A “Human-in-the-loop” Approach for Resolving Complex Software Anomalies

Suresh Kothari, Akshay Deepak, Ahmed Tamrawi, Benjamin Holland, Sandeep Krishnan
The Department of Electrical and Computer Engineering
Iowa State University, Ames, Iowa, USA
Email: {kothari, akshayd, atamrawi, bholland, sandeepk}@iastate.edu

Abstract—**Automated static analysis tools are widely used in identifying software anomalies, such as memory leak, unsafe thread synchronization and malicious behaviors in smartphone applications. Such anomaly-prone scenarios can be bifurcated into: “ordinary” (analysis requires relatively simple automation) and “complex” (analysis poses extraordinary automation challenges). While automated static analysis tools can resolve ordinary scenarios with high accuracy, automating the analysis of complex scenarios can be very challenging and, at times, infeasible. Even when feasible the cost for full automation can be exorbitant: either in implementing the automation or in sifting through the large number of erroneous results manually. Instead, we appeal for a “Human-in-the-loop” approach called “Amplified Reasoning Technique” (ART). While some of the existing approaches do involve human in the analysis process, the roles played by man and machine are mainly segregated. Whereas, ART puts man and machine in a “loop” in an interactive and visualization-based fashion. This paper makes an attempt to convince its readers to make their analysis of software anomalies ART-based by presenting real-world case studies of complex anomalies and how an ART based approach can be very effective in resolving them. The case studies highlight the desired characteristics of an ART based tool and the type of role it plays in amplifying human intelligence.**

I. Introduction

Keywords: software engineering, automated analysis, human-in-the-loop, assistive technology

Static analysis aims to analyze software without executing it. Automated static analysis tools are widely used in detecting software anomalies, such as non-compliance with coding and documentation guidelines, dead code and unused data, software vulnerabilities like unsafe thread synchronization or memory leak, and malware in smartphones.

Generally, an automated tool runs in three steps: (1) a human specifies the software to be analyzed and analysis-parameters, (2) the tools runs on the input and outputs a report of potential anomalies in the software, (3) an analyst goes through the report. A tool is considered sound if it reports all anomalies in the software with no false positives or false negatives. However, quite often it not possible to build a sound tool. Balancing coverage vs. accuracy in an analysis strategy involves an inherent trade-off: one can list only true-positives (low coverage, high accuracy) or one can output all potential anomalies (high coverage, low accuracy). Achieving high coverage and high accuracy in a fully automated tool can be impossible or incur prohibitive cost in terms of implementing the automation and/or sifting through the large number of erroneous results manually.

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

To motivate the case for an alternative to such an automated approach, we classify the set of anomaly-prone scenarios into two groups: “ordinary” and “complex”. Ordinary scenarios correspond to the scenarios that are amenable to automation and do not pose extraordinary analysis challenges. On the contrary, complex scenarios are the ones which pose significant barriers to automation; they involve hard-to-analyze programming paradigms and programming constructs. For example, a fully automated analysis may be intractable because of consumer-producer paradigm in which related events happen in different threads (e.g., a memory allocation in a producer thread and a corresponding memory deallocation in a consumer thread). Programming constructs such as function pointers also make analysis difficult by obscuring the control or data flow.

Even if automation for a complex scenario is possible, it may well be infeasible due to economics of time and effort. Malware analysis of applications in smartphones is a good example of this. What is considered malicious in one application may be considered benign in another because of difference in the purpose and the context of those applications. For example, accessing of contacts by an e-mail client is a legitimate action, but it is illegitimate, and probably malicious, for a weather application to do so. Further, malicious applications frequently blend their overt and malicious purposes. For example, an application meant for sharing pictures can leak to a malicious website without the knowledge of the user. Automating identification of malicious activities in such cases can mean writing a lot of application-specific code. Clearly, this will incur significant additional cost and seriously affect the viability and general applicability of the automated tool. On the other hand, leaving all the complex scenarios to be resolved by a human analyst alone after the automation run can also be prohibitive for reasons similar to why automation was approached in the first place. However, there can be some initial help from the automation run in such cases, but because the roles played by the automated tool and the human analysts are segregated, this help can be limited; leaving the human analyst to do most of the work.

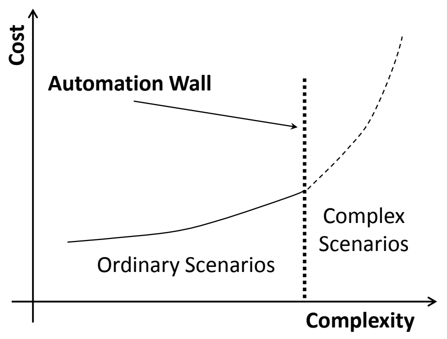


Figure 1: Cost escalates beyond automation wall

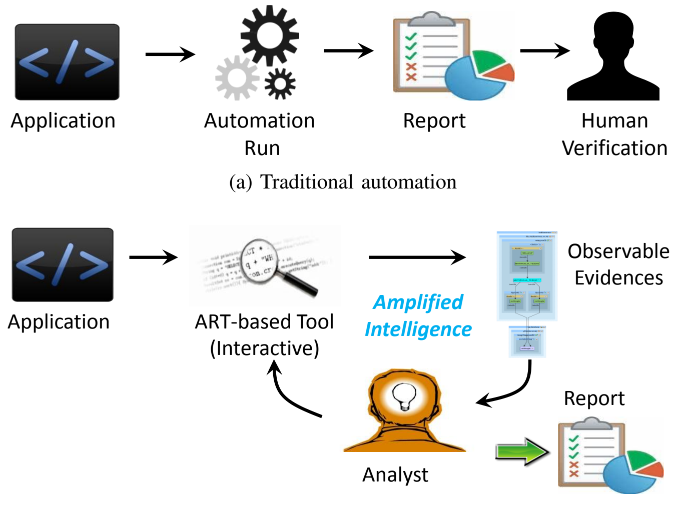


Figure 2: Traditional automation vs. ART-based approach

As summarized in Fig. 1, static analysis tools hit an "automation wall" and the cost for resolving complex scenarios escalates beyond the automation wall. We appeal for "Amplified Reasoning Technique" (ART) as an alternative approach to resolve such complex scenarios. The ART philosophy is: Instead of resolving complex anomalies as *definitive* "Yes/No" answers through a fully automated tool, bring the human in a man-machine *loop* and use the tool to *amplify* human reasoning to resolve such anomalies faster and efficiently. For example, to check the possibility of information leaks in an Android app, the human analyst would want to examine all the occurrences of Android APIs that are relevant in the context of a given app. Any reasoning of the type "examine all" is powerful but it is quite difficult to apply it to large software without an appropriate tool.

Figure 2 brings out the difference between the traditional approach to automation vs. the ART-based technique that has been discussed in this paper. In the traditional automation, the role of human is to sift through the false positives and unresolved cases generated from the automation run, and is segregated from the role played by the machine. Whereas, the ART-based approach puts the man and machine in an interactive loop. Each iteration involves human intelligence guiding the tool to generate refined evidences that bring closer to the final conclusion. Thus, an ART-based approach amplifies the human intelligence in resolving complex scenarios. The guiding principle is what Fred Brooks points out in his Allen Newell address "Computer Scientist as a Toolsmith" [1]:

> If indeed our objective is to build computer systems that solve very challenging problems, my thesis is that IA $>$ AI, that is, that *intelligence amplifying* systems can, at any given level of available systems technology, beat AI systems. That is, a machine *and* a mind can beat a mind-imitating machine working by itself.

We present three case studies of complex scenarios from real-word software to show how human reasoning amplified by a tool can be an effective and practical solution. These studies showcase different barriers to automation and their varying complexity, and the kind of role an ART-based tool plays. The purpose of this paper is to draw researchers' attention involved in resolving complex scenarios towards making their analysis ART-based by illustrating how it can be more cost-effective and productive. The case studies discussed in this paper do not compare the results of an ART-based vs. the typical automated techniques "statistically" to prove the clear superiority of one over the other, but they highlight the advantages an ART based analysis can offer by comparing the process of analysis. That is, the focus is on the process of analysis — rather than the results — and let the researcher decide if making his analysis ART based would be helpful.

II. Case Studies

In this section, we describe three real-world case studies for complex anomalies. The case studies are discussed in the increasing order of complexity of the scenarios. We show how each complex scenario can be resolved through interactions between a human analyst and an ART-based tool that plays an assistive role to the human analyst. Through these examples we illustrate the advantages an ART approach can offer. In each of the cases the ART-based tool being used has following features:

1) It identifies the unresolved cases or complex scenarios.
2) Provides a query-able interface to enable man-machine interaction to help human analyst to understand the complexity of the unresolved cases.
3) The human expert proposes and evaluates hypotheses to resolve complex scenarios. The tool enables the human expert to gather appropriate evidence to confirm or refute the hypotheses, and consequently give a verdict on that anomaly.

A. *Case Study: Safe Synchronization in Linux code*

The first case study is about verifying the safe-synchronization through mutex locking and unlocking in Linux kernel v2.6.31 (C code). For mutex object m let $L(m)$ and $U(m)$ denote the locking and unlocking event of m respectively. Then, the safe synchronization property is verified as follows: For every path in the control flow graph that has a mutex lock event $L(m)$ for a mutex object m, $L(m)$ is followed by $U(m)$ on that path. We first describe the case study (Sect. II-A1), give fully automated (Sect. II-A2) and ART-based (Sect. II-A3) resolutions of the case study, and, finally, contrast the fully automated and ART-based approaches (Sect. II-A4).

1) Description: The case study is about a complex example from the Linux kernel that incurs an extra cost to a fully-automated tool in the form of modeling programming mechanisms such as function pointers. Function $A =$ `stat_seq_start()`[1] calls a `mutex_lock()` on the mutex object `stat_session.stat_mutex` but the lock is never released by a call to `mutex_unlock()` in the same function. Function $B =$ `stat_seq_stop()`[2] calls a `mutex_unlock()` on the same mutex object, but there is no locking preceding the unlocking in the same function. Investigating the case, we found that a static variable `trace_stat_seq_ops`[3] of type `seq_operation` stores a function pointer to functions A and B. Then, functions $C =$ `traverse()`[4] and $D =$ `seq_read()`[5], each call the functions A and B, in that order, using the function pointers stored in the static variable. We found that every call to A is followed by a call to B on every execution path in functions C and D. Thus, this flow is legitimate, i.e., this complex scenario is not an anomaly.

2) A Fully-Automated Tool Resolution: Automating resolution of such complex scenarios is possible, however, it can involve use of complex techniques such as transforming program constructs into Boolean constraints and then using a SAT solver to check program properties [2], manual construction of external environment [3], [4], representing program constructs as finite state machines [5], [6], and symbolic execution [3]. While the usefulness of such techniques in static analysis have been proven beyond any doubt, we appeal that their application for resolving certain complex scenarios—such as the one being currently discussed—may not be worth the resources spent in implementing these techniques. Further, programming constructs such as function pointers, pointer-arithmetic, linked lists, loops and recursion can adversely affect the accuracy and coverage of such automations [2], [3]. Whereas, such scenarios can be resolved very easily by a human mind. Next, we list the details of how a human analyst can approach the solution using an ART-based tool for this case study.

3) ART-based Tool Resolution: We use an ART-based tool T that checks for safe-synchronization through mutex locking and unlocking in C programs through static analysis. The analyst has reasonable expertise in C programming. Let x denote the mutex object: `stat_session.stat_mutex`. First, the human analyst queries all the functions in the Linux kernel that pass x as a parameter to either mutex_lock $L(x)$ or mutex_unlock $U(x)$. T responds with Fig 3.

The analyst observes the following in the response by T (see Fig. 3):

- Functions `reset_stat_session()` and `stat_seq_init()` are highlighted in green indicating that these functions have $L(x)$ followed by $U(x)$ on every execution path. Thus, these case are safe.
- Function $A =$ `sta_seq_start()` is highlighted in red indicating that it has an execution path on which $L(x)$ is not followed by $U(x)$. The same is also indicated by the single call edge from A to `mutex_lock()`.
- Function $B =$ `stat_seq_stop` only calls `mutex_unlock()`, i.e., it has an execution path on which $U(x)$ is not preceded by $L(x)$.

[1]http://lxr.linux.no/linux+v2.6.31/kernel/trace/tracestat.c#L199
[2]http://lxr.linux.no/linux+v2.6.31/kernel/trace/tracestat.c#L232
[3]http://lxr.linux.no/linux+v2.6.31/kernel/trace/tracestat.c#L249
[4]http://lxr.linux.no/linux+v2.6.31/fs/seqfile.c#L65
[5]http://lxr.linux.no/linux+v2.6.31/fs/seqfile.c#L132

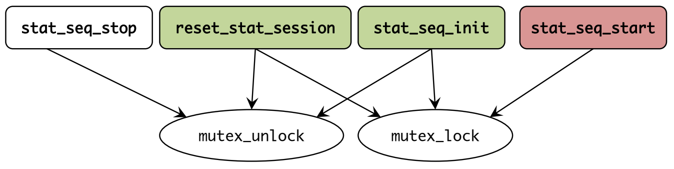


Figure 3: Case study 1: response by the ART-based tool.

At this stage, the human analyst hypothesizes of another function that can synchronize calls to $L(x)$ and $U(x)$ in functions A and B in a single flow of execution. So he queries the callers of A or B. However, the tool returns no callers — refuting his hypothesis. Based on this, the analyst hypothesizes of an invisible calling paradigm (i.e., threads, functions pointers, etc.), so he queries the tool for any artifacts in the Linux kernel that reads/writes a pointer to functions A and B. The tool returns the static variable `trace_stat_set_op`:

```
static const struct seq_operations
    trace_stat_seq_ops = {
    .start = stat_seq_start,/*assignment of
        pointer to function A*/
    .next = stat_seq_next,
    .stop = stat_seq_stop,/*assignment of
        pointer to function B*/
    .show = stat_seq_show };
```

Based on the previous result, the human analyst finds the hypothesis to be correct, i.e., there should be a function that calls start and stop through function pointers. Finally, the analyst queries all the functions that read/write the variable `trace_stat_seq_ops.start` and `trace_stat_seq_ops.stop`.
The tool returns the functions `traverse()` and `seq_read()` — both functions make a call to the function pointer `start` followed by a call to function pointer `stop` on every execution path. Thus, the analyst concludes that the mutex object `stat_session.stat_mutex` is safe.

In the above ART-based resolution, we can see:

- The conducted queries gave the human analyst a better view about the complex scenario
- Those pieces of information lead him to hypothesize about the problem, and verify or refute his assumption based on further narrowed and detailed queries.

Clearly, if an analyst has reasonable expertise in C programming, it would be very easy for him to see that this complex scenario is not an anomaly. There are 11 complex scenarios of this category in Linux kernel (out of total 403 complex scenarios in 1,255 instances of mutex locking).

4) ART-Based vs. Fully-Automated Approach: Based on the aforementioned case studies, the difference between the two approaches lies in implementing the respective techniques. For example, at a high level a typical fully-automated tool developed for the analysis of the above case study transforms program constructs into Boolean constraints and then uses a SAT solver to check program properties. There are also a number of other complexities that the authors in [2] address in modeling various constructs of C programming and in simulating program execution down to bit-level precision.

Whereas, the ART-based analysis uses the assisting tool to gather helpful information about the complex scenario that help the analyst to build hypothesis that will be verified or refuted based on further queries conducted by the analyst.

The queries supported by the ART-based tool are easy to implement on an off-the-shelf framework for manipulation of software program graphs of programs written in C (e.g., Atlas from EnSoft [7]). In summary, the ART-based analysis: resolves cost-effective anomaly-prone scenarios though automation, and reveals the unresolved complex scenarios to a human analyst. With such an assistive role, the human expert will have a hypothesis about the anomalous or safe behavior of the complex scenario. The tool enables the human expert to gather appropriate evidence to confirm or refute the hypothesis, and give a verdict on the analyzed anomaly.

It is easy to see that the amount of effort and time spent in implementing the ART-based tool for the above case study will be considerably less than the corresponding time and effort for a fully-automated tool. In other words, in terms of the theme of our discussion so far, the ART-based approach of analysis can be more cost-effective.

B. Case Study: Memory Leak Analysis in XINU Code

The second case study is about analyzing XINU[6] source code for potential memory leaks. The code is written in C. We use an ART-based tool T that checks for memory leaks in C programs through static analysis. The analyst has reasonable expertise in C programming and has a high-level understanding of the XINU architecture. Tool T is based on Atlas platform mentioned earlier, however, it is still under development: the figures in this case study are not the actual query-results returned by T, but represent what T is expected to return after its completion.

Memory is allocated through a call to function `getbuf()` and freed by a corresponding call to function `freebuf()`. Running T on XINU source code reveals that in function `dswrite()` memory is being allocated to pointer `drptr`, which is of type `struct dreq *`, but is not being freed on every execution path following the allocation. Figure 4 shows the code for `dswrite()`. Call to function `getbuf()` allocates memory to pointer `drptr` — highlighted in blue in Fig. 4. The analyst proceeds as follows:

1) The analyst asks T to show the execution paths where memory is not being freed. T responds with the call graph of function `dswrite()` restricted to the current memory leak scenario (see Fig. 5). The rectangular block corresponding to function `dswrite()` (see Fig. 5) shows the line in the source code where memory allocation happened by a call to function `getbuf()`. The green bubble in the right corner indicates the number of execution paths this memory allocation leads to where the corresponding memory deallocation could not be found by T. In this case there are seven such execution paths. All seven execution paths go through function `dskenq()` — the second rectangular block in the call graph. The rectangular block of function `dskenq()` has four execution blocks—each represented by a rectangular block colored in light coral—that correspond to execution paths where memory is not being freed. The first execution block calls function `dskstrt()`. The second execution block calls function `dskqopt()` and has four no-memory-deallocation execution paths going through it — indicated by the value in the corresponding green bubble. The remaining two execution blocks in function `dskenq()` do not call any function and each contributes a no-memory-deallocation execution path.
2) The analyst decides to investigate the call to function `dskqopt()` first because it contributes four or the seven

[6]XINU [8] is a Unix-like operating system originally developed by Douglas Comer for instructional purposes. See http://www.xinu.cs.purdue.edu/

```
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);
}
```

Figure 4: Function `dswrite()` : writes a block onto a disk device. The highlighted line corresponds to memory allocation.

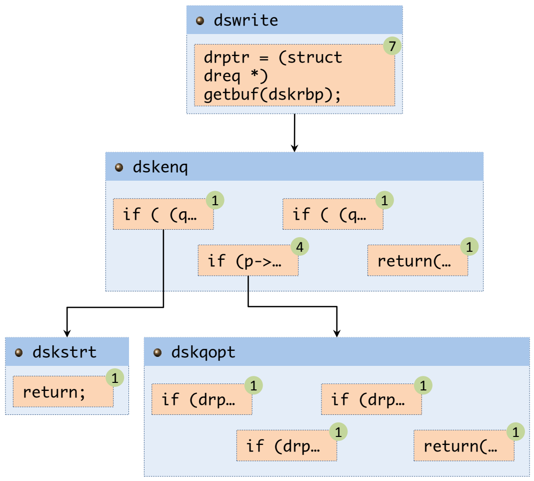
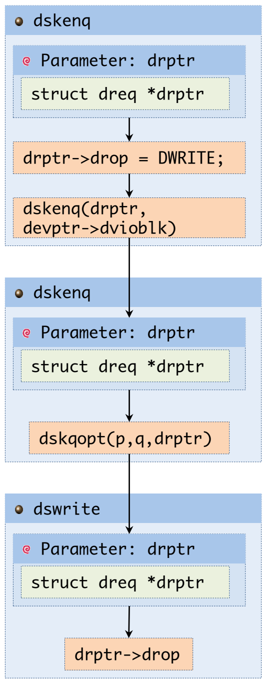


Figure 5: Call graph of `dswrite()` restricted to the memory leak analysis.

```
dskqopt(p, q, drptr)
        struct dreq *p, *q, *drptr; {
        char *to, *from;
        int i;
        DBADDR block;
        if (drptr->drop == DSYNC
                    || (drptr->drop == DREAD && p->drop == DREAD))
                return (SYSERR);
        if (drptr->drop == DSEEK) {
                freebuf(drptr);
                return (OK);
        }
        if (p->drop == DSEEK) {
                drptr->drnext = p->drnext;
                q->drnext = drptr;
                freebuf(p);
                return (OK);
        }
        if (p->drop == DWRITE && drptr->drop == DWRITE) {
                drptr->drnext = p->drnext;
                q->drnext = drptr;
                freebuf(p->drbuff);
                freebuf(p);
                return (OK);
        }
        if (drptr->drop == DREAD && p->drop == DWRITE) {
                to = drptr->drbuff;
                from = p->drbuff;
                for (i = 0; i < DBUFSIZ; i++)
                        *to++ = *from++;
                return (OK);
        }
        if (drptr->drop == DWRITE && p->drop == DREAD) {
                block = drptr->drdba;
                from = drptr->drbuff;
                for (; p != DRNULL && p->drdba == block;
                                p = p->drnext) {
                        q->drnext = p->drnext;
                        to = p->drbuff;
                        for (i = 0; i < DBUFSIZ; i++)
                                *to++ = *from++;
                        p->drstat = OK;
                        ready(p->drpid, RESCHNO);
                }
                drptr->drnext = p;
                q->drnext = drptr;
                resched();
                return (OK);
        }
        return (SYSERR);
}
```

Figure 6: Function $\texttt{dskqopt}()$: optimizes disk requests. The highlighted execution blocks correspond to potential memory leaks as found by tool T.



(a) Reverse taint graph of $\texttt{drptr}-> \texttt{drop}$.

```
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);
}
```

(b) Analysis reveals $\texttt{drptr}-> \texttt{drop}$ is being assigned $\texttt{DWRITE}$.

Figure 7: Reverse taint analysis of $\texttt{drptr}-> \texttt{drop}$.

execution paths that do not free the allocated memory. He asks T to show the corresponding execution blocks in $\texttt{dskqopt}()$. T responds with code for function $\texttt{dskqopt}()$ (see Fig. 6) and highlights the execution blocks where $\texttt{freebuf}()$ is not being called — indicating potential memory leaks.

3) The analyst observes that the governing condition for each of the highlighted execution block is based on the value of $\texttt{drptr}-> \texttt{drop}$. He asks T to do a reverse taint analysis on $\texttt{drptr}-> \texttt{drop}$ restricted to the current memory leak scenario. T responds with the reverse taint graph (see Fig.7a) and the code in $\texttt{dswrite}()$ where $\texttt{drptr}-> \texttt{drop}$ is assigned value DWRITE (see Fig.7b — the assignment is highlighted in blue).
4) The analyst observes that the assignment of value DWRITE to $\texttt{drptr}-> \texttt{drop}$ corresponds to the governing condition of the the last highlighted execution block in Fig. 6. Thus, he concludes that this execution blocks corresponds to the potential memory leak. The execution block is shown separately in Fig. 8.
5) The analyst, however, finds the potential memory to be too obvious. Further, going through the execution block he observes that the pointer drptr is being added to a linked list. Thus, it makes sense that it is being freed by another disjoint thread of execution. Assuming that such a thread exists, the analyst reasons that the two disjoint threads of execution must communicate through a shared data structure. Analysis reveals that the linked list to which the pointer drptr is being added is such a shared data structure. The analyst further reasons that the disjoint thread that frees memory must call $\texttt{freebuf}()$ passing a pointer of type $\texttt{struct dreq}*$ as argument. Recall that drptr—the pointer that is being allocated memory in $\texttt{dswrite}()$—is also of type $\texttt{struct dreq}*$. The analyst asks T to show all invocations of $\texttt{freebuf}()$

```
if (drptr->drop == DWRITE && p->drop == DREAD) {
        block = drptr->drdba;
        from = drptr->drbuff;
        for (; p != DRNULL && p->drdba == block;
                        p = p->drnext) {
                q->drnext = p->drnext;
                to = p->drbuff;
                for (i = 0; i < DBUFSIZ; i++)
                        *to++ = *from++;
                p->drstat = OK;
                ready(p->drpid, RESCHNO);
        }
        drptr->drnext = p;
        q->drnext = drptr;
        resched();
        return (OK);
}
```

Figure 8: Execution-block in `dskqopt()` corresponding to memory allocation in `dswrite()`.

in the XINU code where the parameter that an invocation receives is a pointer of type `struct dreq` $*$. T shows seven such invocations. The analyst observes that four of these are in `dskqopt()` and one each in `dsread()`, `dsksync()` and `dsinter()`.
Function `dskqopt()` already appears in the call graph of function `dswrite()` (see Fig. 5), which the analyst has come across earlier. Thus, he reasons it to be an unlikely candidate for the disjoint thread that frees memory. Further, T shows that the calls to `freebuf()` in `dsread()` and `dsksync()` are already matched with corresponding calls to `getbuf()`. Thus, he rules them out as unlikely candidates for the disjoint thread as well. T also shows that `dsinter()` is a root function, i.e., not called by any other function, and the call to `freebuf()` in it is not matched by a corresponding call to `getbuf()`. Further, the analyst observes that it is an interrupt driven routine, i.e., a separate thread for execution. These facts make `dsinter()` the likely candidate for the disjoint thread that frees memory. He asks T to show the invocation of `freebuf()` in `dsinter()`. The response is shown in Fig. 9. The invocation is highlighted in blue.

6) The analyst observes that the governing condition for the execution block in `dsinter()` containing the invocation of `freebuf()` depends on the value of `drptr->drop` (see Fig. 9). He notices that it matches the assignment made in `dswrite()` during the allocation of memory (see Fig.7b). Further, the given descriptions of these functions are:

`dswrite()`	Write a block (system buffer) onto a disk device.
`dsinter()`	Process disk interrupt.

Based on the above evidence, he concludes the following:
`dswrite()` allocates memory for `drptr`. Since this is a write operation to the disk, `dswrite()` adds `drptr` to a linked list and schedules a write to the disk. When the disk controller interrupts the CPU, marking the successful end of write operation, the interrupt handler `dsinter()` frees the memory allocated for `drptr`. Thus,

```
INTPROC dsinter(dsptr)
        struct dsblk *dsptr; {
        struct dtc *dtptr;
        struct dreq *drptr;

        dtptr = dsptr->dcsr;
        drptr = dsptr->dreqlst;
        if (drptr == DRNULL) {
                panic("Disk interrupt when disk not busy");
                return;
        }
        if (dtptr->dt_csr & DTERROR)
                drptr->drstat = SYSERR;
        else
                drptr->drstat = OK;
        if ((dsptr->dreqlst = drptr->drnext) != DRNULL)
                dskstrt(dsptr);
        switch (drptr->drop) {

        case DREAD:
        case DSYNC:
                ready(drptr->drpid, RESCHYES);
                return;

        case DWRITE:
                freebuf(drptr->drbuff);
                /* fall through */
        case DSEEK:
                freebuf(drptr);
        }
}
```

Figure 9: The invocation of `freebuf()` in function `dsinter()` with a pointer to type `struct dreq` $*$ as argument — highlighted in blue.

this complex scenario does not correspond to a memory leak.
The remaining execution paths are analyzed similarly and none are found to be a memory leak.

Observations: We contrast T with a fully automated tool F that also checks for memory leaks in C programs through static analysis. We highlight some of the advantages that T can have over F.

- In steps 1 and 2 F may only inform that not all paths after memory allocation in `dswrite()` have a corresponding deallocation, and may not keep track of the individual execution paths. Even if it stores the individual execution paths, it may not care to show them to the analyst because it is not meant to interact with the analyst at that level. Contrast this with the call graph of `dswrite()` (see Fig. 5) and the highlighting of individual memory-leak-prone execution blocks in `dskopt` (see Fig. 6) as displayed by T.
- In step 3 it was easy for the analyst to analyze the governing condition for each of the highlighted execution block and decide to do a reverse taint analysis. Again, in step 4 using the reverse taint analysis it was easy for the analyst to conclude that the assignment to `drptr->drop` in `dswrite()` corresponds to the governing condition of the the last highlighted execution block in Fig. 6. Automating either of these decision makings in F is a challenging task. Further, F may not be designed to do and display a reverse taint analysis like the one done in Fig. 7.
- Step 5 and step 6 demonstrate the advantage of human

insight. Knowing the analyst's familiarity with C programming, it is easy to see that the whole reasoning in these steps is natural to him. Automating such a reasoning in a tool like F will require incorporating problem-specific knowledge (in this case some understanding of the use-cases in an operating system like XINU) and modeling complex constructs of programming, such as linked lists, interrupts and multiples threads of execution. Thus, it would be a very challenging task. Further, in step 5, the analyst asks T to show all invocations of `freebuf()` in the XINU code where the parameter that an invocation receives is a pointer of type `struct dreq *`. Not only F may not support such a query but processing it can be quite different than the usual things that F does to identify memory leaks.

C. Case Study: Malware Analysis of an Android Application

Our research group is participating in Automated Program Analysis for Cybersecurity (APAC) program being conducted by DARPA[7]. The goal of APAC program is to develop tools to identify possible malicious code in Android apps so that such apps can be securely and confidently integrated in hardened smartphones. Unlike malware that follow known malicious paradigms and tend to be amenable to signature-based malware detection ([9], [10]), the DARPA APAC program is concerned with sophisticated malware including highly complex malware resulting from insider threats. As the third case study, we discuss an Android malware application (henceforth called app X) that we analyzed as part of our APAC research. Due to confidentiality agreements we are prohibited from releasing the app in public domain, thus, we discuss it at a conceptual level.

App X is an Internet Relay Chat (IRC)[8] client for Android. We use an ART-based tool called Security Toolbox (henceforth called T) that we have developed on top of Atlas framework [7]. Atlas converts source code into a graph database and provides APIs to query the database. The query results can be used for further analysis as well displayed instantly as graphs by the visualization interface. Our toolbox provides custom Atlas scripts that expose malware "hotspots" in an app and facilitate malware-specific exploration of the app using Atlas. App X source is in Java. In its initial run, T analyzes various uses of Java and Android APIs that are commonly abused for malware-purposes. T populates all such malware-prone APIs used in the app in a list. For example, the use of API `java.lang.reflect` in an app indicates possible use of Java Reflection to hide calls to malicious methods. Other examples are class loaders, native code, and application permissions.

1) While going through each API in the list, the analyst hypothesizes possible malicious behaviors that can arise due to use of the Android API. One such malware-prone API that T reveals is the use of Internet. The analyst hypothesizes that the app could be leaking IRC conversations to the Internet. The analyst asks T for all interactions of the app with Internet related APIs. T responds with Fig 10.

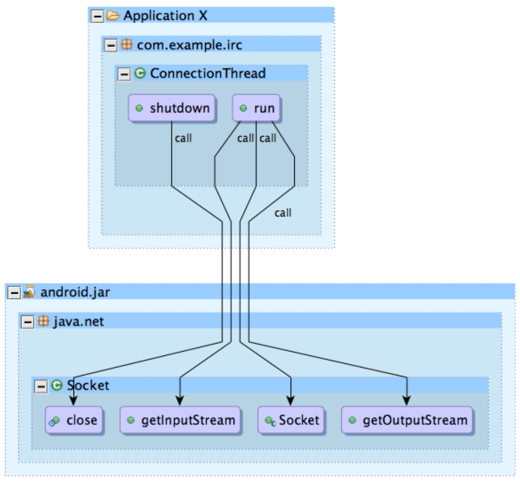


Figure 10: Internet related APIs.

2) The analysts observes that two methods `shutdown()` and `run()` call `java.net.Socket` APIs. The analyst notices that `shutdown()` only closes a network connection — not a candidate for malicious behavior. However, `run()` is a background thread that writes to an instance of `java.net.Socket` from a write-buffer (named `writer`). Suspecting sensitive data being leaked through `writer`, the analyst asks T to show all methods that write to `writer`[9].
3) The response by T contains only one method: `sendCommand()` in class `ConnectionThread`. This is an important point in the analysis because there is a transition from a well-known API in Android (internet APIs), to a domain-specific repackaging. Going through the source code of `sendCommand()`, the analyst observes that it is a private-helper function for sending data to the network connection. Thus, the analyst asks T to show the call graph of this helper function. T responds with Fig 11.
4) The analyst observes several implementations of IRC commands that leverage function `sendCommand()` to send data to the network connection. He observes that the majority of calls to IRC command implementations are made from public functions in class `ConnectionService` (see Fig. 11) and have a single caller. However, he observes the function `sendChat()` in class `ConnectionThread` has two callers; one of which is a private method in class `ProtocolParser`. Thus, the analyst finds the call by function `parseChat()` (in class `ProtocolParser`) to function `sendChat()` (in class `ConnectionService`) an "outlier" (relative to calls of other IRC command implementations in class

[7]http://www.darpa.mil/OurWork/I2O/Programs/AutomatedProgramAnalysisforCybersecurity(APAC).aspx

[8]Internet Relay Chat is a protocol for live interactive Internet text messaging (chat) or synchronous conferencing [11].

[9]The query corresponds to a forward taint analysis on `writer` restricted to write-only `java.io` APIs.

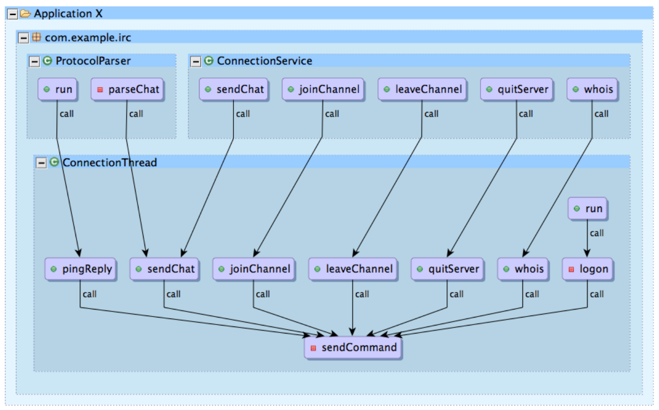


Figure 11: Call graph of sendCommand(). Only the first two levels from the original graph are shown here.

> ConnectionThread — see Fig. 11). Looking at the outlier call, the analyst finds it suspicious, because: "What is the need to send a chat (i.e., call function sendChat()) while parsing a chat (i.e., from function parseChat())?" The analyst investigates the source code of function parseChat() and finds that it sends all incoming chat messages to a hard-coded IRC user "0xFFFF". Following is the corresponding line of code:
> ```
> channel.sendChat("0xFFFF",chat);
> ```
> This is clearly a malware activity because leaking conversations this way is a breach of user-privacy. Thus, the analyst concludes the app to be a malware.

It is a very challenging task to automate the identification of the above malicious code because the malware activity blends very well with the legitimate functionality of the app. Capturing this subtle difference through automation will be a heavy task and can give a lot of false-positives. Further, the same malicious activity can be benign if all IRC conversations are supposed to be monitored by some hypothetical administrator-user "0xFFFF". Whereas, such subtle differences and contradictory scenarios come easy to human reasoning.

III. Discussion and Related Work

The complex anomaly scenarios presented previously are not sporadic, but frequently occurring real-world cases. For example, for checking safe-synchronization in Linux kernel out of 1255 scenarios, 403 are complex (32.1%). Specifically, out of the 403 complex scenarios: 68 have large execution paths ($>$500 nodes), 112 have infeasible paths (paths not feasible at runtime), 11 call functions via pointers, and 212 have unstructured code (e.g., jump statements like goto).

In general, a fully automated and completely accurate data-flow analysis is known to be NP-complete [12]. The number of execution paths grow exponentially with the number of non-nested branch nodes [13]. Apart from theoretical infeasibility, full automation may not be pragmatic because of programming constructs such as flow of data through containers such as linked lists, interrupt-driven routines, multi-threaded programs, function pointers, and the fact that a program may not be a closed system because of its dependence on external library or operating system calls. Given the infeasibility or impracticality of full automation, the ART approach makes sense because it is often relatively easier for a human analyst to reason: both in terms of the reasoning ability as well as a richer contextual understanding of the software. The analyst's difficulty is extracting relevant information from large software. That difficulty can be addressed with an apt tool with query interface. It is important that the analyst be not restricted to canned queries. Instead, the analyst should be able to compose new queries necessary to address varying contexts of complex anomaly scenarios.

Program comprehension is a central activity during software maintenance, evolution, and reuse [14]. Some reports estimate that up 60-70% of the maintenance effort is spent of understanding code [15]. It also plays an important role in an ART-based approach. The complex scenarios involve data and control relationships that require graph representations. Thus, a powerful ART tool needs the capability to represent and refine program semantics as graphs and provide a query language to search and manipulate the graphs.

References

[1] F. P. Brooks, Jr., "The computer scientist as toolsmith ii," *Commun. ACM*, vol. 39, no. 3, pp. 61–68, Mar. 1996.
[2] Y. Xie and A. Aiken, "Saturn: A scalable framework for error detection using boolean satisfiability," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 3, p. 16, 2007.
[3] T. Ball, B. Cook, V. Levin, and S. Rajamani, "Slam and static driver verifier: Technology transfer of formal methods inside microsoft," in *Integrated Formal Methods*, ser. Lecture Notes in Computer Science, E. Boiten, J. Derrick, and G. Smith, Eds. Springer Berlin Heidelberg, 2004, vol. 2999, pp. 1–20.
[4] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *Model Checking Software*, ser. Lecture Notes in Computer Science, T. Ball and S. Rajamani, Eds. Springer Berlin Heidelberg, 2003, vol. 2648, pp. 235–239.
[5] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A system and language for building system-specific, static analyses," *SIGPLAN Not.*, vol. 37, no. 5, pp. 69–82, May 2002.
[6] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, ser. OSDI'00. Berkeley, CA, USA: USENIX Association, 2000, pp. 1–16.
[7] "Atlas, ensoft corp." [Online]. Available: http://www.ensoftcorp.com/atlas/
[8] D. Comer, *Operating System Design: The XINU Approach Linksys Version*. CRC Press, 2011.
[9] R. Fedler, J. Schutte, and M. Kulicke, "On the effectiveness of malware protection on android," Fraunhofer AISEC, Tech. Rep., 2013.
[10] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 3–14.
[11] J. Oikarinen and D. Reed, "Internet relay chat protocol," 1993.
[12] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 323–337, Dec. 1992.
[13] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, 1970.
[14] S. Wei, "A survey and categorization of program comprehension techniques," Ph.D. dissertation, Concordia University, 2002.
[15] A. Von Mayrhauser and A. M. Vans, "Program understanding behavior during adaptation of large scale software," in *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*. IEEE, 1998, pp. 164–172.