CodeHawk: A High Precision Static Code Analyzer for Detecting Cybersecurity Vulnerabilities

Kestrel Technology LLC
3260 Hillview Avenue
Palo Alto, California 94304
www.kestreltechnology.com

Most software vulnerabilities are due to coding errors. Testing is usually the main means for detecting vulnerabilities, but testing alone only explores a small fraction of the possible behaviors of software. Testing can reveal the presence of errors and vulnerabilities, but never their absence. Fortunately, there is a technology, called *static analysis*, that can examine source code and reason about *all* of its behaviors in order to detect coding errors that lead to vulnerabilities.

Kestrel Technology (KT) is developing a breakthrough static analysis tool, called CodeHawk, with an initial focus on buffer overflow errors. Currently it exhibits near-perfect precision on standard benchmark codes produced by NIST and vulnerability experts. That is, when code contains a buffer overflow error, CodeHawk will almost certainly report the error and will report few "false positives" (reports of code locations that are not errors). Standard commercial bug-finding tools find a small fraction of the errors that CodeHawk finds on standard benchmarks. Moreover, CodeHawk provides mathematical evidence that an occurrence of a potential error either (1) is correct, or (2) is an actual error, or (3) may be incorrect on some inputs. This mathematical evidence can be provided to certification authorities and independently checked by them.

The pressures of cyber-crime and cyber-espionage will lead to increased demands from customers for credible evidence of freedom from exploitable vulnerabilities. CodeHawk is a front-runner in a new generation of automated tools that provide both cybersecurity analysis and independently checkable evidence for certifying authorities.

1 Introduction to Static Analysis

The term static analysis denotes the process of analyzing the text of a program, without actually running it, for the purpose of finding defects. Most static analysis tools are "bug finders", i.e. they discover potential defects of a program. A reported defect may be spurious (we say it is a *false positive*) while some actual defects in the program may not

be reported (we call these *false negatives*). This form of static analysis is usually qualified as *lightweight* or *unsound*, because the results cannot be trusted and must be manually verified. There are many companies that commercialize lightweight static analysis tools: Coverity, KlocWork, Fortify, Ounce Labs, etc.

It is possible to do static analysis in such a way that its results can be trusted. A theoretical framework called Abstract Interpretation [3, 4] describes the systematic design of static analyzers based on the mathematical definition of semantics of programming languages. A static analyzer based on Abstract Interpretation discovers properties of a program at each control point, also called invariants. A property of the program at any control point is true for all executions that reach that control point. The collection of all those invariants can then be used to perform the formal verification of safety properties (like the absence of buffer overflows) at each point of the program. If we denote by I_p the invariant discovered by static analysis at program point p and by S_p the safety property to be verified at the same program point, the verification process goes as follows:

1. If we can prove $I_p \implies S_p$ then the operation at control point p is safe for all executions.
2. If we can prove $I_p \implies \neg S_p$ then the operation at control point p violates the safety property for all executions.
3. If we can prove neither of those, we issue a warning that the operation can potentially violate the safety property.

The last case means either that there is an execution that violates the safety property at point p and another one that does not (what is called an *intermittent error*) or that the invariant I_p computed by the analyzer is not strong enough to prove the property (what is called a *false positive*). We call such static analyzers *heavyweight* or *sound*. We define the *precision* of a sound static analyzer as the percentage of all safety properties that can be fully determined using the invariants produced by the static analyzer (cases 1 and 2 above). In practice, the majority of warnings are false positives.

For example, consider the following piece of C code that translates elements of an array:

```
1: int *a[20];
2: j = 10;
3: for (i = 0; i < 10; i++) {
4:   a[i] = a[j++];
5: }
```

Imagine that we are interested in verifying the absence of buffer overflows in the program. A sound static analyzer using the domain of polyhedral invariants of [6] (i.e. that can

be represented by linear inequalities over the scalar variables of the program) would find that the following invariant holds at line 4 before the assignment is executed:

$$I_4 = \begin{cases} 0 \leq \mathtt{i} < 10 \\ \mathtt{j} - \mathtt{i} = 10 \end{cases}$$

Using this invariant, one can automatically check (using a Fourier-Motzkin procedure for example) that both array operations at line 4 stay within the array bounds. In this case the precision of the analyzer is 100% (two out of two safety properties can be determined using the invariants produced by the static analyzer).

2 The precision/scalability trade-off

The goal of a sound static analysis is to perform the bulk of a certain verification task automatically. In order to complete verification, all false positives must be verified manually or by other means. Therefore, the precision of a static analyzer must be as high as possible. However, a static analyzer is only as strong as the invariants it can infer and precision comes at a cost. The polyhedral domain [6] used to verify the absence of buffer overflows in the previous example works well for short codes with a small number of variables (typically under 15). Otherwise, performance rapidly degrades with the number of variables due to the combinatorial explosion of computing higher-dimensional convex hulls. Hence, it is unrealistic to apply such a static analyzer on programs larger than a few hundred lines.

The domain of intervals [2] allows the analyzer to infer bounds for each scalar variable in the program and scales much better than the domain of polyhedra (the domain of intervals can be applied to programs of a hundred thousand lines with good performance). However, the results are not as precise as those obtained with polyhedra. If we run an interval analysis on the previous example, the invariant obtained at line 4 would be:

$$I_4 = \begin{cases} 0 \leq \mathtt{i} < 10 \\ 10 \leq \mathtt{j} \end{cases}$$

Using this invariant, we can prove that the array access on the left-hand side of the assignment is safe, but not the one on the right-hand side of the assignment. Intervals do not capture the linear relation between variables `i` and `j`, hence we do not have an upper bound on the value of `j`. In this case the precision is only 50% (one out of two safety properties have been determined using the invariants discovered by the static analyzer).

This observation underlines the main difficulty of building a practical static analyzer: it is not possible to have an analyzer that both is uniformly precise and scales to large codes. The key to achieving precision while ensuring good performance is to specialize the

analysis process for a certain application or class of applications. Specialization usually requires the combination of multiple classes of invariants to drive precision up and the knowledge of the software architecture to enable/disable the use of certain domains on some parts of program. For example, a static analyzer that computes interval invariants and switches to polyhedra on loops that are smaller than a fixed threshold (e.g. 50 lines) implements a simple customization scheme that is sufficient to analyze the previous example with 100% precision. This analyzer will be both precise and efficient on codes that mostly contain small loops.

In practice, customization is not that simple and requires a fine-tuning of the algorithms based on the architecture of the application(s) to analyze. Real-life examples of specialized analyzers are ASTREE [5], a static analyzer tailored for a certain class of safety-critical applications, and C Global Surveyor [1], which has been designed to efficiently analyze NASA Mars missions' flight-control software. Specialization enables the analysis of larger programs with a low false-positive rate. However, the engineering of a customized static analyzer requires a protracted development effort led by experts in the field, which makes this approach difficult to reproduce in a commercial setting. In order to make static analysis commercially viable one has to streamline the process of customizing a static analyzer. It is this observation that has driven the development of CodeHawk at KT.

3 CodeHawk

CodeHawk is a high-precision static analysis tool for measuring the vulnerabilities in software. It is programmable in the sense that an analyst can customize the analysis process for a particular property of interest, and can tune the analyzer for the target code architecture. The effect of customization and tuning is to enable CodeHawk analyzers to be high precision (low false positive rate and a goal of no false negatives), and to be scalable (allowing the analysis of large code bodies). Moreover, since CodeHawk supplies mathematical evidence for its findings: positive (safe condition), or negative (error condition) its results can be independently and inexpensively checked. In addition, when CodeHawk cannot resolve whether a condition is safe or an error, it issues a warning, but also gives the precise remaining proof obligation to resolve the warning, allowing fast resolution of warnings (e.g. by adding information about the properties and assumptions of library APIs).

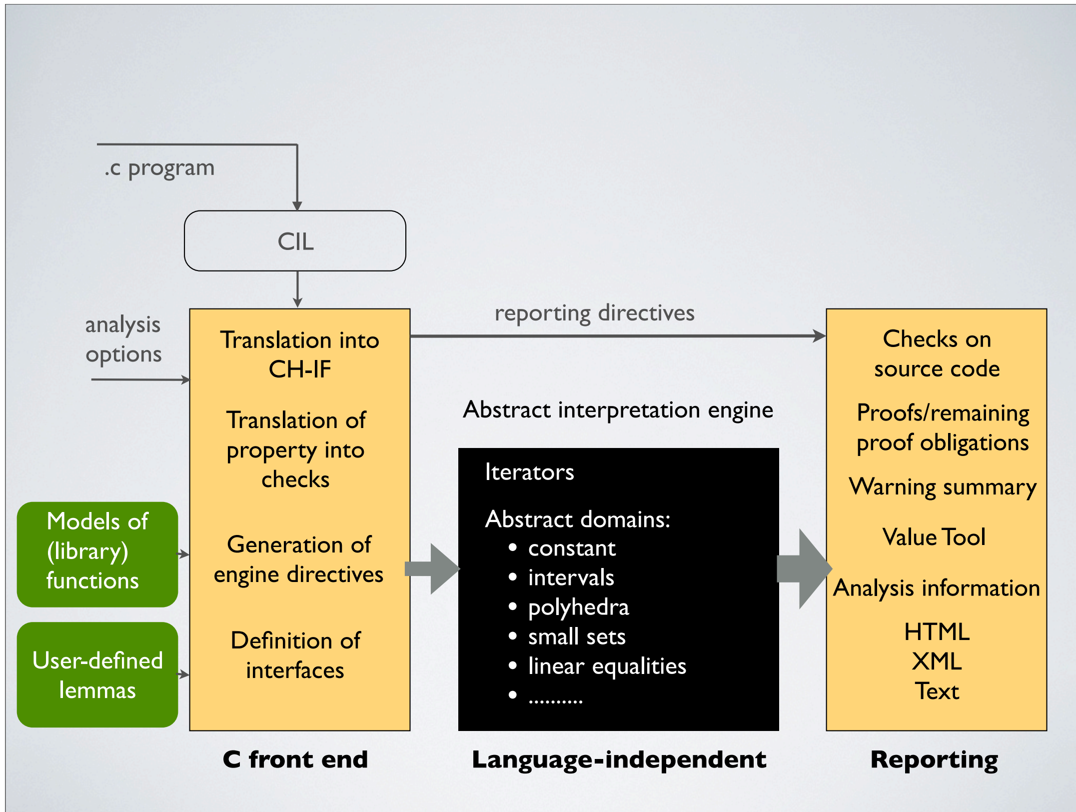

Figure 1: CodeHawk Architecture

3.1 CodeHawk Architecture

CodeHawk consists of three main parts, as shown in Figure 1. The core of the CodeHawk analysis framework is the abstract interpretation engine.

Engine The engine is independent of the programming language being analyzed: it has its own internal language. The engine contains the iterators and a collection of abstract domains.

C Front End The task of the C front end is to translate the (preprocessed) program from C into the internal engine language, to translate the property to be analyzed into a representation that is compatible with the engine representation, and to provide analysis directives to the engine such as domains to be used, structures and functions to be expanded.

External information about the program to be analyzed, such as assumptions about library and user-defined functions and user-defined lemmas are imported in and processed by the C front end before being handed over to the engine.

User Interface and Reporting CodeHawk is currently run from the command line, but Eclipsed-based graphical support is under development. It has a few dozen command-line options to control the analysis and output reports. The reporting facility receives the analysis results from the engine and connects these results with the original program at multiple levels of detail and from multiple perspectives.

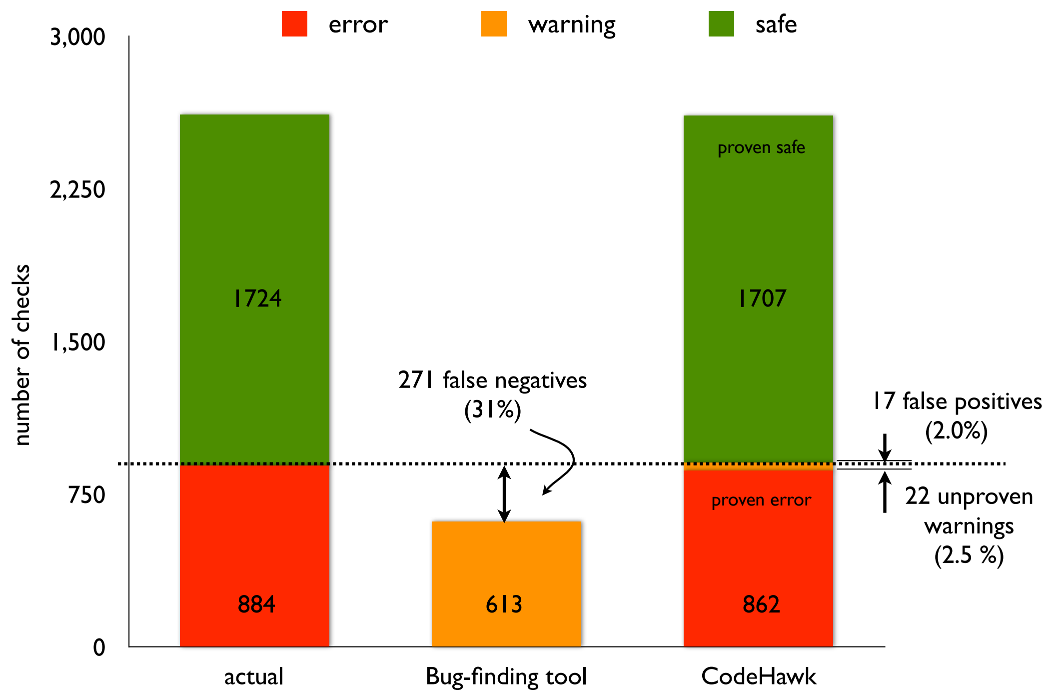

Figure 2: Results for SAMATE benchmarks 115-127

3.2 Analysis Results using CodeHawk

KT has used CodeHawk to find buffer overflow vulnerabilities with near-perfect precision in lab tests versus existing commercially available static code analyzers. In contrast to CH, existing "lightweight" or "unsound" static analyzers test only a subset of software execution paths and use only simple domains for testing. CodeHawk was able to detect all buffer violations in the NIST SAMATE benchmarks 115-1278. Its performance against a commercial "unsound" bug-finding tool is shown in in Figure 2 and the following table:

Small SAMATE Benchmarks: 2608 total buffer accesses 884 known buffer violations 1724 known safe	CodeHawk "Sound" Tool	Typical "Unsound" Tool
Errors Detected	884	613
Errors Not Detected	0	271
Analysis Time	1 minute	1 minute
Proven Errors	862 of 884	None
Proven Safe	1707 of 1724	None

CodeHawk also provides another measurement on benchmarks: which abstract domains were required to detect each error. The following table lists four domains in the first column, in increasing order of difficulty. It shows that 66% of buffer accesses in the SAMATE small benchmarks only required reasoning about constants. This is indicative of relatively simple programs.

Abstract Reasoning Domains	small SAMATE benchmarks	Percent of buffer accesses
Constants	1704	66%
Intervals	537	21%
Polyhedra	16	1%
Indirect Proof	312	12%

The next table measures the domains required to analyze all buffer accesses in the suite of medium size SAMATE benchmarks. The decreased usage of the constants domain and the increasing need for interval reasoning is indicative of a more complex benchmark. The error detection rate of the bug-finding tool for benchmarks 115-1278 was found above to be around 70%, while for the medium benchmarks, the detection rate was less than 10%. Unsound bug-finding tools will do less well on codes that require more sophisticated domains to detect errors.

CodeHawk and the commercial "unsound" bug finding tool were also tested against the SAMATE medium benchmark 1291 for BIND (the most commonly used DNS server on

Abstract Reasoning Domains	medium SAMATE benchmarks	Percent of buffer accesses
Constants	404	20%
Intervals	1529	76%
Polyhedra	67	3%
Indirect Proof	0	0%

the Internet). In this more complex program, only 4% of buffer accesses could be solved with a Constant Domain - the typical capability of commercial "unsound" tools. CodeHawk needed to use its Interval Domain (ranges for individual variables) to solve 76% of the buffer accesses and its Polyhedral Domain (linear relationships between variables) to solve the remaining 20%. With this capability, CodeHawk was able to find the Level 10 buffer overflow error that was exploited to allow unauthorized access by cyber attackers. The commercial bug-finding tool reported no errors on this benchmark.

Abstract Reasoning Domains	SAMATE benchmark 1291 (BIND)	Percent of buffer accesses
Constants	7	4%
Intervals	150	76%
Polyhedra	40	20%
Indirect Proof	0	0%

One additional KT lab test was then performed on a private benchmark created by cybersecurity experts. In this case, the vulnerabilities were more complex than the SAMATE small benchmarks and were more typical of "real world" code. As shown in the table below, in this test Codehawk found 114 of 118 (96.6%) of the known buffer violations versus only 20 of 118 (16.9%) found by the commercial tool.

Cybersecurity Benchmarks: 118 known buffer violations 884 known buffer accesses 1724 known safe	CodeHawk "Sound" Tool	Commercial "Unsound" Tool
Errors Detected	114	20
Errors Not Detected	4	98
Proven Errors	114 of 118	None

The domains required to detect these errors were more complex as shown in the table below. According to our measurements, the larger and more complex the code, the smaller the probability that the commercial tool could detect a given vulnerability.

Abstract Reasoning Domains	Cybersecurity private benchmark	Percent of buffer accesses
Constants	150	34%
Intervals	232	52%
Polyhedra	31	7%
Indirect Proof	32	7%

Because of its capability to achieve near perfect precision through customization, and its capability to produce evidence to prove its findings, CodeHawk is a unique tool for cybersecurity analysis.

References

[1] Brat, G., and Venet, A. Precise and scalable static program analysis of NASA flight software. In *Proceedings of the 2005 IEEE Aerospace Conference* (2005).

[2] Cousot, P., and Cousot, R. Static determination of dynamic properties of programs. In *Proceedings of 2nd International Symposium on Programming* (1976), pp. 106–130.

[3] Cousot, P., and Cousot, R. 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* (1977), pp. 238–353.

[4] Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *Sixth ACM Symposium on Principles of Programming Languages* (San Antonio, TX, January 29–31, 1979), ACM, pp. 269–282.

[5] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. The ASTRÉE Analyser. In *Proceedings of the European Symposium on Programming (ESOP'05)* (Edinburgh, Scotland, April 2–10 2005), M. Sagiv, Ed., vol. 3444 of *Lecture Notes in Computer Science*, © Springer, pp. 21–30.

[6] Cousot, P., and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1978), ACM Press, New York, NY, pp. 84–97.