The second half of the twentieth century saw the birth and the proliferation of computers from one per nation to several per household. Today, computers have become an inseparable part of our lives, and we routinely entrust them sensitive information, such as, for instance, credit card numbers, and even with handling of human well-being. Accordingly, the necessity to ensure that computers, and the software they execute, behave properly has become a very important problem. Simple programming errors may have grave consequences: an out-of-bounds array access may lead to your identity being stolen, a floating-point conversion error may lead to a spacecraft explosion (Ariane 5 crash, 1996), a division by zero may incapacitate an entire Navy ship (USS Yorktown, 1997). My research interests lie in the area of program analysis and understanding, and more specifically, in the area of numeric program analysis, which is dedicated to the detection of such errors.

Background.
There are many different approaches to discovering and eliminating software errors. They range from testing, in which the program's functionality is exercised by an extensive test suite, to program analysis, in which the program in never executed explicitly – instead, the source code of the program is analyzed to ensure the absence of errors.

To check whether a program has an error, an analysis must systematically explore all possible program executions. The term *program state* is often used to refer to a snapshot of program execution. If the set of program states that arise in all possible program executions (referred to as *reachable states*) does not contain states that constitute errors (referred to as *error states*), the analysis may conclude that the program is correct. However, computing the set of reachable program states is by no means trivial: numeric variables may take infinitely many values, dynamic memory allocation precludes the analysis from knowing *apriori* the set of memory locations that must be tracked, and recursion allows program functions to be invoked in an unbounded number of contexts. In fact, computing the set of reachable states is in general undecidable.

Program analysis sidesteps undecidability by approximating sets of reachable states by larger sets that are both decidable and effectively representable in a computer. These over-approximations are called *abstractions*. Since abstractions contain all possible reachable states, program analysis is *sound*. That is, if a program has an error, program analysis will identify it. However, abstractions may contain states that the program does not reach in reality. If these extra states satisfy error conditions, program analysis generates spurious error reports (also known as *false positives*). The central problem of program analysis is searching for abstractions that are both computationally efficient and precise (that is, produce a low number of false positives). Generally, no single abstraction can work well for all programs. Thus a large number of abstractions, each of which is tailored to a specific set of properties, has been designed.

Logically, a program analyzer consist of two components: an *abstract domain* approximates sets of reachable states and manipulates these approximations (referred to as *abstract states*) to reflect the effects of program statements; an *analysis engine* propagates abstract states through the program. The two components are connected through a generic interface. Mathematically, abstract domains are best thought of as partial orders, where order is given by set containment: the smaller the set, the more precise is the abstract state. Two properties of abstract domains are of particular interest: an abstract domain is said to be *distributive* if no information is lost when the abstract states computed along multiple program paths are combined; an abstract domain satisfies the *ascending chain condition* if it is impossible to construct an infinite sequence of abstract states that have strictly decreasing precision. If an abstract domain possess both properties, than any analysis engine is able to compute the optimal results with respect to that abstraction.

My research is centered on *numeric program analyses*: analyses that discover numeric properties of a program. A simple example of numeric analysis is one that discovers a range of values that a variable may have at a particular program location. A more involved example is an analysis that discovers numeric relationships that hold among values of program variables, e.g., establishing that the relationship $a = 4 * b + c$ always holds among the values of variables a, b, and c at a certain program point. Such numeric properties can be directly used to identify program errors, such as out-of-bounds array accesses, integer overflow, and division by zero. While seemingly simple, such errors account for the majority of known security vulnerabilities according to CERT.

The origins of numeric program analysis date back to the early 1970s. Over the years a rich set of numeric abstractions has been developed. These abstractions range from simple ones like *intervals*, which only keep track of upper and lower bounds for each variable, to relational ones, like *polyhedra*, which are able to establish linear relationships among variables, to automata-based *numeric decision diagrams*, which can represent arbitrary Presburger formulas. These abstractions exhibit varying precision/cost trade-offs and have been successfully used in practice. However, most of these numeric abstractions are not *distributive* and do not satisfy the *ascending chain condition*. This makes obtaining precise results somewhat of a black art.

Many program analyses, even those that are not directly concerned with numeric behavior of a program, often rely on numeric program-analysis techniques. To list a few examples:

- Low-level code analysis, such as analysis of x86 binaries, is quintessential for the fields of security and reverse engineering. In low-level code, typically, there are no explicit variables; rather, variables correspond to offsets from the beginning of an activation record or the beginning of the data section. Numeric operations are used to manipulate these offsets. Thus, a sophisticated numeric analysis is required just to determine which memory locations are accessed by each instruction.
- *Shape analysis*, an analysis that establishes properties of heap-allocated linked data structures, may use numeric quantities to represent some aspects of shape abstraction, such as the length of a linked-list segment, or the depth of a tree.
- *Memory-cleanness analysis*, an analysis that checks for memory-safety violations, uses numeric quantities to track the amount of memory allocated for each buffer, the offsets of pointers within the corresponding buffers, and the lengths of C-style zero-terminated strings.

Thesis Contributions.
In my thesis, I develop a suite of techniques with the common goal of improving the precision of numeric analysis. The techniques address various aspects of numeric analysis, such as handling dynamically-allocated memory, analyzing array operations, and regaining precision lost to extrapolation and non-distributivity. In my research, I did not attempt to create new abstractions; instead, I designed intelligent ways to use existing abstractions to obtain greater benefits. In a program analyzer, my techniques fit in-between the abstract domain and the analysis engine, and thus, can be viewed as *program analysis middleware*. I tried hard to make the techniques independent from specific abstractions and specific analysis engines. As a result, my techniques can be easily incorporated into existing program-analysis tools and should be readily compatible with new abstractions to be introduced in the future. Furthermore, even though they were inspired by numeric-analysis issues, the techniques can be used to improve the precision of non-numeric abstractions, as well.

Summarizing abstractions. Existing numeric abstractions are only able to keep track of a fixed, finite set of numeric variables. However, if a program manages memory dynamically, the set of variables that the analysis must keep track of may change as the program executes, and may not be statically bounded. For instance, keeping track of values stored in a linked list poses a problem to existing numeric analyses because it is impossible to model each individual list element. A typical approach that pointer analyses use to deal with dynamically allocated memory is to partition memory locations into a fixed, finite set of groups and reason about locations in each group collectively. Partitioning can be as simple as grouping together all memory locations created at a particular allocation site, as is done by many pointer-analysis algorithms; or as complex as maintaining a *fluid* partitioning that changes in the course of the analysis, as is done by state-of-the-art shape analyses. However, existing numeric abstractions cannot be used in this setting because they are incapable of such collective reasoning.

As part of my research, I designed a framework for automatically lifting *standard* numeric abstractions to support reasoning about potentially unbounded groups of numeric variables: instead of establishing numeric properties of individual variables, lifted abstractions capture *universal* properties of groups of variables [1]. For instance, a lifted polyhedral abstraction can capture the property that the value of each element in an array of unbounded size is equal to its index times two. Lifting is done by assigning a non-standard meaning to the existing abstraction. The sound and precise transformers for the lifted abstraction are automatically constructed from the transformers for the original abstraction. I used the above ideas to add numeric support to TVLA, a state-of-the-art shape-analysis framework. I also collaborated with Bertrand Jeannet (IRISA, France) on distilling these ideas into a novel relational abstraction for functions [6, 7].

Array analysis. Arrays are simple data structures that are heavily used in practice due to their simplicity and efficiency. However, arrays pose a major challenge to program analysis: array operations are rarely implemented for arrays of fixed size; instead, the size of an array is specified symbolically. Thus, the goal of the analysis is to establish that the operation behaves properly for arrays of any possible size. Furthermore, to successfully analyze array operations, the analysis must be able to capture relationships between the structure of an operation and the properties of array elements. For instance, to handle a simple array-initialization loop, the analysis must be able to maintain the property that array elements with indices that are less than the value of the induction variable have been initialized, while the rest of the elements have not.

I addressed the problem of array analysis by using a combination of *canonical abstraction*, an abstraction that dynamically partitions memory locations into groups based on their properties, with my summarizing numeric ab-

stractions [4, 5]. Numeric relationships among indices of array elements and values of variables that index into the array are used to partition array elements into groups. The numeric relationships among groups of array elements are established with the use of summarizing numeric abstractions. A prototype implementation of the analysis was able to successfully analyze a number of array-initialization and sorting routines.

Extrapolation. Many existing numeric abstractions require the use of *extrapolation* (also referred to as *widening*). The idea behind extrapolation is to guess a program invariant by observing the changes in program properties that are discovered early in the analysis. Extrapolation excels for programs with simple behavior. However, for programs whose behavior is harder to predict, extrapolation tends to lose a lot of precision. This precision loss makes the use of extrapolation very tricky in practice: the famous adage goes: "If you widen without principles, you converge with no precision!" A number of *adhoc* techniques for reclaiming lost precision have been proposed over the years. These mostly rely on invariant guesses supplied by either a programmer or an external analysis.

As part of my research, I designed a framework for *guiding* the state-space exploration performed by the analysis and used this framework to improve the precision of extrapolation [2]. The main idea is to restrict the analysis to parts of the program that have simpler behavior than the entire program, and thus, can be analyzed with more precision. After the analysis of a particular program part completes, the next program part is revealed to the analyzer. The framework is parametrized by a selection mechanism that identifies program parts to be analyzed. The selection mechanism that works best for extrapolation is one that applies analysis to each individual program phase in separation. I showed how to implement this particular instantiation of the framework efficiently in a way that easily integrates into existing analyzers.

Interprocedural analysis. Recently, Weighted Push Down Systems (WPDSs) emerged as an attractive engine for performing interprocedural program analysis. In WPDSs, weights capture the effect of program statements on program state. Algebraically, weights have strong similarities to abstractions. The existing techniques for model checking WPDSs guarantee precise results if weights are both distributive and satisfy the ascending chain condition. Most numeric abstractions, however, do not possess either of these properties. As part of my research, I experimented with using WPDSs for numeric program analysis. My primary concern was the precision loss due to non-distributivity of numeric abstractions.

Library summarization. Program analysis works best when it operates on the entire program. However, in practice, large parts of a program's functionality is hidden in libraries. Often, these libraries come from third-party software vendors, and their source code is not available. In practice, to perform the analysis, a collection of *stubs* that emulate the behavior of library functions is created. Usually, the stubs are created manually — a process that is both error prone and lengthy.

I applied the numeric analysis techniques that I developed to the problem of automatic derivation of library-function summaries [3]. Each function summary consists of a set of error triggers (that is, numeric conditions that, if satisfied at the function callsite, imply program failure during function invocation) and a transformer that indicates how to transform program state at the callsite of the function to the program state at the return. The technique applies directly to library implementations (x86 binary code, at the moment). CodeSurfer/x86, a tool for binary-code analysis, is used to perform initial analysis of the binary. The results are used to generate a numeric program, which is then fed into a WPDS-based numeric program analyzer to produce error triggers and library-function transformers.

Future Research Directions.

I anticipate low-level code analysis to be of particular importance in the near future. Today, a simple click of a mouse in a web-browser window may lead to software being downloaded, installed, and executed on a computer (without the user's explicit knowledge) to display, for instance, some media content. The same software may steal the user's private information from the hard drive, or use the computer as part of a botnet for launching distributed cyber-attacks. As a result, educated users are forced to forgo the first scenario (the convenience), to avoid the second scenario (the insecurity). An attractive goal for static program analysis is to automatically establish that a particular executable (e.g., the one downloaded from the net) is going to behave according to the first scenario, but not according to the second.

Low-level code poses a number of challenges to program analysis: there are no explicit variables, type information is not available, bitwise operations are often used to perform numeric operations (and vice versa), requiring the analysis to model program behavior on a bit level, etc. Furthermore, intruders (and attackers) are allowed to be as creative as possible, whereas program analysis is founded on the concept of modeling "the common case" precisely and losing precision on "creative" programs. An interesting research question is whether it is possible to construct scalable abstractions that are capable of modeling both numeric and bit-vector properties of numeric variables.

Another topic that captures my interest is *guided state-space exploration*: many useful abstractions (numeric abstractions, in particular) are not distributive and thus the precision of the analysis depends heavily on the order in which the program is presented to the analyzer. A number of *adhoc* techniques that derive benefit from guiding the state-space exploration exist: *lazy abstractions* explore the program in a way that avoids performing joins; *concolic* (*conc*rete+symb*olic*) techniques perform analysis along a particular execution path to generate test cases that exercise alternative program paths; some techniques for concurrent-program analysis use program under-approximations to reduce the number of interleavings that have to be considered. In my work, I used a sequence of specially-constructed syntactic program restrictions to improve the precision of widening. An interesting research direction is to attempt to generalize the above techniques into a single framework for guiding state-space exploration.

References

1. D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–529, 2004.
2. D. Gopan and T. Reps. Lookahead widening. In *Int. Conf. on Computer Aided Verification*, pages 452–466, 2006.
3. D. Gopan and T. Reps. Low-level library analysis and summarization. Submitted for review, 2007.
4. D. Gopan, T. Reps, and M. Sagiv. Numeric analysis of array operations. Tech. Rep. 1516, Comp. Sci. Dept., Univ. of Wisconsin, September 2004.
5. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Symp. on Princ. of Prog. Lang.*, pages 338–350, 2005.
6. B. Jeannet, D. Gopan, and T. Reps. A relational abstraction for functions. In *Int. Workshop on Numerical and Symbolic Abstract Domains.*, 2005.
7. B. Jeannet, D. Gopan, and T. Reps. A relational abstraction for functions. In *Static Analysis Symp.*, pages 186–202, 2005.