Research Statement

Gogul Balakrishnan

My research interests broadly span programming languages and compilers. Recent research has shown that program analysis based on abstract interpretation, model checking, type checking, dynamic analysis, etc., can be effectively used to build tools for program understanding, program verification, bug finding, and so on. However, most of the research concentrates on program analysis at the level of source code, and the issue of analyzing executables is largely ignored. This is unfortunate because there is a growing need for tools that analyze executables. For instance, commercial companies and the military increasingly use Commercial Off-The-Shelf (COTS) components to reduce the cost of software development. They are interested in ensuring that COTS components do not perform anything malicious (or can be forced to perform anything malicious). Viruses and worms have become ubiquitous. A tool that aids in understanding the behavior of viruses and worms can ensure timely dissemination of signatures, and thereby control the extent of damage caused by them. In this document, I outline my dissertation research on the analysis of executables that attempts to fill this gap, and my agenda for future research.

Background

In the past few years, there has been a considerable amount of research activity to build tools that aid in program understanding, bug finding, and security-vulnerability detection. However, most of these techniques assume that the program’s source code is available to be analyzed, and therefore cannot be applied directly to programs for which source code is not available, such as COTS components, viruses and worms, third-party device drivers, third-party applications, activex plugins, etc. Moreover, in certain situations, such as finding security vulnerabilities, analyzing source code may miss some vulnerabilities because of the WYSINWYX [6] phenomenon: “What You See Is Not What You eXecute”. That is, there can be a mismatch between what the programmer intended and what is actually executed on a machine. The following fragment of code from a login program is an example of such a mismatch:

\[
\text{memset(password, `\0', len);} \\
\text{free(password);} \\
\]

Because the user password is sensitive data, the login program zeroes out the password buffer before freeing it. Although no vulnerability exists in the source code, an overzealous optimizer might remove the call to `\text{memset}', and thereby introduce a vulnerability in the compiled code. (This is not hypothetical; a similar vulnerability was discovered during the Windows security push in 2002 [9].) This is just the tip of the iceberg; there are many other advantages of analyzing executables:

- Programs typically use libraries, including Dynamically Linked Libraries (DLLs), for which one may not have the source code. A common practice that is used when analyzing such programs is to write stubs for the bodies of library functions. The process of writing stubs is usually error-prone and time-consuming. If the library executable can be analyzed directly, there is no need for such stubs.

- One can analyze programs that are written in more than one high-level language (such as a mixture of C and Java), and programs with inline assembly.

- The WYSINWYX phenomenon is not restricted to the presence or absence of procedure calls; on the contrary, it is pervasive: security vulnerabilities can exist because of a myriad of platform-specific details due to features (and idiosyncrasies) of the compiler and the optimizer. These can include (i) memory-layout details (such as offsets of variables in the run-time stack’s activation records (ARs) and padding between fields of a struct), (ii) register usage, (iii) execution order, (iv) optimizations,
and (v) artifacts of compiler bugs. Such information is hidden from tools that work on intermediate representations (IRs) that are built directly from the source code.

- Because of the WYSINWYX phenomenon, tools that analyze executables will often explore a smaller set of behaviors compared to tools that analyze source code, which may lead to improved precision. A source-code tool that wants to achieve the same fidelity as a tool that analyzes executables would have to duplicate all of the choices made by the compiler and the optimizer.

The overall goal of my research is to build a tool that can be used by an expert to find problems in an executable, either through manual inspection or through additional analysis built on top. As a means to this end, the main focus of my research is to develop algorithms and techniques to extract an IR from an executable that is similar to the one used by analysis tools that start from source code. Specifically, the IR should at least contain the following information: (1) control-flow graph (w/ indirect jumps resolved), (2) call graph (w/ indirect calls resolved), (3) a set of variable-like entities, (4) values of pointers, and, (5) used, killed, and possibly-killed variables for CFG nodes. From the perspective of the model-checking community, one would consider this problem to be that of model extraction. From the perspective of the compiler community, one would consider this problem to be that of IR recovery. The algorithms that I have developed have been integrated into a tool called CodeSurfer/x86 [1, 5].

Once such an IR is obtained, it is possible to leverage existing research on program analysis for bug finding, program understanding, program verification, etc. In my most recent work (as yet unpublished), I have begun to perform such analyses on device-driver executables.

Dissertation Research

Challenges One of the important steps in IR recovery is understanding the memory-access operations in an executable. Existing techniques are either extremely conservative [8] or unsound [7]. Neither approach is satisfactory: the former produces very approximate results; the latter produces information that cannot be relied upon.

One obstacle to creating a successful memory-access analyzer is identifying an appropriate set of entities to track memory operations. A program written in a high-level language accesses data in memory either directly through a variable or indirectly through a pointer. On the other hand, executables access memory either directly—by specifying an absolute address—or indirectly—through an address expression of the form “[base + index × scale + offset]”, where base and index are registers, and scale and offset are integer constants. (Square brackets denote dereferencing in Intel x86 assembly code.) While source-level variables can be used to track memory operations during source-code analysis, it is not clear from the expressions used in executables what entities (analogous to variables) are appropriate for analyzing an executable. If debugging information is available (and trusted), this provides one possibility; however, even if debugging information is available, analysis techniques have to account for bit-level, byte-level, word-level, and bulk-memory manipulations performed by programmers (or introduced by the compiler) that can sometimes violate variable boundaries [12]. Moreover, debugging information may not always be available, and even if it is present, it cannot not be relied upon in the case of malicious programs, such as viruses and worms. For these reasons, the algorithms that perform IR recovery should not assume the presence of debugging information (or at least not rely on it alone). Because executables do not have intrinsic entities (analogous to source-level variables) that can be used for analysis, a crucial step in the analysis of executables is to identify variable-like entities.

A second obstacle is determining the variables that can be accessed by an instruction. If an instruction uses direct addressing, the address is in the instruction itself; no analysis is required to determine the memory location (and hence the corresponding variable) referred to by the operand. On the other hand, if the instruction uses indirect addressing, the set of memory locations is computed from the registers used in the address expression. In such cases, to determine the memory locations (and hence the corresponding variables) referred to by the operand, the values or the addresses that the registers hold at this instruction need to be determined. It is tempting to use existing pointer-analysis algorithms for this task. However, they are not suitable for executables: (1) in an executable, an address and a numeric value can be used interchangeably, which existing pointer-analysis algorithms are not designed to handle, and (2) existing
pointer-analysis algorithms usually ignore pointer arithmetic, which is used extensively in an executable; for instance, even an access to a local variable involves pointer arithmetic: a local variable is usually accessed by adding an offset to the value of the frame pointer ebp and then dereferencing the resulting address.

To address these challenges, I have developed the following static-analysis algorithms:

**Value-Set Analysis (VSA)** Value-Set Analysis (VSA) addresses the problem of determining the variables accessed by an instruction. VSA [2] is a combined pointer-analysis and numeric-analysis algorithm that is suitable for executables. VSA has similarities with pointer-analysis algorithms because the results of value-set analysis can be used to answer queries such as “What are the addresses that a variable holds at a given program point?” On the other hand, VSA also has some of the flavor of numeric static analyses (like constant propagation, interval analysis, etc.) because the results of value-set analysis can be used to answer queries such as “What are the numeric values that a variable holds at a given program point?” Briefly, VSA is a flow-sensitive, context-sensitive, abstract-interpretation algorithm (parameterized by call-string length) that determines an over-approximation of the set of numeric values and addresses that each variable holds at each program point. The key features of VSA are that it tracks integer-valued and address-valued quantities simultaneously and takes into account pointer-arithmetic operations. These features make it suitable for analyzing executables.

The basis of the value-set analysis algorithm is the value-set domain [12]. A value-set represents a set of addresses and numeric values, and is used to represent the results of VSA. The value-set domain is designed in such a way that it can represent commonly occurring sets of addresses and numeric values without losing too much precision. For instance, unlike intervals, value-sets are capable of representing a set of addresses with a common minimum difference (stride), which may be used to represent the addresses of the elements of an array. The value-set domain is discussed in detail in [12].

**Recovering variables from executables** For the initial version of VSA [2], the executable’s variables were provided by IDAPro [10], a commercial disassembler. IDAPro’s variable-recovery algorithm is based on the observation that access to global variables appear as “[absolute-address]”, and access to local variables appear as “[ebp + offset]” or “[ebp – offset]” in the executable. Thus, absolute addresses and offsets that occur explicitly in the executable (generally) indicate the starting addresses of program variables. Based on this observation, IDAPro identifies each set of locations between two neighboring absolute addresses or offsets as a single variable. Such a local approach produces poor results in the presence of indirect memory operands. For instance, it does not take into account accesses to fields of structures, elements of arrays, and variables that are only accessed through pointers, because these accesses do not fall into any of the patterns that IDAPro considers. Therefore, it generally recovers only very coarse information about arrays and structures. Moreover, this approach fails to provide any information about the fields of heap-allocated objects, which is crucial for understanding programs that manipulate the heap.

To improve the state of the art, I developed a variable-recovery algorithm [4] that combines Value-Set Analysis (VSA), and Aggregate Structure Identification (ASI) [11] to recover variables that are better than those recovered by IDAPro. ASI is an algorithm that infers the substructure of aggregates used in a program based on how the program accesses them, and it requires information about the data-access patterns in the executable. Unlike high-level languages where data-access patterns are clear from the syntax of the language, they are not apparent from the assembly instructions in an executable. For instance, the address expression [eax+ebx] might represent an access to an array or a field of a structure, depending upon the values of eax and ebx. The main insight behind the algorithm in [4] is that the value-sets computed by VSA provide data-access patterns that we can use to identify the structure of memory, and hence the variables, using ASI. The combination of VSA and ASI allows CodeSurfer/x86 (a) to recover variables that are based on indirect accesses to memory, rather than just the explicit addresses and offsets that occur in the program, and (b) to identify structures, arrays, and nestings of structures and arrays. Moreover, when the variables that are recovered by our algorithm are used during VSA, the precision of VSA improves. This leads to an interesting abstraction-refinement scheme; improved precision during VSA causes an improvement in the quality of variables recovered by our algorithm, which, in turn, leads to improved precision in a subsequent round of VSA, and so on.

Initial experiments show that the technique is successful in correctly identifying 88% of the local variables and 89% of the fields of heap-allocated objects. Previous techniques [2, 10] recovered 83% of the local

\[ \text{Initial experiments show that the technique is successful in correctly identifying 88% of the local variables and 89% of the fields of heap-allocated objects. Previous techniques [2, 10] recovered 83% of the local} \]
variables, but 0% of the fields of heap-allocated objects. Moreover, the values computed by VSA using the variables recovered by our algorithm would allow any subsequent analysis to do a better job of interpreting instructions that use indirect addressing to access arrays and heap-allocated data objects: indirect operands can be resolved at up to 51% of the sites of writes and up to 14% of the sites of reads. (These are the memory-access operations for which it is the most difficult for an analyzer to obtain useful results.) This represents a substantial improvement over the best previous algorithm, which could only resolve up to 14% of sites of writes, and up to 8% of the sites of reads.

**Recency-Abstraction for Heap-Allocated Storage** A great deal of work has been done on algorithms for flow-insensitive points-to analysis (including algorithms that exhibit varying degrees of context-sensitivity), as well as on algorithms for flow-sensitive points-to analysis. However, most of these algorithms use a very simple abstraction of heap-allocated storage, called the *allocation-site abstraction*:

> All of the nodes allocated at a given allocation site \(s\) are folded together into a single summary node \(n_s\).

The initial version of VSA [2] also used the allocation-site abstraction for heap-allocated storage. In terms of precision, the allocation-site abstraction can often produce poor-quality information because it does not allow strong updates to be performed. A strong update overwrites the contents of an abstract object, and represents a definite change in value to all concrete objects that the abstract object represents. Strong updates cannot generally be performed on summary objects because a (concrete) update usually affects only one of the summarized concrete objects. If allocation site \(s\) is in a loop, or in a function that is called more than once, then \(s\) can allocate multiple nodes with different addresses. A points-to fact “\(p\) points to \(n_s\)” means that program variable \(p\) may point to one of the nodes that \(n_s\) represents. For an assignment of the form \(p->selector1 = q\), points-to-analysis algorithms are ordinarily forced to perform a weak update: that is, selector edges emanating from the nodes that \(p\) points to are accumulated; the abstract execution of an assignment to a field of a summary node cannot kill the effects of a previous assignment because, in general, only one of the nodes that \(n_s\) represents is updated on each concrete execution of the assignment statement. Because imprecisions snowball as additional weak updates are performed (e.g., for assignments of the form \(r->selector1 = p->selector2\)), the use of weak updates has adverse effects on what a points-to-analysis algorithm can determine about the properties of heap-allocated data structures. To mitigate the effects of weak updates, many pointer-analysis algorithms in the literature side-step the issue of soundness [3]. At the other extreme is a family of heap abstractions that have been introduced to discover information about the possible shapes of the heap-allocated data structures to which a program’s pointer variables can point. Those abstractions generally allow strong updates to be performed, and are capable of providing very precise characterizations of programs that manipulate linked data structures; however, the methods are also very costly in space and time.

Because weak updates to heap-allocated data affect the precision of VSA, I developed an abstraction for heap-allocated storage, referred to as the *recency-abstraction* [3], that is somewhere in the middle between the extremes of one summary node per malloc site and complex shape abstractions. In particular, the recency-abstraction enables strong updates to be performed in many cases, and at the same time, ensures that the results are sound. The recency-abstraction is similar in some respects to the allocation-site abstraction, in that each abstract node is associated with a particular allocation site; however, the recency-abstraction uses two abstract objects per allocation site \(s\): MRAB\([s]\) and NMRAB\([s]\).

- **MRAB\([s]\)** represents the most-recently-allocated block that was allocated at \(s\). Because there is at most one such block in any concrete configuration, MRAB\([s]\) is never a summary object. Consequently, an abstract transformer can perform a strong update on a field of MRAB\([s]\)

- **NMRAB\([s]\)** represents the non-most-recently-allocated blocks that were allocated at \(s\). Because there can be many such blocks in a given concrete configuration, NMRAB\([s]\) is generally a summary object.

As an application of the recency-abstraction, I used it to resolve virtual-function calls in stripped executables (i.e., executables from which debugging information has been removed). This approach succeeded in resolving 55% of virtual-function call-sites, whereas previous tools for analyzing executables fail to resolve any of the virtual-function call-sites.
Research Agenda

In the short term, I plan to explore the new possibilities that my tool for analyzing executables has opened up. In my most recent work, I have already started to use CodeSurfer/x86 for finding problems in device-driver executables. I am very excited about this line of research because, until now, several analysis problems that involved programs without source code were not amenable to principled static-analysis — only ad-hoc solutions were proposed. For instance, consider the problem of binary-compatibility checking. Ensuring binary compatibility is a major issue for the implementors of libraries and operating systems. There is usually a well-defined interface to access the functionality provided by a library or the OS. However, for a variety of reasons, such as to improve performance, to work around bugs, etc., an application might break the interface by accessing a feature in an undocumented way. When a new version of the library or OS is released such applications may fail to work correctly. Compatibility problems are usually discovered through testing, which may fail to find all problems. A tool like CodeSurfer/x86 makes such analysis problems amenable to static analysis. Binary-compatibility checking is just one instance of the new opportunities, and I am eager to pursue other opportunities too.

In the short term, I am also interested in improving the algorithm for IR recovery. There are several ways in which the current IR-recovery algorithm may be improved. One possible approach is to use dynamic techniques to complement or refine the information obtained from static analysis. Such a combination would be particularly useful for analyzing executables that have features like self-modifying code (such as polymorphic viruses and worms). Another possible approach is to tune the abstractions automatically to track only the information needed for the analysis problem at hand. Such abstraction techniques have been successfully used in source-code analysis tools, such as “property simulation” in the ESP tool (from Das et al.), and the lazy abstraction scheme in the BLAST tool. Using such parsimonious abstractions for IR recovery would improve precision as well as make it more scalable.

In the long term, I would like to explore new directions in program analysis and verification beyond executable analysis. I look at my current work on the analysis of executables as a springboard to my long-term efforts. Although, I have only worked on executables so far, the techniques and algorithms are general and can be adapted for source-code analysis. In fact, I believe that my experience in executable analysis would enable me to approach analysis and verification problems with a holistic view of the issues right from the source-code level to the executable level. The driving force behind programming languages and compiler research, right from the very first compiler to recent innovations in optimizations, program analysis, etc., is to improve the way programs are written and used. Just as my research in graduate school was directed by this need, my future efforts will be, too.

References


