Towards the Industrial Scale Development of Custom Static Analyzers

John Anton, Eric Bush, Allen Goldberg, Klaus Havelund, Doug Smith, Arnaud Venet
Kestrel Technology LLC
4984 El Camino Real #230
Los Altos, CA 94022
{anton,ericbush,goldberg,havelund,smith,arnaud}@kestreltechnology.com
http://www.kestreltechnology.com

***Abstract*— This paper presents a high level overview of a technology called CodeHawk whose purpose is to support verification of software properties. Today's commercially available static analysis tools identify potential runtime and vulnerability problems based on properties described in the semantics of the programming language. While CodeHawk will detect those classes of problems, it is distinguished by the user's ability to generate high performance static analyzers for the verification of application-specific properties. Today's static analyzers may also trade off assurance and flexibility for speed in handling very large code sets. Our goal with CodeHawk is to handle industrial sized code sets with the highest speed in the industry among those offering 100% verification assurance. CodeHawk's customizability opens up additional uses of the core technology beyond detecting runtime or vulnerability exposures. In this paper we describe one such use, namely static analysis in support of optimized dynamic analysis.**

I. Introduction

In this paper we present our approach to static analysis of large software systems using a platform enabling the rapid development of custom static analyzers: CodeHawk. Unlike some static analysis approaches that are optimized to identify bugs, but not prove the absence of bugs, our objective is to achieve full code coverage so that there are no false negatives with respect to a set of well-defined properties. This is appropriate for high assurance systems, particularly those that must pass a rigorous certification process. In particular CodeHawk can prove properties of a C program's memory accesses that are sufficient for 100% assurance of the absence of buffer overflow errors. Insuring there are no false negatives together with an acceptably low rate of false positives raises a challenging scaling problem. Our approach to achieving scalability is to customize the analysis to the application domain, and to use algorithms engineered for high performance.

CodeHawk is a component of a larger system that combines static analysis with dynamic analysis. Dynamic analysis refers to monitoring the execution of a program for conformance with a set of properties. Static and dynamic analysis interact in two ways. First, static analysis can either establish that a property holds, establish that it does not, or fail to come to any conclusion. Dynamic checks may be inserted in the code to assist this process. Second, checking of dynamic properties may be optimized by static analysis. Within our framework dynamic properties are complex temporal properties expressed in a rich specification notation and the validity of such a property may depend on establishing related *sub-properties* at many different program points. Static analysis may verify these sub-properties.

The remainder of this paper is organized as follows. The next section overviews abstract interpretation, the theory on which CodeHawk is based. This is followed by an overview of CodeHawk. The next section describes how domain-specific properties are incorporated into CodeHawk through motivating examples. Then we discuss dynamic analysis and its integration with static analysis. The final section states conclusions.

II. Abstract Interpretation

Static analysis is a generic term encompassing a variety of techniques that vary greatly in scope and nature (type checking, coding style analysis, model checking, dataflow analysis, statistical pattern inference, pointer analysis, etc.). Abstract Interpretation [5], [6] is a theoretical framework enabling the systematic construction of *sound* static analyzers. By soundness, we mean that *all* possible execution paths are taken into account in the analysis. Hence, the properties of the program discovered by such an analyzer are guaranteed to hold in any configuration of the program. Formal verification of program properties can thus be achieved by Abstract Interpretation. A precise description of Abstract Interpretation is beyond the scope of this paper. We rather give the main intuitions underpinning the theory.

The behavior of a program is described by the set of its execution traces under all possible inputs. Execution traces can be formally described using a mathematical modeling technique called *operational semantics* [3]. Abstract Interpretation allows us to build a finite machine-computable model of the operational semantics of a program using two tools: *partitioning* and *abstraction*. Partitioning consists of grouping program configurations into a finite number of disjoint sets as illustrated in Fig. 1. For example, we can partition program configurations with respect to program control points, i.e., two configurations are in the same partition iff they reach the same

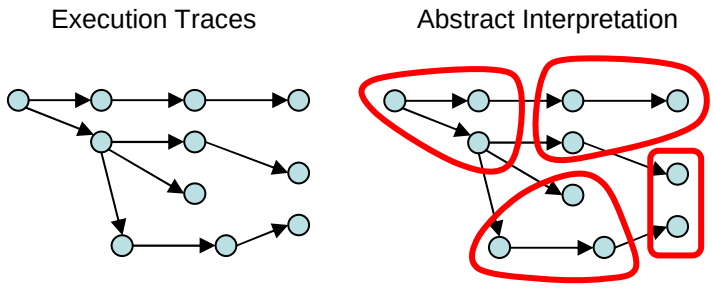

Fig. 1. Partitioning of program configurations

statement in the program. Abstraction consists of defining a single finite representation of all configurations in a partition. For example, if all the program variables are integer-valued, a possible abstraction consists of assigning an interval to each variable that contains all possible values of the variable in any configuration of the partition [4].

The abstraction process may cause the representation to denote program configurations that never occur in real executions. For example, if we have two configurations in a partition where `i = 2` and `i = 4`, variable `i` is represented by the interval `[2, 4]`, denoting the spurious configuration where `i = 3`. This explains the existence of *false positives* in program verification by Abstract Interpretation. A property may very well hold for all program executions, however the static analyzer cannot verify this is true, because it is violated for spurious configurations resulting from the abstraction employed. Note that we cannot have *false negatives*, i.e., a property cannot deemed true by the analyzer, even though it is violated in some executions. This is because *all* program configurations are covered by the abstraction.

Without entering in too much detail, we will just say that Abstract Interpretation provides a methodology and a collection of techniques that allow us to construct an *abstract semantic model* $\mathcal{M}$ of the program, that is a machine-representable structure representing the program dynamics on the abstract partitioning of configurations. The abstract semantic model is usually defined by induction on the syntax of the program and can be automatically generated by a proper translator. The envelope of $\mathcal{M}$, denoting the set of all possible configurations of the program, can be computed iteratively using well-studied fixpoint algorithms [2]. This structure can then be used to conduct automatic verification of the desired program properties.

We illustrate the abstract interpretation process on a small example. Consider the following piece of C code that initializes an array of double-precision floating point numbers:

```
1:  double a[10];
2:  int i;
3:
4:  for(i = 0; i < 10; i++) {
5:    a[i] = 1.0;
6:  }
7:  a[i] = 3.0;
```

Now, assume that we are interested in assessing the correctness of all array accesses. In the example, this translates into verifying the property $0 \leq \mathtt{i} < 10$ at lines `5` and `7`. The abstract semantic model defines the level of abstraction at which the analysis algorithms will operate. For example, one can choose to ignore all information stored in data structures. This makes sense for applications like embedded systems where the control structure is essentially driven by local variables, as described in [10]. This abstraction may be inappropriate for other families of programs. Once the abstract semantic model has been determined, abstract interpretation algorithms compute an envelope of all possible values for the program variables. If we choose an abstraction of numerical variables based on intervals, the analysis will automatically infer that the range of variable `i` is $[0, 9]$ at program point `5`, and $[10, 10]$ at program point `7`. Then, the computed ranges are used to check the safety properties for array access.

III. CodeHawk Technology

The Abstract Interpretation approach to static analysis looks attractive, but it presents major hurdles. Building a static analyzer based on Abstract Interpretation is a complex engineering task that can require substantial domain expertise. Designing the abstract semantic model $\mathcal{M}$ and writing the translator that takes the program text and produces $\mathcal{M}$ is the most time-consuming part of the process. Moreover, the abstract semantic model is specially designed to support the verification of a small number of program properties (usually one). Scaling to large code-bases has been proven possible by tailoring the abstract semantic model toward the particular structure of the software analyzed [10], [7]. All these factors lead to large, complex, monolithic static analyzers that are able to deal only with a handful of program properties. This approach is impractical for all but a few critical applications, and then only those blessed with a large V&V budget.

The purpose of the CodeHawk technology precisely consists of bringing the Abstract Interpretation approach to a practical production level. CodeHawk is built on top of the Specware [8] formal specification environment developed by Kestrel Institute. The main capability offered by CodeHawk is that of building a fully functional and efficient static analyzer by assembling components drawn from a library of predefined abstractions. The Specware environment is particularly supportive of that activity. In particular, the code of the whole analysis engine can be automatically generated from the specification of the analyzer. A static analyzer checking for a certain class of properties and tailored for a specific class of software can be rapidly specified and generated using CodeHawk.

CodeHawk's precursor, CGS [10] is a static array-bound checker tailored for NASA's flight mission software. It can scale up to half a million lines of C code, with a false positive rate $< 20\%$. CGS is written in C and has a monolithic architecture. Modifying the tool in order to have it analyze specific constructs of flight software more precisely is a complex and time consuming task. We found that the remaining 20% false positives were essentially due to array bounds transmitted between threads through message queues. Modifying the abstract semantic model in order to track this information precisely was not difficult in theory, but the impact on the implementation was enormous. This basically stopped us from further specializing the analyzer. CodeHawk aims at simplifying this specialization process by generating analyzers that have a flexible and tunable architecture.

Building an abstract semantic model from scratch is facilitated by CodeHawk, but still remains the job of an expert. We are currently working toward a specification environment built on top of CodeHawk that offers the capability of specifying custom program properties to verify and generate the corresponding analyzer. This specification environment will provide a high-level interface to CodeHawk that is accessible to the non-specialist and enable the construction of static analyzers for a broad spectrum of properties. The SAMATE database [9] will provide the basis for studying the specification language.

IV. Static Analysis from Numerical Specifications

In this section we illustrate the concept of a specification environment men- tioned above on two examples: a string copy function and a communication application using "nonces". These examples rely on the core capability currently implemented in CodeHawk: the analysis of numerical computations. They show analyzers for vulnerable use of the programmming language itself, resp. an application-specific property.

A. Buffer Overflow Violations

Consider the function `test` defined below:

```
void test(char *str){
  char buf[10];
  memccpy(buf, str, 0, 10);
  printf("results: %s\n", buf);
}
```

which is an extract of example 000-001-314 in [9]. The function takes as argument a string and prints it out, although in an unnecessarily complicated, and subsequently unsafe, manner, that embodies a potential for a buffer overflow. The function declares an array `buf` of size `10`. This array is then filled up with the text string. This is done by a call of the function:

```
void *memccpy(void *s1, const void *s2,
              int c, size_t n);
```

the description of which is:

> `memccpy` copies bytes from memory area s2 into s1, stopping after the first occurrence of c has been copied, or after n bytes have been copied, whichever comes first.

The problem occurs when `strlen(str)` (the length of the string) is bigger than or equal to the size of the array it is copied to (here 10), since in this case a final '0' is not copied into `buf`, and hence if `buf` is now used as a string the end of this string will not be clearly marked. We want to enforce the policy that the length of the copied string is strictly less than the size of the array. Then are we sure that a final '0' is copied in.

In order to detect such an error statically, a specialized algorithm can be programmed that performs a numerical abstraction of the program and analyzes this abstraction with respect to a desired property. In this specific case the abstraction keeps track of sizes of arrays and sizes of strings, and the specification states that any call of `memccpy` should copy a string with a smaller size than the size of the target array. The alternative to hard coding a static analyzer for this specific problem is to apply our generic approach and synthesize a static analyzer from an abstraction specification and a property specification stating a property to be checked over the abstraction.

The *property specification* now states the desired property, i.e., that calls of `memccpy` copy fitting strings:

```
check NoBufferOverflow =
  always(memccpy(arr, str, 0, N)
         -> size(str) < size(arr))
```

Of course this property can also be checked dynamically during program execution, and this might be a solution in case the property cannot be checked statically.

B. Nonce Repetition Violations

The above example illustrated the detection of runtime errors in the form of buffer overflows. The following example

illustrates a security problem concerned with uniqueness of authentification keys called *nonces*. Nonces are used for example in authentication protocols as a means of preventing replay attacks. A nonce is a "number used once". That is, the creator of the nonce should insure that it has not been used in previous runs of the protocol and that it is not guessable by an attacker.

Typically, randomly generated numbers are used as nonces. The SAMATE database [9] includes test cases (for example example test case 000-000-054) asserting that nonces should be used for the present occasion and only once. Here we consider how static and dynamic analysis can be used to assure correct uses on nonces. We assume a protocol is implemented by a collection of procedure calls, and that if a step in the protocol requires a nonce it is a parameter to a specific procedure `sendNonce`.

The abstraction specification would in this case define an abstract state that maps each nonce to an integer indicating how many times it has been used. It will also state how this abstract state is updated as a result of execution of program statements. The property specification will state that the integer associated to a nonce should never go beyond 1. Static analysis can also be used to check that the source of the value of the nonce parameter is a random variable library function.

However, if that cannot be statically validated, dynamic analysis can check that the nonce parameter is distinct at each invocation. This property can be expressed in our EAGLE monitoring language [1] as:

```
monitor NonceOnlyOnce =
  always(sendNonce(x) -> NonceNotSeen(x))

rule NonceNotSeen(int x) =
  previously(sendNonce(y) -> x != y)
```

V. Combining Static and Dynamic Analysis

Above it was mentioned that properties can be specified in a formal specification language and then checked statically. In case the static analysis cannot demonstrate the property, the whole property, or the part of the property that cannot be checked statically, can be checked dynamically during program execution using runtime monitoring techniques. A different way of thinking about this is to regard static analysis as a technique to optimize runtime monitoring: given a property to be monitored during execution, optimize the monitoring as much as possible in order to minimize the impact on execution time and memory consumption. In reality these are two views of the same problem, but from different perspectives.

These ideas can be brought even further by observing a current trend within Aspect Oriented Programming (AOP): the extension of pointcut languages with tracecuts (predicates on execution traces). In a traditional AOP language such as AspectJ an aspect contains advices of the form: "when this *piece of code is encountered*, execute this other piece of code". With *tracecuts* it is possible to state properties even more succinctly: "when this *temporal property is true about the execution trace*, execute this other piece of code". Such a framework can furthermore be supported by static analysis in the sense that static analysis statically attempts to determine when the tracecuts are true in the program and hence the new code can be inserted. In case it cannot be determined, monitors must be inserted in the code, which trigger the new code when reaching specific states.

The MODE system currently under development at Kestrel Technology combines static and dynamic analysis in such an AOP framework with tracecuts, in MODE called policies. MODE focuses on (1) a policy language based on state machines for expressing system safety and information assurance constraints, (2) static analysis mechanisms for detecting policy applicability in a program, and (3) enforcement mechanisms and associated assurance arguments and evidence. An overarching objective is to lower the cost of producing certified software.

MODE uses fast static analysis algorithms provided by CodeHawk to match each policy against the program. The engineer can specify whether to check a policy or enforce a policy. For each location in the program where static analysis determines that a policy applies, MODE either checks that it holds (generating a diagnostic message when it fails to hold) or automatically generates enforcement code for insertion at that location. MODE outputs a program that is consistent with the original program and that is guaranteed to satisfy the enforced policy.

VI. Conclusion

Static analysis for 100% verification of runtime, safety and security properties, is important. But to be practical, it must satisfy two requirements. First, it must scale to application code sizes used in industry. Second, it must support verification of properties that include those better defined in terms of the application's objectives, in addition to today's focus on those defined in terms of a programming language's usage. We have introduced a technology platform called CodeHawk that can meet those requirements.

References

[1] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of Fifth International VMCAI conference (VMCAI'04)*, volume 2937 of *LNCS*. Springer, January 2004.

[2] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *Lecture Notes in Computer Science*, 735, 1993.

[3] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, 1981.

[4] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 238–353, 1977.

[6] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, New York, NY, 1979.

[7] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *Proceedings of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, 2005.

[8] Kestrel. *Specware System and documentation*, 2003. http://www.specware.org/.

[9] NIST SAMATE Reference Dataset Project. Software Assurance Metrics and Tool Evaluation, NIST. `http://samate.nist.gov/SRD/srdFiles/index.php`.

[10] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the International Conference on Programming Language Design and Implementation*, pages 231–242, 2004.