2-1-2005

Artemis: Practical Runtime Monitoring of Applications for Errors

Long Fei

Samuel P. Midkiff

ARTEMIS: PRACTICAL RUNTIME MONITORING OF APPLICATIONS FOR ERRORS

LONG FEI
SAMUEL P. MIDKIFF

TR-ECE 05-02
FEBRUARY 2005

SCHOOL OF ELECTRICAL
AND COMPUTER ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, IN 47907-2035

Artemis: Practical Runtime Monitoring of Applications for Errors

Long Fei and Samuel P. Midkiff[*]
School of Electrical and Computer Engineering
465 Northwestern Ave.
Purdue University
West Lafayette, IN 47907-2035
{lfei, smidkiff}@purdue.edu

[*]This work was supported by the National Science Foundation by grant 0325603-CCR. The opinions expressed are those of the authors, and not the National Science Foundation.

Contents

Abstract

A number of hardware and software techniques have been proposed to detect dynamic program behaviors that may indicate a bug in a program. Because these techniques suffer from high overheads they are useful in finding bugs in programs before they are released, but are significantly less useful in finding the much harder to detect bugs in long-running programs – the bugs that are the most difficult to find using traditional techniques. In this paper we propose the Artemis[1] compiler-based instrumentation framework that complements many pre-existing runtime monitoring techniques, yielding an average asymptotic lower bound on overhead of 11% on the seven SPEC benchmarks tested.

[1]Artemis is the Greek goddess of the hunt and wild animals. Our framework guides the hunt for wild bugs.

1 Introduction

Program correctness and reliability are two of the greatest problems facing the developers, deployers and users of software. The financial implications are tremendous – a NIST report estimates that $59.6 billion dollars a year, or 0.6% of the GDP, are lost every year because of software errors [24]. Single failures of software can disable businesses like Charles Schwab and Ebay for hours and days, costing millions of dollars in revenues. Moreover, purely malicious and politically motivated sociopaths exploit software errors to disrupt civil society.

Simultaneous with the increasing risks of bug-ridden code has been the rise in complexity of individual components and in the systems composed of these components. The number of lines of code, the number of paths through a program, and the running times of programs have risen dramatically over time, reducing the effectiveness of both manual debugging and path coverage based testing mechanisms. Moreover, it is commonplace to construct instances of large systems by lashing together components such as Apache Webserver, J2EE, SQL, and so forth, where each component may be replaced by a component with similar interfaces, but different implementations. For testers of these components, coverage of a significant number of paths is impossible because of the complexity of code. Moreover, because instances of real systems are constructed independently of the developers of the individual components, *it may not be possible for the developer of a component to test the component in the same software and hardware environment in which it will execute.*

This argues for debugging techniques that can be used at runtime during production runs of the program, i.e. for debugging to be a continuous, ongoing part of deployed software. Previously developed techniques, discussed in more detail in Section 2, while invaluable in their targeted context, are typically either (i) statically based [22, 14, 3, 34, 9, 10, 37, 15, 36], and therefore cannot examine a program in the context it actually executes in, (ii) are fast enough to use at runtime, but require hand-modifying the source code and/or user input [23, 6], or (iii) are too slow to be used in production runs. Sampling across many applications has been proposed as a solution to reduce the overhead for any given instance of an application, and thereby enable production run monitoring. Sampling, however, requires many copies of a program be in place, and that the artifacts of the program be observable. This raises issues about sensitive data leaking from these programs, and the monitoring of mission critical, but heavily customized programs (and essentially single image programs) like online trading systems used by brokerage and auction sites.

In this paper we present a framework, called Artemis, to enable the use of *baseline* monitoring schemes (such as those mentioned above) with single threaded C programs. The framework reduces the overhead of the monitoring schemes sufficiently that monitoring often becomes fast enough to be used in a production system. Our framework is predicated on the following observation: because single threaded regions of programs are essentially deterministic, if the context of that region being entered is the same as the context on a previous, non-anomalous execution of a region, it is likely that the current execution will also be non-anomalous. In this case, fine grained and high overhead monitoring of the current execution instance of the region will be less profitable than when the context is different. This observation motivates the design of our framework, which effectively pre-filters execution instances, only monitoring (and incurring the cost of monitoring) for those instances which are likely to be anomalous. The baseline monitoring schemes can be any of the various schemes that have been proposed, including those in [17, 32, 2, 18, 7, 21, 23, 29]. Moreover, because our framework converges, in long running programs or across multiple runs, to a steady-state where regions are rarely monitored, it is possible to use two or more underlying monitoring techniques with our framework, even when each of the underlying techniques is too expensive to use in production runs.

This paper makes the following contributions:

- it describes the first framework that uses program context information to reduce the overhead of monitoring to an acceptable level in long-running programs, and shows how the framework is used with
  - a hardware based monitoring technique (AccMon [38]) that checks for bugs using *program counter invariance*;
  - a software based monitoring technique (DIDUCE [16]) that checks for bugs using *value invariance*;
  - synthetic monitoring technique that allows us to examine the behavior of our framework when used with arbitrarily expensive baseline techniques; and
  - a program analysis tool that performs approximate integer range analysis.
- it describes a runtime technique for tracking pointers in the face of aliasing;
- it provides experimental data showing the performance and precision of using our framework with different underlying baseline techniques.

2 Related Work

Runtime monitoring tools make use of runtime information to detect bugs that cannot be detected statically. Existing dynamic monitoring schemes fall into two categories: programming-rule-based (PRB) and statistical-rule-based (SRB).

PRB checks for violations of programming language specifications or software development specifications. For example "array index cannot exceed the array bounds" and "concurrent accesses to a shared variable should be synchronized" are the kinds of rules PRB checkers use to detect bugs. Much work has been done in this category, including Purify [17], rtcc [32], SafeC [2], Eraser [30], Jones & Kelly's tool [18], StackGuard [7], runtime-type-checking [21], [5], [25], CCured [23, 6], CRED [29], and so on. Each of the tools is capable of detecting violations against one or a few programming rules.

SRB extracts rules (*PC and value invariants*) statistically from successful runs or multiple periods of a single long-running execution, and then uses these rules to check for violations in a later execution or later periods of an execution in a long-running job. Value invariance maintains, for each variable, a set of values that the variable has held. When a value not in the set is found at runtime, it is recorded as an anomalous value. Program Counter (PC) invariance maintains for each memory location a set of program counters that access the variable. When a datum is accessed by a program location that has not previously accessed it, it is recorded as an anomalous event.

A few studies have been conducted on SRB bug detection. DAIKON [13, 12] is a pioneering system which detects value invariants at runtime. DIDUCE [16] detects bugs on the fly by automatically extracting value invariants and using them to detect violations during execution. Both DAIKON and DIDUCE use value-based invariants. Liblit et. al. [20] uses statistical analysis to find the difference between abnormal and normal runs for postmortem bug analysis. AccMon [38] detects bugs using PC-based invariants. Similarly, each SRB tool detects runtime violations of one category of statistical rule.

While the designers of runtime monitoring tools strive to make their tools as efficient as possible, runtime overhead remains one of the biggest challenges in making these tools practical. One approach to reducing overheads is to make use of static information provided by static analysis tools, compiler, or user annotations. Examples in this category include [5] and [6]. While static information alleviates the overhead problem in detecting certain categories of bugs in some programming languages, it is not a general solution and may involve considerable modification to source code [6]. More importantly, it does not solve the overhead aggregation problem that occurs when multiple tools are applied together to achieve better coverage.

Another approach to reducing overhead is to use sampling [20] to amortize the monitoring overhead among a large number of releases. This approach is only applicable when data can be collected from a huge set of sample runs, and is not a general solution for debugging a piece of software that has limited distribution. It also requires special data representations to ensure the communication efficiency and client privacy, which also limits its applicability to existing runtime monitoring schemes. A recent variant of sampling, called adaptive sampling (Chilimbi and Hauswirth [4]), uses a sampling rate inversely proportional to the frequency of code segment execution. Adaptive sampling can ensure the coverage of all executed program segments in a single run. However, adaptive sampling cannot be used to capture bugs that occur only once in the execution, which makes it ineffective in detecting the most common bugs like buffer overflow.

A third approach is to perform runtime checking in parallel with the main execution by either creating a shadow process [28] or executing the checking code speculatively [27]. These techniques are useful when there are multiple CPUs available and parallelism can be discovered between the main execution process and runtime checking process. These techniques are complementary to the Artemis framework. Artemis can use these techniques to further reduce the runtime overhead.

Our work is also related to static analysis and model checking ([22], [14], [3], [34], [9], [10], [37], [15], [36]). Model checking can be used to prove there is no error, but models are usually difficult to construct and may not always be feasible. Static checking tools are usually best-effort based tools focused on one particular category of bugs [11]. Static tools usually use program annotations or user specifications to improve precision.

Artemis differs from all of these in that it is not, primarily, a technique for anomaly detection, but rather a framework to reduce the overhead of other anomaly detection tools. The Artemis framework can be implemented using source-level or binary-level instrumentation tools like Cetus [19], ATOM [31], Trimaran [33]. We use Cetus.

3 An Overview of Artemis

In this section we give an overview of our framework, and describe its capabilities.

The problem with existing monitoring techniques is that they are expensive –

large numbers of accesses must be monitored. Our technique proposes a form of filtering to reduce this overhead. Specifically, we observe that single threaded programs are deterministic – for a given execution context the program, or a region of the program, will have functionally equivalent behaviors. We define the execution, or dynamic, context of a program region to be the program state accessed in that region. Unfortunately, checking this context can be at least as expensive as checking the sets of invariants themselves. Moreover, determining precisely the equivalence of two contexts can be difficult, particularly in the presence of pointers and aliasing. Our technique is predicated on the idea that if the *global context* (the state of all in-scope variables, method parameters, and storage reachable from these) is the same then the outcome of an execution of the region, when projected onto the outcomes of "correct" and "anomalous", will also be the same. The global context is, of course, an approximation to the whole context, and program behavior can be non-linear (i.e. small changes in the context can lead to large changes in the outcome, including an incorrect result) but our experimental results show that with this sampling we can catch errors in programs, while significantly reducing the monitoring cost.

We approximate global contexts depending on the values and types of the object in the context as described in Section 4.1. As always, our goal is to minimize the overhead of monitoring and maintaining the state of the context, while gaining sufficient information about the actual context to be useful in finding potentially anomalous execution instances of the region to monitor.

The Artemis framework is used as follows. The program is automatically instrumented by the compiler with context checks and the monitoring required by the baseline monitoring technique. In the current implementation, a context checking test consists of an `if` statement that selects one of two versions of a procedure. The first version is unchanged from the original program, and the second version is instrumented with the baseline monitoring scheme. The program can then be optionally executed, with a profile of "normal" values collected to both train the context checks and the baseline monitoring schemes. After the optional training runs, the program is put into production. The initial contexts are taken from training runs, or prior production runs. When the current context for a region is found to not belong to the context being tested against, the baseline monitoring technique(s) are turned on for the region, and the invariant context for the region is updated by adding the values of the current context to it. After the execution any anomalies found by the monitoring techniques are reported, and the newly formed contexts are saved as *invariant profiles* (or profiles, for short) for use by the next run of the program.

We note now, and discuss in further detail later, that over time the framework becomes trained and the invariant of a region's context matches almost all contexts that lead to non-anomalous behaviors. As a consequence, very few regions are subjected to fine-grained monitoring, and the runtime overhead of the Artemis framework becomes increasingly close to the cost of comparing the contexts. This means that the granularity of regions, and therefore the number of contexts tested are strongly correlated to the asymptotic overhead of the system. Because procedures introduce parameters, and are natural units of functionality, making regions larger than procedures seemed unwise, whereas making regions smaller than procedures (e.g. at loop nest boundaries) seemed likely to lead to untenable overheads in non-numerical program, and with little gain in precision. In numerical programs, where loop nests may contain a large amount of work, defining regions at this finer granularity may prove useful, and is currently being investigated. Profiling, conducted during training runs, can provide information to guide region formation, but this not done in the current version.

4 Implementation

4.1 Context Representation

As discussed previously, exactly representing and comparing contexts is too expensive. In the presence of pointer aliasing, this check potentially requires comparing the entire memory space. Therefore, it is infeasible to record the precise context for future comparison, or to perform a precise comparison of the context at the entrance of a code region.

We approximately model the context of a code region as a set of *independent* variables and pointers (ignoring the correlations between the elements) that are accessed in the code region. Each variable in the set is represented by its corresponding `int` value (by directly casting, or hashing multiword objects), and each pointer in the set is represented by its type in declaration. The scheme of approximating a variable's value by an integer value and approximating a memory object being pointed to by the pointer's runtime type is inspired by the techniques in [16] and [8].

Previously observed contexts are recorded as *context invariants*. Each context invariant contains these components: a list of value invariants (each value invariant records all previously seen values of one variable in the context), and a table of pointer types (each entry contains a mask representing the types of pointers which

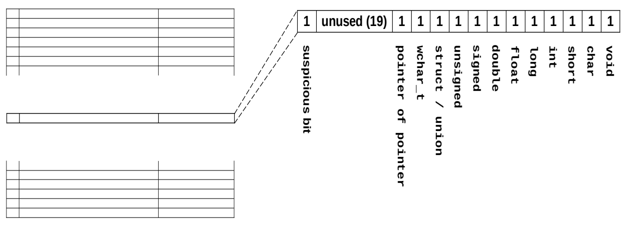

Figure 1: Pointer type invariant table. The table is indexed by the lower 10 bits of the pointer value (cast to `unsigned int`). Each entry contains a bit mask representing the types of pointers pointing to that location observed previously.

previously pointed to the same address, indexed by the pointer value). Context invariants are bound to regions, i.e. each context invariant records only previously seen contexts at the entrance of a code region.

A variable's *value invariant* uses the value invariant representation described in [16]. Each value invariant contains two parts: a `base_value` and a `mask`. Each bit in `base_value` records the value of that bit when the value invariant records its first integer value; each bit in `mask` records if a different value of that bit has been observed in a later recording. Given first integer value of V, the value invariant is initialized to $\texttt{base\_value} = V; \texttt{mask} = \neg 0$. Suppose the current observed value is V': it matches the invariant only if $(V' \oplus \texttt{base\_value}) \wedge \texttt{mask} == 0$, where $\oplus$ is *exclusive or*. The invariant can be relaxed when a new value is observed by updating the mask: $\texttt{mask} = \texttt{mask} \wedge \neg(V' \oplus \texttt{base\_value})$. The Artemis framework can also be configured to prevent invariant updating after sufficient learning, which is an effective mechanism to prevent malicious training.

In order to achieve high efficiency and to handle pointer aliasing, a global pointer type invariant table is used to record the runtime types of pointers. The layout of the pointer type invariant table is shown in Figure 1. The table (1024 entries) is indexed by the lower 10 bits of the pointer value (cast to `unsigned int`). Each entry in the table holds a type invariant, which is a bit mask representing the pointer types observed previously. The MSB of each entry is `suspicious bit`. For example, if a previously seen pointer's declared type (from its declaration statement) is `unsigned int`, the bits corresponding to `unsigned` and `int` will be set. In this scheme, different pointers pointing to the same memory object will be indexed into the same entry in the table, avoiding the global pointer aliasing problem.

Each entry in the pointer type invariant table is initialized to 0. To check if a pointer's type matches the type invariant at runtime, the lower 10 bits of its pointer value are used to index the pointer type invariant in the table. The pointer type invariant is then compared with a statically assigned type mask (with only bits corresponding to the pointer's declared type set to 1). Suppose the pointer type invariant is `ptr_inv` and the pointer's type mask is `ptr_type_mask`, then the pointer's runtime type matches the pointer type invariant only if $(MSB(\texttt{ptr\_inv})) \vee ((\texttt{ptr\_type\_mask} \wedge \texttt{ptr\_inv}) \oplus \texttt{ptr\_type\_mask}) == 0$. If the pointer is not `NULL`, and is a pointer to another pointer, the runtime type of the target pointer is also checked. The pointer type invariant can be updated if necessary by $\texttt{ptr\_inv} = \texttt{ptr\_inv} \vee \texttt{ptr\_type\_mask}$.

In addition to resolving aliasing, the pointer type invariant table is designed to reveal at runtime the potential existence of an important category of bugs overlooked by most existing tools: referencing by an incompatible pointer. For instance, using a `char`-typed pointer to reference to a `double`-typed variable will result in a pointer pointing into the middle of a variable, and dereferencing a `double`-typed pointer pointing to a `char`-typed variable will result in corruption of data. By checking the runtime type of a pointer against the pointer type invariant, which represents the types of "compatible pointers types", a potentially mistyped pointer can be detected during context checking. The observant reader may have noticed that the pointer type invariant table itself is a useful tool to detect mistyped pointer dereference errors. A further exploration of this capability is beyond the scope of this paper.

To determine if the current context representation matches a context invariant at the entry to a code region, Artemis checks each variable and each pointer in the set against their corresponding value invariant and pointer type invariant. If any of the checks return mismatch, the current context is considered to be different from any previously observed context.

Context at the program level is represented using `max_argument_length` (maximum string length of any argument in `argv[]`) as the program level approximation. `max_argument_length` records the length of the longest argument observed previously, and is carried across different runs of the program via the invariant profile (discussed in Section 4.4). If the length of any argument (`argv[i]`, `i = 1, 2, ... argc-1`) in the current commandline exceeds `max_argument_length`, Artemis sets the "suspicious bit" of the entry in the pointer type invariant table corresponding to `argv[i]`. This ensures that if a pointer `p` is aliased to an over-sized commandline argument, context checks at the entrances of code regions accessing `p` will report a mismatch, and the baseline detection tool will be activated to monitor for potential violations. `max_argument_length` is preset to be 20, which is large enough to allow most commandline arguments, yet too small for most exploits from commandline arguments [26].

In the default configuration, Artemis updates the invariants (variable value invariants, pointer type invariants, and `max_argument_length`) when it reports a context mismatch. This ensures that the new observations can be included in the context invariant if the new context turns out to be a legal context. It can also be configured to freeze context invariants to protect itself against malicious training.

4.2 Inserting Context Checks

We use the Cetus C compiler [19] to insert context-checking instrumentation into the source code. At each procedure entrance, we insert instrumentation to check the current context representation against the learned context invariants (as discussed in Section 4.1) at that particular program point. Code is also inserted to turn on monitoring if the current context representation does not match the learned context invariant, and to turn off monitoring otherwise. Before we turn on/off monitoring, the current monitoring state (ON or OFF) is saved, and restored when the function returns. This ensures that a decision made within the callee does not change the monitoring in the caller after the callee returns.

If the baseline monitoring scheme is binary-level or hardware-based, we turn on monitoring before each call to a library function whose source code is not available. The current monitoring state is remembered before the library call, and is restored after the library call returns. For standard C libraries, an optimization effort is underway to build a list of *safe library functions* for which monitoring is not needed. This technique is similar to the way CCured [6] handles standard C library calls. With this technique, monitoring overhead within safe standard C libraries can be reduced.

The handling of small functions is a special case. Since the overhead savings from not monitoring a region should exceed the overhead of checking the context and deciding whether or not to monitor, we always leave monitoring on for small functions. Our assumption is that it is not profitable to check the context in order to save the monitoring overhead of very few statements. In our current implementation, small functions are those functions that contain fewer than five *simple* statements (i.e. no loops, `switch` statements, etc.). The threshold is a tunable configuration parameter. It should be set smaller when the baseline monitoring scheme is expensive since the monitoring overhead on a few statements may still exceed the context check overhead in that case.

The compiler instrumentation of the Artemis framework is easily configured by a few parameters. By default, Cetus inserts at a procedure entrance context checks and the code to turn on/off baseline monitoring based on the result of the context check. Code replication (discussed in Section 4.5) is enabled if the baseline tool is based on source-level instrumentation; library call wrapping is enabled if the baseline tool is based on binary instrumentation or monitoring hardware.

4.3 Interfacing with Baseline Monitoring Schemes

Artemis is designed to be a general framework that works with source-level, binary-level and hardware baseline monitoring schemes. The interface between the baseline monitoring scheme and Artemis is a switch, which is used to turn the baseline monitoring on and off. This simple interface makes it easy to adapt an existing monitoring tool to work with Artemis.

4.4 Invariant Profile

In its default configuration, Artemis uses an *invariant profile* to avoid losing the learned context invariants when the program terminates. During a normal run, each time a new context is observed, Artemis learns the new negative observation by updating the invariants in the context invariant. Without profiling, this learning process needs to be started from the beginning in every run. Artemis reduces unnecessary activations of the baseline monitoring tool during the learning process by dumping the context invariants into an invariant profile before the program terminates, and loading the invariant profile the next time the program runs.

Because pointers can point to dynamic objects whose addresses in memory are not necessarily the same in two different runs, Artemis does not dump the pointer type invariant table as part of the invariant profile.

Invariant profiling also makes it possible for software companies to build profiles during their in-house testing phase, and then to ship the released software with runtime monitoring code together with the invariant profile. This makes it possible to use heavy-weight runtime monitoring schemes in real world applications at the cost of a light-weight monitoring scheme.

4.5 Optimizations

If the baseline tool works by instrumenting the source code with checking functions, *code replication* [1, 20] can significantly improve the performance of the resulting code when interfacing the baseline tool with Artemis. Figure 2 shows an example where Artemis works with a runtime null pointer checker. The naive

```
foo()
{
    if (context match){
        MON_OFF;
    }else{
        MON_ON;
    }

    ...
    if (MON_ON){
        baseline_check_NULL_ptr(p);
    }
    *p = 2;
    ...
}
```

(a)

```
foo()
{
    if (context match){
        ...
        *p = 2;
        ...
    }else{
        ...
        baseline_check_NULL_ptr(p);
        *p = 2;
        ...
    }
}
```

(b)

Figure 2: (a) – a naive interface between baseline tool and Artemis; (b) – applying code replication on function body

version (Figure 2(a)) allows Artemis to control the monitoring by inserting the testing code at each monitoring call site. This simple scheme is not desirable in practice because the existence of a branch precludes compiler optimizations that could have been applied. A more efficient way of interfacing with Artemis is presented in Figure 2(b). By creating two versions of the function body (one instrumented with runtime monitoring, one not instrumented), execution can take a fast path when Artemis determines that no monitoring is necessary. Because most context checks succeed after some initial learning, the fast path is usually taken, completely avoiding the overhead of the baseline monitoring. In our current implementation, Artemis replicates the function body by default. [2]

5 Experimental Results

Benchmark	ammp	bzip2	equake	gap	mcf	parser	vpr	**MEAN**	**STD**
running time increase (%)	0.28	16.71	10.45	18.86	-0.46	19.42	11.75	11.00	8.29
binary size increase (X)	0.84	0.84	0.49	1.23	0.57	1.01	0.92	0.84	0.25
profi le size (k)	2.11	1.37	0.45	12.81	0.36	4.90	4.11	3.73	4.36

Table 1: Time and space overhead of using Artemis framework (with no profile) without baseline tools.

We present the results of four experiments. In the first experiment, we focus on the reduction in monitoring overhead when using the Artemis framework

[2]Currently, the only exception to the default policy is when there are `static` local variables inside the (nested) compound statement. It is ongoing work to develop the analysis and transformation to allow code replication in the presence of `static` variables.

with baseline monitoring schemes. We also present the asymptotic overhead of the Artemis framework, simulations showing Artemis framework overheads with generic monitoring schemes, and how the monitoring overhead approaches an asymptotic overhead over time. In the second experiment, we use the Artemis framework on AccMon, a state-of-the-art hardware-based monitoring tool for memory bugs [38]. We present the interface between Artemis and AccMon, the monitoring overhead, and three examples of detecting real bugs in benchmark programs. In the third experiment, we present an application of the Artemis framework on a program analysis problem – dynamic whole program integer range analysis. We present its runtime overhead improvement and precision in discovering the integer ranges. In the fourth experiment, we use our framework with a variant of DIDUCE [16] for C (C-DIDUCE) – a state-of-the-art software monitoring scheme based on value invariants [16]. We present the implementation of C-DIDUCE, its interface with Artemis, the runtime overhead improvement and the accuracy in detecting runtime invariant violations. All these experiments are conducted on a DELL Precision 350 workstation (3.0GHz Pentium IV with Hyperthreading, 1.5G memory) running RedHat Linux 9.0 with `gcc 3.3.3`. All of the programs are compiled with `gcc -O2` except those used with AccMon, which use the compiler for the iWatcher simulator compiler, a variant of `gcc` for `MIPS`.

5.1 Overhead Limits

As discussed in Section 3, Artemis' context representation will converge to the closure of previously observed contexts over time. Consequently, the runtime monitoring overhead approaches its asymptotic overhead, i.e. almost all of the runtime monitoring is skipped and the only overhead is that of context checking. The context checking overhead is the asymptotic lower bound of Artemis with any baseline monitoring scheme, regardless of the overhead of the baseline scheme.

Table 1 shows the overhead of running Artemis without any baseline monitoring tools, measured using seven SPEC 2000 C programs. No invariant profile is loaded. Using Artemis alone adds 11% overhead on average with a standard deviation of 8.29%; the resulting binary is 0.84 times larger on average with standard deviation of 0.25. Since the instrumented programs suffer only context checking overhead under this configuration, this is the theoretical asymptotic lower bound of the overhead of using Artemis together with any baseline monitoring scheme. This lower bound is attainable only when Artemis and the baseline monitoring scheme are *perfectly interfaced*, i.e. no baseline monitoring overhead is incurred when all the context checking succeeds. For example, if all the procedures are replicated as shown in Figure 2(b) and the fast path executes as efficiently as the original procedure body, only context checking overhead remains when all the contexts match the context invariants. However, the interface shown in Figure 2(a) is not perfect because the `if(MON_ON){}` statement adds some overhead whether or not the context check succeeds. In practice, this lower bound is often not attainable for several reasons. First, not all the function bodies are replicated (as discussed in Section 4), thus the interface is not perfect in all the functions. Second, the replicated version is not as efficient as the original version since code replication incurs overhead [1, 20]. Third, monitoring may happen in library calls if the call site is wrapped (as discussed in Section 4.2). Finally, because we do not dump the pointer type invariant table into the profile, Artemis incurs some initial context check failures when pointers are involved in the contexts. Therefore, all of the monitoring is not skipped even if the context was seen in a previous run of the program. Although the asymptotic lower bound is not always achieved, we still achieve a significant overhead reduction when using the baseline monitoring scheme with the Artemis framework, as will be shown in this section.

To measure overhead when using various baseline monitoring schemes, we use a simple monitoring simulator. Our simulator models a generic baseline monitoring tool which inserts a call to a checking function at program points of its interest. This is by far the most widely used monitoring paradigm in today's monitoring tools (e.g. Purify, DIDUCE). Without any knowledge of what program points should be monitored and what the check function actually does, our simulator inserts a dummy checking function call at a program point with probability *p*. The parameter *p* and the overhead within the dummy check function can be adjusted to cause the desired monitoring overhead. Artemis controls an `ON` and `OFF` switch. The checking function is called only when the monitoring is turned on.

Figure 3 shows how the overhead of {Artemis + simulated baseline monitoring tool} changes along with the baseline monitoring overhead. The data is obtained on SPEC 2000 benchmark programs (those used in Table 1) without profile information from previous runs (this is the case when the program is executed for the first time). Each of the seven benchmark programs is run five times, each time with a different simulated baseline monitoring scheme and overhead. Each dot in the figure represents an (x, y) pair, where x is the baseline monitoring overhead and y is the overhead of baseline monitoring controlled by the Artemis framework. As can be seen, by eliminating unnecessary monitoring calls, the overall overhead of {Artemis + simulated baseline monitoring tool} increases much slower than

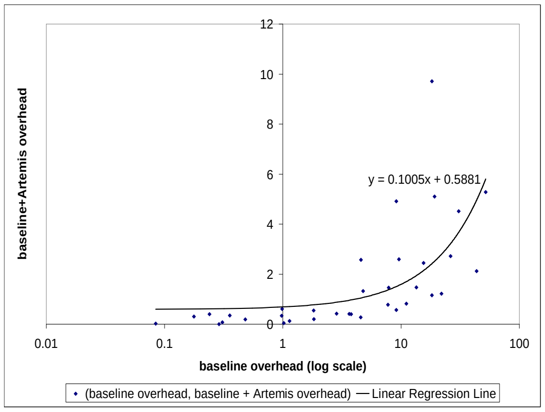

Figure 3: simulated overhead of the baseline monitoring and baseline + Artemis without profile.

the baseline overhead. A simple linear regression shows that when the baseline monitoring scheme causes x times slowdown, using Artemis with the baseline scheme (without profile) will bring the overhead down to $y = 0.1005x + 0.5881$. As discussed previously, when a profile is used the monitoring overhead will approach the asymptotic overhead shown in Table 1. Although the performance varies with the program and the input, the trend shows that using Artemis brings significant performance benefits when the baseline overhead exceeds a certain threshold. This threshold is 65% for SPEC-like programs without a profile (by solving the equation $x = 0.1005x + 0.5881$), and 11% (theoretical asymptotic lower bound) for long-running programs or with a profile. Given that most runtime monitoring schemes suffer much higher overheads, and the overhead is even higher when multiple monitoring schemes are used in combination, Artemis' approach has significant performance benefits, which increases the applicability of the underlying baseline monitoring scheme.

To demonstrate the convergence of overhead using $\{$Artemis + simulated baseline monitoring tool$\}$ in long-running programs and across multiple runs, we use the `bzip2` benchmark with a simulated "representative" baseline monitoring tool which has a runtime overhead of approximately 4X (this is a "representative overhead" of many bug-detection tools). Figure 4 shows the convergence of overhead (of the first run, without profile) of $\{$Artemis + simulated baseline monitoring tool$\}$ when the memory image size increases (the larger the memory image, the longer the running time). As seen from the figure, the overhead approaches the lower bound as the memory image size increases. Figure 5 shows the convergence

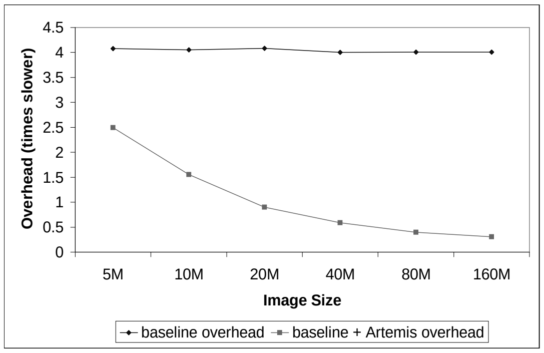

Figure 4: Convergence of overhead of bzip2 when the input size increases. The inputs are randomly generated ACSII files of various sizes.

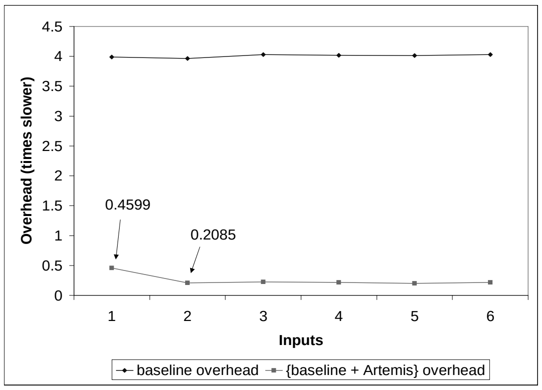

Figure 5: Convergence of overhead of bzip2 over multiple runs with profiles. The inputs are randomly generated ACSII files.

of overhead of {Artemis + simulated baseline monitoring tool} over multiple runs. Each run uses a different input. Each input is a randomly generated ASCII file of size 5M. A profile is used by Artemis to carry invariants from a previous run to the next except for the first run which starts without a profile. As shown in the figure, with the Artemis framework the overhead quickly converges to around 21% after the first run, reducing the overhead by a factor of 18.8.

5.2 Artemis + AccMon

AccMon [38] is a state-of-the-art hardware-based runtime monitoring tool for memory-related bugs. It relies on PC invariants to detect illegal accesses. During the training phase, it forms the set of PC's that access each monitored datum in each monitored memory region. During the detection phase, if a memory region is accessed by an instruction whose PC is not present in the memory region's legal PC set, an alert is issued.

AccMon is built on the iWatcher infrastructure [39]. iWatcher allows a user to associate a user-specified checking function with each monitored memory region. When the monitored memory region is accessed, the checking function is automatically triggered by the hardware without generating an exception to the operating system.

To use AccMon with Artemis, we make one change to the checking function used in AccMon: each time the checking function is invoked, it tests a flag. If the flag is set to `OFF`, the checking function returns without performing the checking, otherwise it checks for PC invariants violations as usual.

In our experiments, AccMon is configured as described in [38] except that we do not use thread level speculation. We use programs with real bugs (see [38]) in our experiments. Due to compatibility problems between Cetus and the simulator's backend compiler, we are only able to build `ncompress-4.2.4`, `polymorph-0.4.0` and `gzip-1.2.4` with the Artemis framework. `ncompress` is a compression and decompression utility that is compatible with the original `UNIX` compress utility, `gzip` is a popular compression utility provided by the GNU project and `polymorph` is a tool to convert Windows-styled file names to `UNIX` file names.

All three of these programs have buffer overflow problems. In `ncompress-4.2.4`, input file name longer than 1024 bytes overflows a stack buffer and corrupts the function's return address, in `gzip-1.2.4`, an input file name longer than 1024 bytes overflows a global buffer, and in `polymorph-0.4.0` an input file name longer than 2048 bytes overflows a global buffer. If this overflow in `polymorph-0.4.0` is not detected, the global buffer is then copied into a stack buffer using `strcpy()`, which corrupts the function return address. These bugs represent the buffer overflow bugs most frequently exploited by malicious users [26].

For all three programs, the Artemis framework detects a program context change and sets the monitoring to `ON` at the functions where the overflow first happens. For `ncompress-4.2.4` and `gzip-1.2.4`, AccMon then immediately detects the violation. For `polymorph-0.4.0`, AccMon is activated but does not detect the bug when the global buffer first overflows because no memory region monitored by AccMon is overwritten. Artemis does not detect any context change when the global buffer is used to overflow the stack buffer (the second buffer overflow). But the bug is detected by Artemis' library call wrapping mechanism, which turns on monitoring before `strcpy()`. AccMon does not catch

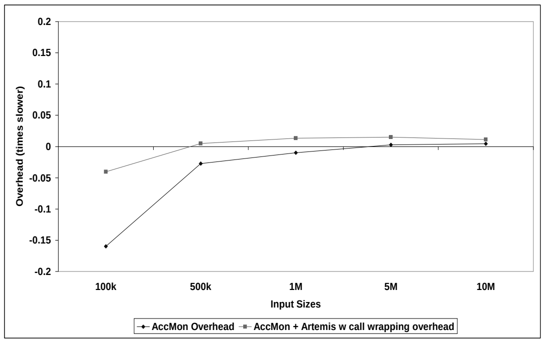

Figure 6: Monitoring overhead (AccMon only and {AccMon + Artemis}) of `ncompress-4.2.4` when the input size changes

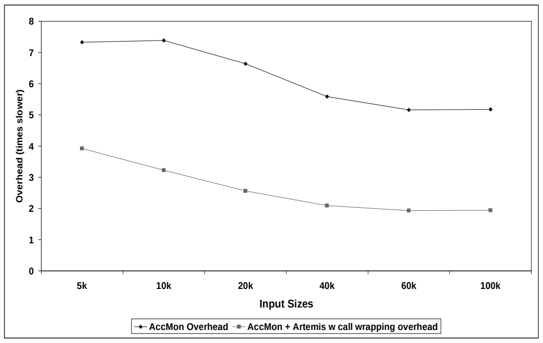

Figure 7: Monitoring overhead (AccMon only and {AccMon + Artemis}) of `gzip-1.2.4` when the input size changes

the first overflow because it only overflows the buffer by 100 bytes and does not access a region monitored by AccMon.

We evaluate the overhead reductions when using Artemis by providing inputs of different sizes to `ncompress-4.2.4` and `gzip-1.2.4`[3]. The inputs are randomly generated ASCII files. Figure 6 shows the performance on `ncompress-4.2.4`. Because AccMon has an extremely low overhead (and even a small speedup on some inputs) on this benchmark with the given data set, using our framework results in a small slowdown. Figure 7 shows the performance on `gzip-1.2.4`. Because AccMon has relatively high overhead, using our framework significantly improves the monitoring overhead by a factor of 2.67.

[3]The running time of `polymorph-0.4.0` with the default input data set is signifi cantly less than the time to initialize the Artemis framework data structures, and so we do not report numbers for it since our framework is intended for long running programs.

5.3 A Practical Dynamic Whole-Program Integer Range Analysis

Integer range analysis computes the ranges of integers in the program. It is a useful tool for program analysis and debugging purposes. For instance, it is shown in [34] that the array bounds checking problem is a special case of integer range solving. By extracting the array subscript expression and assigning it to a unique temporary local integer variable, any array bounds check problem can be converted into an integer range solving problem by testing if the range of the temporary variable exceeds the array bounds. In applications with resource constraints (e.g. embedded systems), integer range information can be used to guide system-specific optimizations (e.g. determining a good initial buffer size). We note that even approximate information is useful for these purposes.

When applied to array bounds check, static integer range analysis suffers a severe precision loss (as reported in [34], the false alarm to real vulnerability ratio is 10 to 1, and missed one real bug in `sendmail 8.7.5`) due to its conservativeness and pointer aliasing. Although runtime integer range analysis has the advantage of using runtime information to improve precision, its application is limited by its overhead, particularly when integer range analysis is used to provide information to other analyses or runtime systems. That is, adding the additional overhead of runtime integer range analysis to the other already high runtime overheads is usually not profitable.

With the Artemis framework, however, the overhead is less of a concern. Because Artemis approaches its asymptotic overhead regardless of the baseline monitoring scheme, even if a combination of monitoring schemes are used, runtime integer range analysis can be efficiently used to assist other runtime analyses or monitoring tools. For example, with integer range information, even a nonprofessional programmer can write an efficient and fully compatible array bounds checker by comparing the ranges of array subscript expressions to the array's declared length, while the state of the art runtime checking tools like StackGuard [7], Jones & Kelly's tool [18], and SafeC [2] all involve advanced compiler techniques and incur runtime overheads of $69\% - 125\%$ (StackGuard Canary), 1200% on average (J&K), and $130 - 540\%$ (SafeC), and often suffer from compatibility problems because of special tricks used in code generation.

We implement a simple integer range analyzer, which probes integer variable's value when it is are read or written[4]. In our current implementation, we do not

[4]With sufficient pointer aliasing information, we can eliminate some of the probes

break an expression into multiple subexpressions and evaluate the intermediate values. To interface with the Artemis framework, we build a switch to skip probing when the monitoring is turned off.

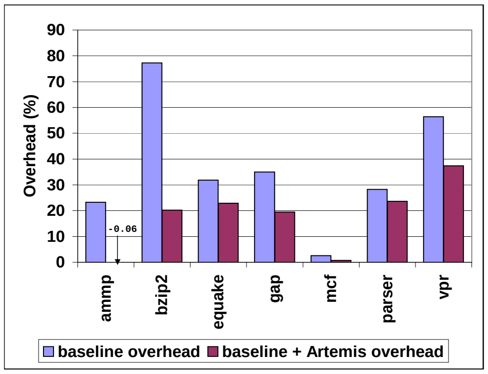

Figure 8: Overhead of the baseline whole-program integer range analysis and the overhead of the baseline analysis + Artemis

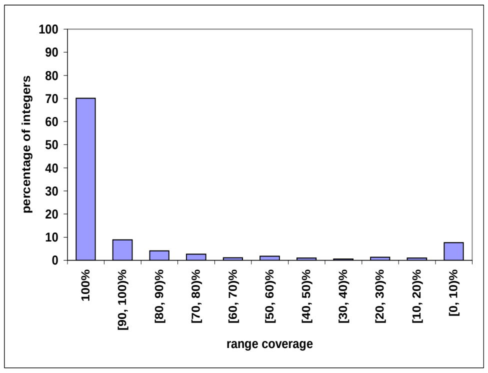

Figure 9: Accuracy of the whole-program integer range analysis using baseline analysis + X43. Each bar in the graph represents the percentage of the integer ranges that have the accuracy given below (compared with the ranges found by the baseline analysis).

We apply this analyzer to seven SPEC 2000 benchmark programs, and compare the results obtained when this analyzer is used with and without Artemis. Figure 8 shows the improvement in overhead when the programs run for the first time (no profile) with and without the Artemis framework. The overhead will approach the asymptotic overhead over time. Figure 9 shows the average accuracy

when the values are read.

on the ranges discovered. From the figure, we can see that when using the Artemis framework, 70.12% of ranges are detected precisely, and 79.02% ranges are detected with 90% or higher precision[5]. This accuracy is good enough to provide information to optimize the initial buffer size and to detect array out-of-bound errors Statistically, we have $> 70\%$ chance to detect off-by-one-byte overflows, and $> 79\%$ chance to detect those overflow by more than 10%. This is very good compared to existing buffer-overrun detectors evaluated in [35].

5.4 Artemis + DIDUCE

DIDUCE [16] is a state of the art runtime bug detection tool for Java programs. DIDUCE's bug-detection mechanism is based on runtime value-based invariant detection and checking. Violations of learned invariants are considered indications of possible bugs. DIDUCE instruments the original program (class files) with calls to the DIDUCE runtime system, passing the values of tracked expressions to DIDUCE checking and reporting functions. The runtime system is responsible for learning invariants and detecting invariant violations. DIDUCE checks expressions at the following program points: 1. object read and writes (including arrays); 2. static variable read and writes; 3. procedure call sites. The expressions DIDUCE tracks by default include: 1. the value being read or written; 2. the non-array parent object of a field accessed; 3. the difference between the values of the location accessed before and after a write operation. DIDUCE's value invariance is based on integer types only, ignores all values of floating point data types, and handles references by taking the hashcode of the String object containing the name of its runtime type. DIDUCE ranks the invariant violations it detects based on *confidence change* before and after an invariant update (only if the value does not match the invariant), where *confidence* is defined as the ratio of the number of times the expression has been evaluated and the the number of values the corresponding invariant can accept.

Because the current Artemis implementation works on C programs only[6], we implemented a C variant of DIDUCE (C-DIDUCE) to work with the Artemis framework. Our implementation is based on [16]. The C variant conforms to

[5]We conservatively consider coverage to be 0% if the range found by $\{$baseline + Artemis$\}$ is not a subset of the corresponding range found by baseline. This rare case happens when an integer stores a memory address, is assigned a random value, or is assigned an integer value read from input file.

[6]The authors see no technical difficulty to do a similar implementation for a different language.

Benchmark	C-DIDUCE overhead	D + A overhead	Top 5	Top 10	Top 20
ammp	4.92%	-0.63%	4	8	13
bzip2	80.59%	21.38%	2	6	14
gap	351.64%	42.17%	0	1	4
mcf	13.45%	0.05%	5	6	16
parser	81.50%	54.20%	0	3	8
vpr	56.48%	20.66%	0	2	5
Average	98.10%	22.97%	1.83	4.33	10
Overhead Ratio: 23.42%			Top 20 Coverage: 50%		

Table 2: Overhead and coverage of using C-DIDICE with Artemis. ‘D + A’ in the top row stands for {C-DIDUCE + Artemis}. y in ‘Top X’ column means “among the top X invariant violations found by C-DIDUCE, y of them are still among top X invariant violations when using C-DIDUCE with Artemis”.

the specifications given in the paper as strictly as possible. Due to differences in the language features of Java and C, we made our own implementation decisions when no corresponding specification from the original implementation is applicable to the C variant. Our general strategy is to avoid adding features that are not proven to be effective in [16].

C-DIDUCE is different from the original implementation in the following places: 1. Because C is not an object-oriented language, we track program points which read from or write to global variables instead of objects. `static`-typed objects are also tracked. 2. For multi-word user-typed objects (including `struct`), we cast the first word into an integer value and use this as the value of the object. 3. Since C has no object references or inheritance relations, we ignore pointer types. Otherwise, C-DIDUCE is implemented the same as the original DIDUCE (including the violation ranking scheme). C-DIDUCE instruments the source code with calls to the C-DIDUCE runtime library (containing C-DIDUCE checking and reporting functions).

Since there are no C benchmarks with bugs suitable for DIDUCE, we measure the overlap the invariant violations in the SPEC 2000 benchmarks detected and ranked using C-DIDUCE and {C-DIDUCE + Artemis}. This is justified since C-DIDUCE (and DIDUCE) only detect and rank runtime violations of invariants – they do not actually identify bugs. Therefore, the extent that Artemis changes the rankings given by C-DIDUCE is indicated by how many violations detected by C-DIDUCE and ranked top X is still detected and ranked top X when using C-DIDUCE with Artemis.

We used C-DIDUCE to detect and rank runtime invariant violations in six

SPEC 2000 programs[7]. We then use C-DIDUCE controlled by Artemis on the same programs. Table 2 reports the running times of C-DIDUCE and {C-DIDUCE + Artemis}, and how many of the Top 5, 10, and 20 violations found by C-DIDUCE are still ranked Top 5, 10 and 20 in {C-DIDUCE + Artemis}. On average, using C-DIDUCE controlled by Artemis detects 50% of the original top 20 violations at 23% of the original overhead.

We interpret the loss of precision as a combination of two factors. The first is a result of the loss of precision when approximating the context with our context representation. The second is due to DIDUCE's violation-ranking scheme. The *confidence* value is computed as a function of the number of times the invariant is evaluated. Because Artemis reduces overhead by skipping monitoring when possible, C-DIDUCE computes a different number of times the invariant is evaluated, which leads to ranking differences.

6 Conclusions

As hardware costs have fallen, and organizations have become increasingly dependent on computers, reliability and productivity have become increasingly important. Runtime monitoring of programs for execution anomolies are potentially invaluable tool for increasing the robustness of programs and the productivity of programmers by giving them clues to why long running programs failed. The framework described in this paper works with wide range of monitoring tools, imposing an asymptotic overhead of only 11%, and giving speedups of up to **3.2** times using third party monitoring tools on SPEC benchmark programs. It does this not by random sampling, but by using runtime properties of the program to determine what regions of a program need to be examined more closely. Because the Artemis framework reduces the overhead of the underlying monitoring framework, it makes runtime monitoring a reality for many programs, and dramatically increases the scope of applicability of the underlying monitoring technique.

7 Acknowledgments

We would like to thank the members of PROBE project, in particular Pin Zhou, for access to the AccMon software and their help in running it.

[7] `equake` is a fbating-point-intensive program, which doesn't show any violations on the invariants C-DIDUCE tracks.

References

[1] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 168–179. ACM Press, 2001.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301. ACM Press, 1994.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[4] T. M. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 156–164, 2004.

[5] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269. ACM Press, 2002.

[6] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. Ccured in the real world. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 232–244. ACM Press, 2003.

[7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[8] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252. ACM Press, 2003.

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, Oct. 2000.

[10] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*, pages 57–72. ACM Press, 2001.

[11] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Proceedings of VMCAI '04*, 2003.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[13] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd international conference on Software engineering*, pages 449–458. ACM Press, 2000.

[14] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: a tool for using specifications to check code. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 87–96. ACM Press, 1994.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, 2002.

[16] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM Press, 2002.

[17] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter Technical Conference*, 1992.

[18] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Third International Workshop on*

Automated Debugging, pages 13–26. Linkoping University Electronic Press, 1997.

[19] S.-I. Lee, T. A. Johnson, and R. Eigenmann. Cetus – an extensible compiler infrastructure for source-to-source transformation. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2003.

[20] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 141–154. ACM Press, 2003.

[21] A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps. Debugging via run-time type checking. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 217–232. Springer-Verlag, 2001.

[22] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. Cmc: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[23] G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139. ACM Press, 2002.

[24] Software errors cost u.s. economy $59.5 billion annually, 2002. NIST News, Release 2002-10.

[25] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178. ACM Press, 2003.

[26] A. One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[27] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the 10th International Conference on Architectural support for programming languages and operating systems*, pages 184–196. ACM Press, 2002.

[28] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Software Practice and Experience*, 27(1):87–110, 1997.

[29] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004.

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[31] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 196–205. ACM Press, 1994.

[32] J. L. Steffen. Adding run-time checking to the portable c compiler. *Software Practice and Experience*, 22(4):305–316, 1992.

[33] http://www.trimaran.org/.

[34] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[35] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, San Diego, California, February 2003.

[36] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM Press, 2003.

[37] Y. Xie and D. Engler. Using redundancies to find errors. *ACM SIGSOFT Software Engineering Notes*, 27(6):51–60, 2002.

[38] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO'04)*, 2004.

[39] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iwatcher: Efficient architectural support for software debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 224. IEEE Computer Society, 2004.