, the $\mathcal{L}$-SAP tool, Internet-enabled collaboration, and the enabling technology to support analytical reasoning with large software.

Before going into an overview of our approach, we briefly review the verification problems we address.

2.1 Matching Pair Verification (MPV)

We characterize MPV as the 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* of information for a confidentiality breach in the context of cybersecurity. In Section 4, we present examples from the verification of locking and unlocking in the Linux kernel.

The major challenges in verifying an instance of an MPV problem are: (a) inter-procedural complexity, e.g. tracking relevant functions is crucial for completeness, but it is challenging because the control flow across functions may not be explicit due to interrupts, multithreading, and function pointers, (b) intra-procedural complexity, e.g. path-sensitive analysis is crucial to avoid false negatives but such analysis is challenging because the control flow paths proliferate exponentially due to branch points, and (c) path feasibility complexity, e.g. checking path feasibility is crucial to avoid false positives but the feasibility check is challenging because it involves the NP-complete *satisfiability problem.*

2.2 Evidence-Enabled Collaborative Verification - Visual Models

The approach we use to verify MPV problems is Evidence-Enabled Collaborative Verification (EECV).

EECV can be rigorous, computationally efficient, and automated enough to keep the human effort within reasonable limits, but it does not have to be *completely* automated. The automation enables and simplifies human cross-checking, which is especially important when the stakes are high.

We present *visual modeling* as a medium for students to understand and apply the fundamentals of data and control flow analyses. The complexity of software is rooted in its own version of the *butterfly effect* [7, 11]. A small change at one point can impact many parts of the software and cause an unforeseen effect at a very distant point of the software. This impact propagation is hard to decipher from software viewed as lines of code; it makes program comprehension and reasoning tedious, error-prone, and almost impossible to scale to large software. A visual model makes impact propagation explicit, amplifies and empowers human intelligence to reason about large software.

Specifically, we will present two models: the Matching Pair Graph (MPG) [8] for managing inter-procedural complexity due to impact propagation, and the Event Flow Graph (EFG) [15] (derived from the Control Flow Graph (CFG)) for managing intra-procedural complexity of control flow paths explosion. Both models enable powerful program analysis, reasoning, and visualization with an unprecedented level of interaction - which is important for students to understand non-trivial program analysis and reasoning concepts.

These concepts are implemented in our tool, $\mathcal{L}$-SAP [14].

2.3 Internet-enabled Collaboration with $\mathcal{L}$-SAP

The benefits of teaching students model-based reasoning and collaborative verification are fairly easy to understand in principle, but not easy to realize in practice. DARPA funded research projects have enabled us to create robust tools that enable our approach on large software.

2.3.1 Creating Verification Instances

The $\mathcal{L}$-SAP tool breaks the verification problem into verification instances that get automatically posted on a website [2] for collaboration and competition. $\mathcal{L}$-SAP automatically verifies as many instances as possible with a strong inherent guarantee of correctness. The $\mathcal{L}$-SAP verification results fall into three categories:

- $\mathcal{C}1$: the automatically verified instances with no violation,
- $\mathcal{C}2$: automatically verified instances with one or more violations shown by missing unlock(s) on feasible path(s),
- $\mathcal{C}3$ the remaining instances where the verification is inconclusive.

Each verification instance corresponds to a lock call site. A lock call $L(o)$ can be paired with an unlock call $U(m)$ iff the objects o and m are the same.

Instance Signature. We use the notion of a type-based *signature* as an approximation to match the objects for the lock and unlock. Signatures facilitate the computationally efficient pairing algorithm implemented in $\mathcal{L}$-SAP. They also help analytical reasoning. The signatures work well in practice but it is not a perfect mechanism. The students must watch for errors caused by imperfect signatures. Signatures are commonly used in manual verification by experienced analysts. $\mathcal{L}$-SAP derives them automatically and offers them as a part of the evidence.

The signatures work as follows. Consider the pointer $\mathcal{P}$ given by the expression: $(a_n \cdots a_3(\texttt{.}||\texttt{->})a_2(\texttt{.}||\texttt{->})a_1)$. In this expression, $\mathcal{P}$ is being accessed through a chain of member-selection C operators (`.` and/or `->`). We define the *hierarchal type* for $\mathcal{P}$ as the tuple $(T_{a_n}, \cdots, T_{a_3}, T_{a_2}, T_{a_1})$, where T_{a_i} denotes the type associated with the member a_i. For example, the hierarchal type for pointer `x->y->z` is given by the tuple (X, Y, Z) where $T_x = X, T_y = Y$, and $T_z = Z$. For a directly referenced pointer, the hierarchal type is the same as its type. For example, the hierarchal type for pointer `k` is T_k. The *object signature* (S_o) denotes: (1) the object (variable) *name* if o is a global variable, and (2) the *hierarchal* *type* of o otherwise. A lock call $L(o)$ is mapped to unlock call $U(m)$ iff $S_o = S_m$.

2.3.2 Instance Verification Kit (IVK)

Verification instances are bundled into an IVK. The IVKs are designed as a vehicle to integrate automation and human intelligence to solve the verification problem. The IVKs are produced automatically and they provide verification-critical, compact evidence for the human analyst to reason with. Each IVK includes: (a) the source location of the lock call, (b) the instance signature, (c) the source locations of the paired unlock calls, and (d) the visual models MPG (Section 3.2), CFG, and EFG (Section 3.3).

2.3.3 The Collaboration Website

We use a website [2] to make it easy for students to collaborate and compete. Figure 1 shows the hierarchy of the generated website. Figure 1(a) allows the students to select the Linux kernel version of interest. Figure 1(b) lists all the `spin` and `mutex` lock calls and their source correspondence for the selected version. Figure 1(c) displays the IVK associated with each lock call.

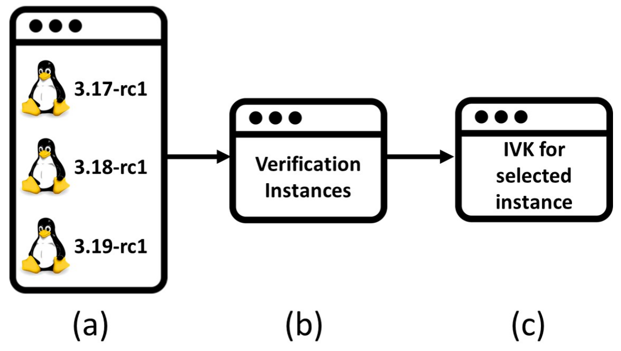


Figure 1: IVKs website hierarchy

2.3.4 Classroom Workflow

In the first part of the project, each student team is assigned a subset of verification instances from each of the above three categories. Their goal is to cross-check the results in categories $\mathcal{C}1$ and $\mathcal{C}2$.

The second part of the project is open-ended where each team works on instances of their choice from category $\mathcal{C}3$ to show that they can handle a wide spectrum of verification challenges. For example, a challenge could be that the function `f` with the lock call and the function `g` with a corresponding unlock call are not connected by a call chain. This can happen when `g` is called by an interrupt service routine. The MPG (Section 3.2) and EFG (Section 3.3) reflect various challenges and thus assist the students in selecting the instances that cover a wide spectrum.

The third part of the project is to audit the verification results posted by other teams using a specified standard format. Students audit the results for errors or incompleteness. For example, an audit can find that the proof is incorrect because the path feasibility check is wrong or the proof is incomplete because the global variable through which the lock object is communicated for asynchronous unlocking is not identified.

2.3.5 Interactive Reasoning

For verifying the harder cases, interactive reasoning becomes especially important. Interactive reasoning can be performed using visual models and their source correspondence. The interactions can go from one visual model to another one, go from a visual model node to its corresponding source code, or go from source code to a corresponding visual model. One can click on a function node in a visual model (e.g., call graph (CG)) to open up the CFG for the selected function in CG and observe the control flow paths within the function. The CFG nodes correspond to statements in the source code. One can click on a CFG node to observe the corresponding source code. Similarly, one can click on source code to invoke a corresponding visual model.

Atlas provides a shell and one can interactively query and mine programs to gather additional evidence. For example, the CG does not cover the functions that are called using function pointers. While auditing a verification instance, the students may suspect that the verification is incomplete due to a function `f` missing in the CG because it may have been called using a function pointer. The students can issue queries to find such a function.

Atlas also provides Smart Views to provide instant feedback and interactive software graph visual models as the student clicks on code artifacts or other visual models. A number of out-of-the-box smart views are provided for common queries for building call graphs, data flow graphs, type hierarchies, dependency graphs, and many other useful results. For example, when a student clicks on a function either on a visual model or source code, the call graph smart view (if selected) will instantly produce the call graph for the selected function. The produced smart view appears on a side tab that does not interfere with the student and keeps him focused on the current task.

3. PREPARING THE STUDENTS

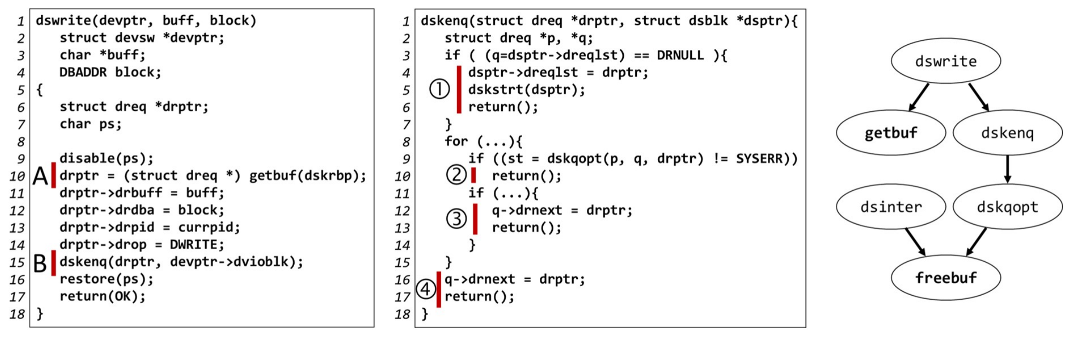
Before beginning the verification process, students go through a series of structured labs designed to teach them the fundamentals of data and control flow analyses, and to understand the MPG and EFG visual models in depth. It is easy to use the visual models mechanically but that would be somewhat like using a calculator without understanding addition. The visual models involve non-trivial abstractions derived by combining graph theory and program analysis, thus it is critical that the students understand the models before applying them.

For the labs, we use XINU [3], a small operating system with about five thousand lines of code and 200 functions. XINU is complex enough to bring out key verification challenges. The labs bring out various challenges for pairing a memory allocation call with a corresponding deallocation call. In XINU, `getbuf` and `freebuf` are respectively the allocation and the deallocation functions.

The starting point is the `dswrite` function shown in Figure 2(a)). Function `dswrite` calls `getbuf` but it does not call `freebuf`. The verification problem is to match the `getbuf` call in `dswrite` with the corresponding `freebuf` call(s).

3.1 Motivating the use of Visual Models

We will discuss an example that motivates the use of MPG. As seen from the `dswrite` code shown in Figure 2(a), `dswrite` calls and passes the allocated memory pointer `drptr`



```
dswrite(devptr, buff, block)
    struct devsw *devptr;
    char *buff;
    DBADDR block;
{
    struct dreq *drptr;
    char ps;

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

(a) Function dswrite

```
dskenq(struct dreq *drptr, struct dsblk *dsptr){
    struct dreq *p, *q;
    if ( (q=dsptr->dreqlst) == DRNULL ){
        dsptr->dreqlst = drptr;
  ①     dskstrt(dsptr);
        return();
    }
    for (...){
        if ((st = dskqopt(p, q, drptr) != SYSERR))
      ②    return();
        if (...){
      ③    q->drnext = drptr;
            return();
        }
    }
④   q->drnext = drptr;
    return();
}
```

(b) Function dskenq

(c) Evidence

Figure 2: The importance of evidence to reason about the possibility of a memory leak in the function `dswrite`

to function `dskenq`. So, function `dskenq` (Figure 2(b)) must be examined. The function `dskenq` introduces difficult verification challenges.

As shown in Figure 2(b), `dskenq` has four paths. In path 1: `drptr` is passed to function `dskstrt` which does not free the allocated memory. But, we cannot conclude that it is a memory leak because on the same path `drptr` is assigned to `dsptr->dreqlst` where `dsptr` (passed as a parameter to `dskenq` from `dswrite`) points to a global data structure. In path 2: `drptr` is passed to function `dskqopt`. In paths 3 and 4: `drptr` is assigned to `q->drnext` and earlier `q` is set to points to `dsptr->dreqlst`. Thus, on three paths (1, 3 and 4) the allocated memory is not freed but the pointer to the allocated memory becomes accessible through a global data structure.

Since the pointer to the allocated memory gets passed as a parameter to other functions, the call chains must be tracked. Moreover, one path in `dswrite` multiplies into 4 paths in `dskenq` and the path proliferation continues through functions down the call chain. Tracking the call chains and the proliferation of paths gets tedious and challenging. It is most challenging when the pointer to the allocated memory is assigned to a global variable. Then, any other function could free the memory. It is then a huge verification challenge to figure out the relevant functions. To make matters worse, these relevant functions may operate asynchronously and thus we cannot reach them following a call chain. There is a dire need to assist the human analyst to address this difficulty. The MPG comes to rescue here by providing the crucial evidence to figure out the relevant functions.

3.2 MPG for Inter-procedural Evidence

We have a structured lab to teach how the Matching Pair Graph (MPG) [8] serves as valuable evidence and simplifies the human effort. By design, the MPG is a directed graph with edges representing function calls and the roots of MPG are asynchronous functions; they either belong to different threads or some of them may be called through interrupts. From the several hundred XINU functions, the MPG shown in Figure 2(c) has narrowed down the relevant functions to 6.

Besides producing a small set of functions, the MPG (Figure 2(c)) provides other very valuable pieces of evidence. The MPG shows a call chain from `dswrite` to `freebuf` which indicates the possibility of execution paths on which the pointer to the allocated memory is passed as a parameter, eventually to function `dskopt` which deallocates the memory. More importantly, the MPG includes `dsinter` function which turns out to be a very important clue. `dsinter` is not connected to `dswrite` by forward or reverse call chains.

Strangely, `dsinter` calls `freebuf` but *not* `getbuf`, which is actually a critical clue. The student can hypothesize that `freebuf` in `dsinter` can potentially pair with the `getbuf` in `dswrite`. The student can also observe that `dsinter` and `dswrite` must operate asynchronously and communicate through a global data structure `D` because the two functions are roots of the MPG. To prove the hypothesis, the student must identify the global structure `D` and complete the verification. The student has a good suspect for `D`, namely the data structure to which the allocated memory pointer `drptr` is assigned. This makes it easy for the student to complete the verification.

This example illustrates how the MPG can be of tremendous assistance for analytical reasoning. The example presents a case not amenable to automation because of asynchronous processing. An automated pointer analysis would also hit a barrier in this example because the allocated memory pointer `drptr` is inserted in a global linked list `D`. `dsinter` draws the pointers from the global list and frees the memory for each pointer. And the fact that it does so until the list becomes empty shows that it is not a memory leak.

This is not a wayward example. It stems from the well-known *producer-consumer* pattern [16], a classic example of a multi-process synchronization.

3.3 EFG for Intra-procedural Evidence

We have a structured lab to teach how the Event Flow Graph (EFG) [15] addresses the challenges of: (a) an exponentially-increasing number of paths in a control flow graph (CFG), and (b) checking feasibility of paths in a CFG. EFG addresses these challenges by introducing an *equivalence relation* on the CFG paths to partition them into equivalence classes. It is then sufficient to perform analysis on these equivalence classes rather than on the individual paths in a CFG. The EFG has two major advantages over its corresponding CFG: (a) although the number of paths in a CFG can be exponentially large, the essential information to be analyzed is captured by a small number of equivalence classes, and (b) checking path feasibility becomes simpler.

The EFG is a compact derivative of its corresponding CFG. Each path in the EFG represents an equivalence class of paths in the CFG. The EFG is defined with respect to events relevant to the verification. The equivalence classes are thus guaranteed to preserve all the event traces in the original CFG. In `dswrite` example, the events relevant to the verification include allocation and deallocation (`getbuf` and `freebuf`) calls, passing the pointer to the allocated memory (e.g., `dskenq(drptr)`), and assigning the pointer to the allocated memory to another variable (e.g., `dsptr->dreqlst = drptr`).

The EFG for the function `dskenq` is shown in Figure 3. As shown in Figure 2(b), `dskenq` has four paths to exit the function. All four paths are retained in the EFG because each of these paths has a unique sequence of events relevant to the verification.

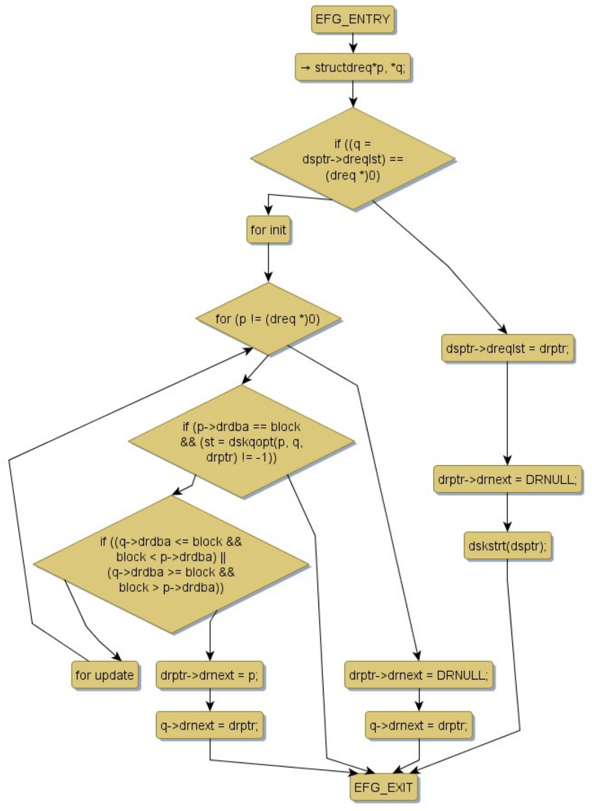


Figure 3: EFG for the function `dskenq`

Figure 4 shows an example of a CFG and its corresponding EFG. The EFG is constructed with respect to lock and unlock events which are relevant to verify the correct pairing of lock and its corresponding unlocks on all execution paths. This example illustrates how the EFG can greatly simplify the verification task. The EFG has only one path corresponding to the CFG paths that contain the same sequence of events relevant to the verification. The CFG in Figure 4(a) has 5 branch nodes resulting in 8 paths after the lock. Some of these paths go through a complex loop with two exits. The 4 out of 5 branch nodes are irrelevant to the verification because all the paths branching from them lead to the unlock and are thus equivalent. These 4 branch nodes get eliminated in the EFG and the 7 paths are represented by a single path in the EFG. Thus, the EFG simplifies the verification task by compacting the CFG.

The EFG also simplifies the path feasibility check. As seen

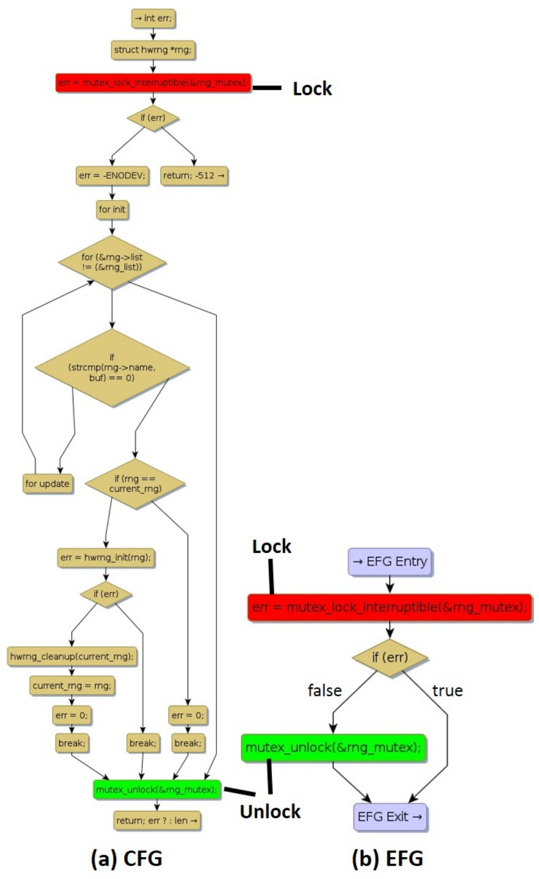


Figure 4: CFG and EFG for the function `hwrng_attr_current_store`

from the EFG in Figure 4(b), there is a path with missing unlock and the feasibility of that path must be checked. If feasible, it is a violation otherwise the particular instance of the lock is correctly verified. The EFG has retained only the condition that is necessary to verify the path feasibility, the other 4 conditions from the CFG are not retained in the EFG. The student can easily cross-check the verification by observing that if the lock is granted then the particular condition is `false`. So, the `true` path in Figure 4(b) is not feasible and thus it is not a violation.

3.4 Learning MPG Fundamentals

The purpose of this structured lab is to teach students the theory behind MPG and relate it to the fundamentals of data and control flow analyses. The students gain an hands-on experience in evolving ideas that culminate in MPG. The students use the Atlas query language to implement the ideas and visualize the results. We have evolved the concept of MPG from our previous work [8] where it was first introduced.

As we observed from the `dskwrite` and `dskenq` code examples, the matching `freebuf` call(s) can exist in many different functions and somehow the pointer to the allocated memory is communicated to these functions. The MPG is essentially the collection of these functions and others that communicate the pointer. The MPG is an efficient mechanism that gathers for a particular lock call, all functions that should be examined, and it does so all at once without requiring an analyst to traverse through each function.

Students learn the different ways functions communicate through: parameters, return, and global variables. If it was only through parameters, the functions could be gathered by going through the call tree. But because of the other two modes of communication, the call tree is not enough.

Students are then taken through a progression of ideas from the paper [8] to arrive at a good notion of MPG. The first idea is the *reverse call graph* (**rcg**) of the functions: `getbuf` and `freebuf`. The next refinement is an induced subgraph of the ***rcg*** to include only the functions that *read* or *write* to `dreq`, the type of the structure for which the memory is allocated. The next refinement is yet another induced subgraph that separates `dswrite` from the other functions that also allocate memory for structures of type `dreq`. It is easy to construct these refined models using a conceptual sequence of queries:

(Q_1) `var m1 = rcg(getbuf, freebuf);` - `rcg` is stored in variable `m1`.

(Q_2) `show(m1)` - shows the `rcg` visual model.

(Q_3) `var temp = ref(dreq);` - `temp` captures the functions that read or write to `dreq`.

(Q_4) `var m2 = m1.intersection(temp);` - `m2` is the second refined model.

(Q_5) `show(m2)` - shows the refined visual model.

Figures 5(a) and (b) show the visual models produced by queries Q_2 and Q_5. Producing the third refined model (Figure 5(c)) requires a little more work which is left as an exercise for students.

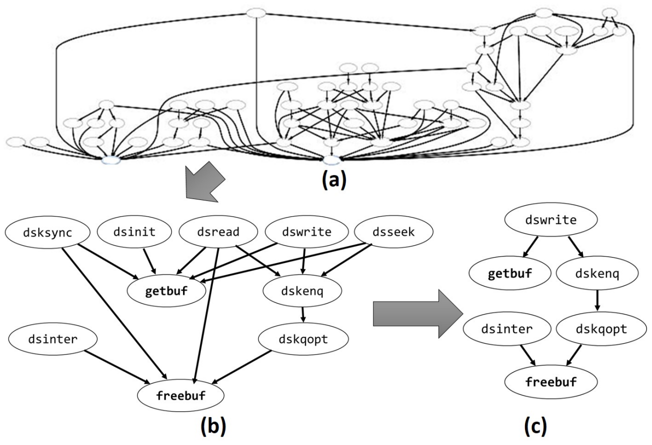


Figure 5: A progression of models leading to MPG

While conceptual queries serve the purpose of communicating ideas, it is far easier for students to learn if they can execute the queries, observe the results, and learn through experiments. This is the topic of Section 5 where we discuss Atlas [6], the graph database platform. The above queries can be executed using Atlas.

3.5 Lessons from the Labs

The labs bring out two important barriers to understand why complete automation is not a practical solution to solve complex software problems. First, it is a control flow challenge because there may not be a control flow path to reach relevant functions. In the lab example, the relevant function `dsinter` is interrupt-driven and it cannot be reached from `dswrite` using control flow analysis. Second, it is a data flow challenge because an automated pointer analysis hits a barrier when the pointer to the allocated memory is inserted in a linked list.

While the students understand the fundamental data and control flow barriers to achieve complete automation, they also see how automatically generated visual models empower analytical reasoning to achieve sound and complete verification. The analytical reasoning and refining ideas through experiments are valuable for forming a lifelong learning habit. Specifically, the students learn:

- The *call graph* is neither sufficient nor necessary as a set of relevant functions for the verification.
- The *reverse call graph*, with matching pair events (e.g., lock and unlock calls) as leaves, is sufficient as a set of relevant functions for the verification. However, it would include many irrelevant functions and thus it is not efficient.
- Inter-procedural analysis must account for the following mechanisms for communicating the pointer to the allocated memory from one function to another: the pointer is (i) passed as a parameter, (ii) passed as a return, and/or (iii) assigned to a global.
- The type of the structure for which the memory is allocated can be used as a good approximate constraint to limit the set of functions for verification. For example, a candidate for MPG could be the reverse call graph but restricted to the set of functions that read and/or write to the structure type for which the memory is allocated.
- The notion of event traces relevant to the verification and the EFG as a compact form of CFG to capture those event traces efficiently.

4. LINUX CASE STUDY RESULTS

The results in this section provide quantitative and qualitative assessments to evaluate how EECV has worked in practice. The case study is based on EECV projects in a senior undergraduate course and a graduate course. We present examples of verification instances from the three result categories ($\mathcal{C}1$, $\mathcal{C}2$ and $\mathcal{C}3$ in Section 2.3.1) to give a qualitative sense of how the automatically generated evidence simplifies the verification task

4.1 Quantitative Assessment

We have applied $\mathcal{L}$-SAP [14] to produce IVKs for three recent versions (3.17-rc1, 3.18-rc1, and 3.19-rc1) of the Linux kernel totalling $>$ 37 MLOC. $\mathcal{L}$-SAP is fast and scalable, and it is able to accurately pair 66,151 (99.3% of the total) locks in 3 hours. To date no errors have been discovered through manual cross-checks of these results. The verification instances and the IVKs are available at the website [2].

4.1.1 Quantitative Assessment of Visual Models

The visual models are meant to produce evidence to reduce the burden on the human analyst. Optimizing the visual models is crucial for reducing the human effort. For example, without a good optimization, the MPG could have too many irrelevant functions. The larger the MPG the more the burden on the human analyst.

There is one MPG per lock. The average MPG size (the number of nodes in the MPG without the lock and unlock function nodes) over all the 66,609 locks is 1.3 with the maximum MPG size of 40 in case of one lock. The MPG size is one (i.e., only intra-procedural verification is needed) for 62,663 locks. The manual cross-checks revealed 161 cases where the MPG points to missing evidence, i.e. some functions relevant for verification are not included in MPG. In most of these case, relevant functions are missing from the MPG because they are called using function pointers. These cases happen because our current implementation of MPG does not take into consideration calls through function pointers. The $\mathcal{L}$-SAP tool could be improved to process at least some cases of function pointers automatically. Later in Section 4.3.2, we present an example of how the students can augment the MPG and complete the verification by issuing queries to gather the missing functions called through function pointers.

There is one EFG per function. The larger the EFG the more the burden on the human analyst. The average EFG size (the number of nodes in the EFG without the unique entry/exit nodes) over the 55,215 functions is 4. The maximum EFG size is 63 for function `wsm_set_edca_params`.

The reductions from CFGs to EFGs are particularly important for complex CFGs, and especially for CFGs with a large number of branch nodes. Table 1 lists the reductions for the ten functions, from the 55,215 relevant functions in all MPGs, with the largest number of branch nodes in their CFGs. For example, for function `dst_ca_ioctl` the reductions from CFG to EFG are: from 349 to 2 nodes, from 518 edges to only one edge, and from 163 to no branch nodes.

Table 1: A comparison of CFG vs. EFG

Function Name	Nodes		Edges		Branch Nodes	
	CFG	EFG	CFG	EFG	CFG	EFG
`client_common_fill_super`	1,101	15	1,179	28	249	13
`kiblnd_create_conn`	731	18	925	34	197	15
`CopyBufferToControlPacket`	392	20	559	39	180	18
`kiblnd_cm_callback`	662	38	831	56	170	15
`kiblnd_passive_connect`	622	22	784	44	164	20
`dst_ca_ioctl`	349	2	518	1	163	0
`qib_make_ud_req`	621	10	821	15	156	5
`cfs_cpt_table_al`	522	7	672	13	153	6
`private_ioctl`	569	16	732	24	148	8
`vCommandTimer`	490	47	623	75	143	28

4.1.2 Manual Effort

The manual cross-check of automatically verified instances with no violation (category $\mathcal{C}1$) took on average 2 minutes per instance. For the remaining instances (categories $\mathcal{C}2$ and $\mathcal{C}3$), the manual verification using the visual models took on average 8 minutes per instance with the maximum time of 30 minutes for a few difficult instances.

4.2 Qualitative Assessment of Evidence

We present examples of verification instances to give a qualitative sense of the human effort and how the MPG and EFG are used.

- **Cross-checking:** In Section 4.2.1, we present an example to give a qualitative sense of the human effort to cross-check an automatically verified inter-procedural instance with no violation (category $\mathcal{C}1$, 99.3% instances). The example shows how the MPG and EFG can be used for cross-checking.

- **Signature Issue:** In Section 4.2.2, we show an example of miss-reported bug. In this example, the $\mathcal{L}$-SAP tool fails to analyze an instance correctly because the signature is the same for two consecutive locks without an unlock in-between. In reality, the two locks operate on different objects. After a manual examination of verification instances related to signature problem, we have found cases difficult to fix by improving automation. For example, an array of pointers to different mutex objects of the same type is a difficult case to solve by automation.

- **EFG Points to Missing Evidence:** In Section 4.2.3, we present an example of an EFG that points to missing evidence. In this example, the missing evidence is a lock that occurs before a loop. The loop has an unlock followed by another lock. On entering the loop, the unlock is unpaired unless the loop is preceded by another lock.

- **MPG Points to Missing Evidence:** This is the case where function `f` has an unlock without a lock preceding it. Thus, one expects the MPG of `f` to contain a parent function `g` which has a lock and a call edge from `g` to `f`. As discussed in Section 4.1.1, the current implementation of MPG does not include such a parent function. This leads one to hypothesize that the parent function `g` exists but the call is via a function pointer. In this case, the student can use the Atlas query language to search for `g` that calls `f` using a function pointer. After finding `g`, the student can complete the verification. This is an interesting case that leads to a bug discovery.

In all these cases, the evidence provides valuable clues to identify the issues and empowers analytical reasoning to either correct or complete the verification.

4.2.1 Example 1

The example shows how easy it is to cross-check an automatically verified instance. Figure 6 shows the visual models for a lock that $\mathcal{L}$-SAP has reported to be correctly matched. Figure 6(a) shows the MPG for the lock in the function `hso_free_serial_device`. Figures 6(b) and 6(c) show the EFGs for the MPG functions `hso_free_shared_int` and `hso_free_serial_device`, respectively.

In this example, it is easy to observe from the EFG of function `hso_free_serial_device` that the lock is followed by a branch node with two paths: (1) one path leads to a matching unlock (intra-procedural), and (2) the other path leads to a call to function `hso_free_shared_int` (blue-colored node). The EFG of the called function `hso_free_shared_int` (Figure 6(b)) shows a matching unlock on all paths within the called function. This evidence simplifies the inter-procedural cross-check to conclude that the automatic verification is correct.

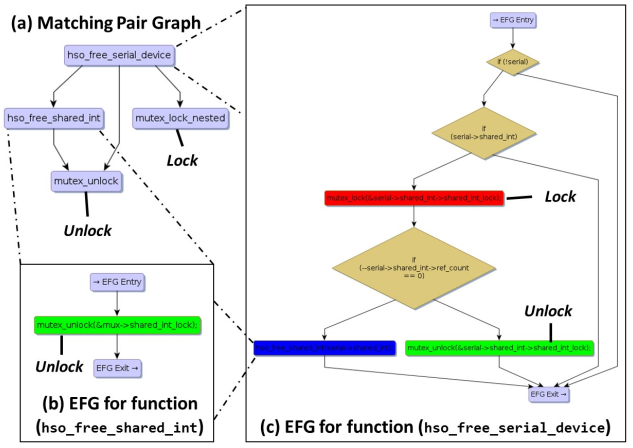

(a) Matching Pair Graph
hso_free_serial_device
hso_free_shared_int
mutex_lock_nested
Lock
mutex_unlock
Unlock
→ EFG Entry
if (!serial)
if (serial->shared_int)
mutex_lock(&serial->shared_int->shared_int_lock);
Lock
if (--serial->shared_int->ref_count == 0)
Unlock
hso_free_shared_int(serial->shared_int);
mutex_unlock(&serial->shared_int->shared_int_lock);
EFG Exit →
→ EFG Entry
mutex_unlock(&mux->shared_int_lock);
Unlock
EFG Exit →
(b) EFG for function (hso_free_shared_int)
(c) EFG for function (hso_free_serial_device)

Figure 6: Visual models for an automatically verified instance

4.2.2 *Example 2*

In this example, $\mathcal{L}$-SAP reports a deadlock because a lock is followed by another lock with the same signature. $\mathcal{L}$-SAP deems the two objects to be the same because they have identical signatures. In reality, the two objects are different but $\mathcal{L}$-SAP lacks a refined notion of signature to distinguish them.

Figure 7 shows the EFG for function `ucma_lock_files`. The EFG shows a lock immediately followed by another lock. It represents either an inadvertent coding error or a mistaken identity for two locks that are different but have the same signature. Here, the source correspondence is important to resolve the matter. By clicking on each lock call, one can find that the two locks operate on different objects. There are 82 instances with this issue.

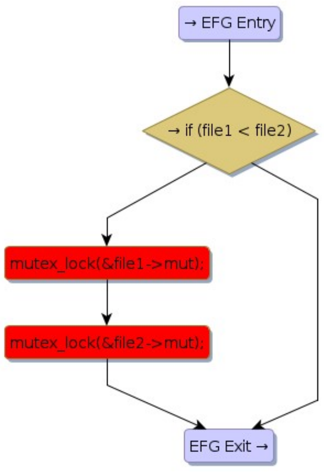


Figure 7: The EFG for function `ucma_lock_files` shows incorrect automatic verification

4.2.3 *Example 3*

This example shows how an EFG may point to an unusual situation and it is not difficult to verify through human intervention.

Figure 8(a) shows the MPG for the lock in the function `destroy_async` and Figure 8(b) shows the EFG for function `destroy_async`. The EFG shows that the lock is matched correctly with the two unlocks on two paths. However, there is a dangling unlock upon the entry to the loop. This points to the possibility of a lock before the loop. If such a lock is there then the verification is complete with no dangling unlock. The students need to realize that the lock before the loop would have its own verification instance. They need to look for that lock and its verification instance and verify it correctly. Actually, $\mathcal{L}$-SAP works correctly in this case but the students get confused by the dangling unlock they observe in the EFG.

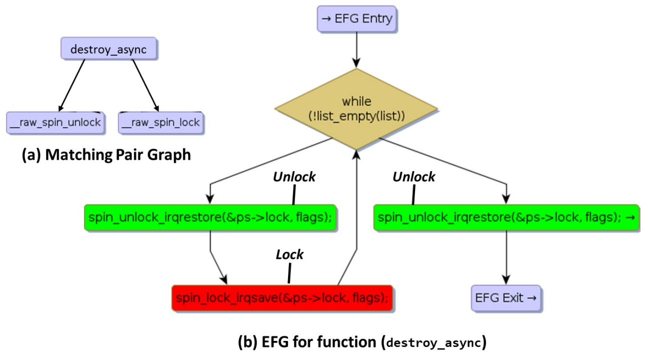

→ EFG Entry
destroy_async
__raw_spin_unlock
__raw_spin_lock
(a) Matching Pair Graph
while (!list_empty(list))
Unlock
Unlock
spin_unlock_irqrestore(&ps->lock, flags);
spin_unlock_irqrestore(&ps->lock, flags); →
Lock
spin_lock_irqsave(&ps->lock, flags);
EFG Exit →
(b) EFG for function (destroy_async)

Figure 8: EFG points to a missing lock preceding a loop

4.3 Examples of Bugs Discovered

4.3.1 *A missing unlock*

Figure 9 shows the visual models for a discovered bug. This bug was discovered automatically by $\mathcal{L}$-SAP and then cross-checked manually. Figure 9(a) shows the MPG for the lock in the function `toshsc_thread_irq`. Figure 9(b) shows the EFG for `toshsc_thread_irq`.

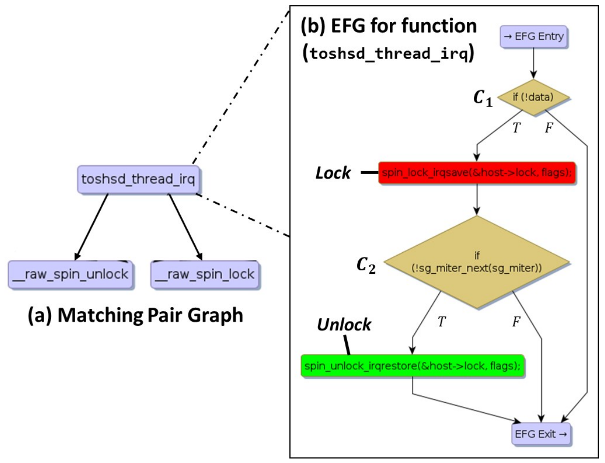

(b) EFG for function (toshsd_thread_irq)
→ EFG Entry
C_1
if (!data)
T
F
Lock
spin_lock_irqsave(&host->lock, flags);
C_2
if (!sg_miter_next(sg_miter))
Unlock
T
F
spin_unlock_irqrestore(&host->lock, flags);
EFG Exit →
toshsd_thread_irq
__raw_spin_unlock
__raw_spin_lock
(a) Matching Pair Graph

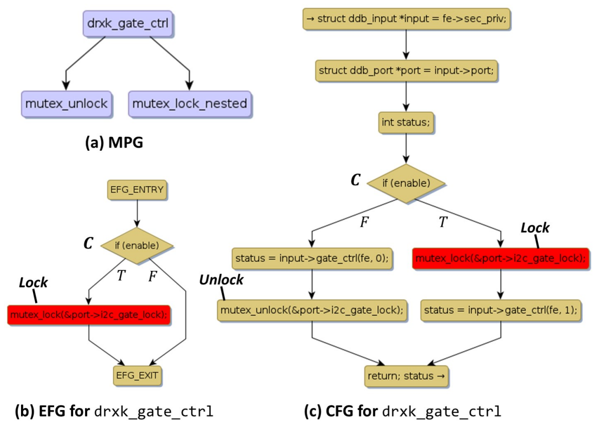
Figure 9: A bug discovery using visual models

The EFG for `toshsc_thread_irq` shows a path on which the lock in not followed by an unlock. As seen from the EFG, the path is feasible if the boolean expression $(C_1\overline{C_2})$ is true. To complete the verification, the student must verify that the boolean expression is satisfiable and concludes that the automatically reported violation is indeed a violation. This bug was reported to the Linux organization and it is fixed.

4.3.2 *An instance with a function pointer*

This example brings out interactive reasoning where the querying capability in Atlas is crucial. The MPG points to the possibility of functions relevant to the verification, but may be missing because they may have been called using function pointers. The querying capability is needed to find these functions.

Figure 10 shows the visual models for the lock in function `drxk_gate_crtl` reported as unpaired by $\mathcal{L}$-SAP. Figures 10(a), (b) and (c) show the MPG, EFG, and CFG. The MPG shows that function `drxk_gate_crtl` calls lock and unlock, however, the EFG shows that the lock is *not* matched by an unlock. The corresponding CFG shows why they are not matched. The lock and unlock are on disjoint paths: if $C =$ `true`, the lock occurs, otherwise, the unlock occurs.



drxk_gate_ctrl
mutex_unlock
mutex_lock_nested
(a) MPG
EFG_ENTRY
C
if (enable)
T
F
Lock
mutex_lock(&port->i2c_gate_lock);
EFG_EXIT
(b) EFG for drxk_gate_ctrl
→ struct ddb_input *input = fe->sec_priv;
struct ddb_port *port = input->port;
int status;
C
if (enable)
F
T
Lock
status = input->gate_ctrl(fe, 0);
mutex_lock(&port->i2c_gate_lock);
Unlock
mutex_unlock(&port->i2c_gate_lock);
status = input->gate_ctrl(fe, 1);
return; status →
(c) CFG for drxk_gate_ctrl

Figure 10: Visual models for `drxk_gate_crtl` pointing to presence of calls via function pointers

The lock and unlock on disjoint paths could match if `drxk_gate_crtl` is called twice, first with $C =$ `true` and then with $C =$ `false`. This amounts to using `drxk_gate_crtl` first as a lock and then as an unlock. A quick query shows that `drxk_gate_crtl` is not called directly anywhere. Thus, it is either dead code or `drxk_gate_crtl` is called using a function pointer. It is apparent that the evidence in this example is not sufficient, however, it gives the students valuable clues to start with.

As shown in Figure 10, function `tuner_attach_tda18271` calls `drxk_gate_crtl` via function pointer. `demo_attach_drxk` sets the function pointer to `drxk_gate_crtl`, the pointer is communicated by parameter passing to `dvb_input_attach`, then to `tuner_attach_tda18271`. Figure 11 shows the refined MPG after it is augmented with these functions newly discovered by human intervention.

This interactive reasoning for discovering the functions called via function pointers can be conducted visually using Atlas. The queries amount to asking the following questions:

1. What are functions that set function pointers to function `drxk_gate_crtl`?

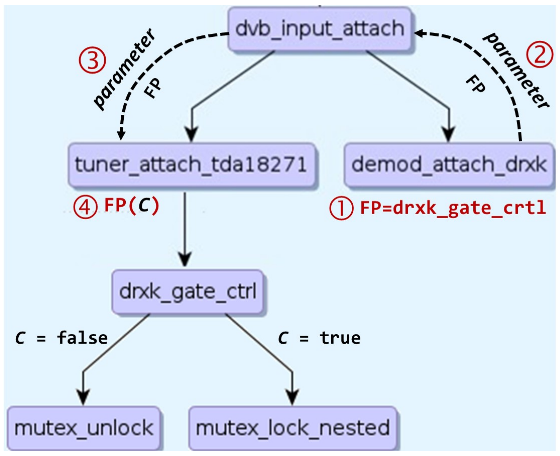


Figure 11: The augmented MPG for `drxk_gate_crtl` after resolving calls via function pointers

2. What are the functions that communicate the function pointer?
3. What are the functions that invoke calls via the function pointer?

We found a bug while working on this instance. Recall that `drxk_gate_crtl` must be called twice; first it acts like a lock and then as an unlock. There is a path on which there is a return before the second call and thus a bug because the second call for unlocking does not happen on that return path.

5. GRAPH DATABASE PLATFORM AND VISUAL MODELING

Visual models such as the MPG are crucial to empower human intelligence. These models reveal that the knowledge is hard to decipher from bare lines of code. Program mining and modeling tools are critically important for building such models. The models represent software in a way that enables powerful program analysis and reasoning. The same representation also enables visualization with an unprecedented level of interaction - which is important for students to understand non-trivial program analysis and reasoning concepts.

We have designed a graph database platform called Atlas [1, 6]. The database is used to store program artifacts. Atlas provides a query language to mine programs and build visual models. Currently, Atlas supports Java, Java byte code, and C. Atlas is available as an Eclipse plug-in. It has a graphical query language for constructing and analyzing visual models of software. The queries can be written in Java and packaged in Eclipse plug-ins as extensions to the platform, or executed using an interactive shell. Atlas builds a graph database of relationships between program artifacts. The user can issue queries against this database and the results are shown as visual models. The queries are composable. The result of a query can be stored in a variable and then passed as an input to another query. The Atlas platform has some built-in graph algorithms to transform and traverse graph models to serve as building blocks.

Atlas provides a rich API for writing queries to mine software and create visual models. The API provides capabilities for selecting, traversing, refining graphs to improve accuracy of analyses, and constructing subgraphs from the universe software graphs initially created by Atlas using a host of static analyses. Using the Atlas API, it is possible to write powerful analyzers in minutes or hours that would otherwise take weeks or months to write. For example, the function `cg` for constructing a call graph can be written with just two lines of code.

A function to construct call graph using Atlas APIs:

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

The eXtensible Common Software Graph (XCSG) schema includes several relationships (edges) which can be selected by other keywords in place of `XCSG.Call` as shown in `cg`. Through the ongoing research funded by the DARPA STAC program, we are advancing XCSG as a common schema to write language independent analyzers. So, functions such as `cg` can work for different programming languages.

Readers can get more information about XCSG and Atlas from the papers [6, 9] and the website [1].

6. RELATED WORK

The papers on teaching model-based software engineering [4, 5, 12] are typically either about formal methods or about UML modeling. Our primary goal is to teach problem-solving with large and complex software with visual models as a practical way to abstract a variety of software problems. The formal models or the UML models focus on teaching specific modeling methods and they require a significant amount of upfront time to teach modeling techniques. UML is a language independent notation for modeling software structure and behavior. XCSG, the graph schema for visual models, could be described that way, but there are significant differences. Broadly, XCSG is representing the details of the implementation, and follows the syntax of the language more closely, and merges concepts common across languages. If a student is familiar with Java syntax, the correspondence to XCSG becomes clear through interaction with the visualization, so explicit education about XCSG itself is less necessary.

Kramer in [10] has called abstraction the "*key skill*" in computing. Roberts in [13] emphasizes the use of models to remove unnecessary details and to generalize concepts and find patterns. As we have discussed earlier, this paper emphasizes on the use of visual models to provide abstractions by removing unnecessary details to amplify and empower human knowledge to cope with large software.

7. CONCLUSION

This paper proposes EECV as a vehicle to teach analytical reasoning in the context of software engineering. It summarizes a project to teach and conduct collaborative verification in a classroom setting. It describes labs to get students over the hump of abstraction and realize how modeling can be interesting and useful. Taught in this way, modeling can transcend software engineering and help students learn how to think clearly and effectively using mathematical abstractions. We have taught this material to undergraduate and graduate students at Iowa State University and through workshops to undergraduate students and teachers in India. We have received many comments through course and instructor evaluation forms indicating that the students found this topic thought provoking and useful. We also offer parts of this material through tutorials at conferences including: Automated Software Engineering (ASE), International Symposium on Software Reliability (ISSRE), International Conference on Information Systems Security (ICISS), and the Premier International Conference for Military Communications (MILCOM). The paper provides a website [2] to the IVKs for three recent Linux versions used in the class projects. The Atlas platform, the $\mathcal{L}$-SAP tool, the structured labs for the project, and the lecture slides are available upon request for academic use.

8. REFERENCES

[1] Ensoft corp. http://www.ensoftcorp.com.

[2] Linux results. http://kcsl.ece.iastate.edu/linux-results/.

[3] XINU. http://en.wikipedia.org/wiki/XNU.

[4] P. J. Clarke, Y. Wu, A. A. Allen, and T. M. King. Experiences of teaching model-driven engineering in a software design course. In *Online Proceedings of the 5th Educators' Symposium of the MODELS Conference*, pages 6–14, 2009.

[5] A. J. Cowling. Modelling: a neglected feature in the software engineering curriculum. In *Software Engineering Education and Training, 2003.(CSEE&T 2003). Proceedings. 16th Conference on*, pages 206–215. IEEE, 2003.

[6] 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 International Conference on Software Engineering*, pages 588–591. ACM, 2014.

[7] J. Gleick and R. C. Hilborn. Chaos, making a new science. *American Journal of Physics*, 56(11):1053–1054, 1988.

[8] K. Gui and S. Kothari. A 2-phase method for validation of matching pair property with case studies of operating systems. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 151–160. IEEE, 2010.

[9] B. Holland, T. Deering, S. Kothari, J. Mathews, and N. Ranade. Security toolbox for detecting novel and sophisticated android malware. *ICSE*, 2015.

[10] J. Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, 2007.

[11] E. N. Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141, 1963.

[12] L. Pareto. Teaching domain specific modeling. In *Symposium at MODELS 2007*, page 7, 2007.

[13] P. Roberts. Abstract thinking: a predictor of modelling ability? 2009.

[14] A. Tamrawi and S. Kothari. L-SAP: Scalable and Accurate Lock/Unlock Pairing Analysis for The Linux Kernel. *Submitted to ACM Transactions on Software Engineering and Methodology.*

[15] A. Tamrawi and S. Kothari. Event-flow graphs for efficient path-sensitive analyses. *arXiv preprint arXiv:1404.1279*, 2014.

[16] A. S. Tanenbaum and H. Bos. *Modern operating systems.* Prentice Hall Press, 2014.