Practical Automated Vulnerability Monitoring Using Program State Invariants

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

Lorenzo Cavallaro
Royal Holloway, University of London
lorenzo.cavallaro@rhul.ac.uk

Andrew S. Tanenbaum
Vrije Universiteit Amsterdam
ast@cs.vu.nl

***Abstract*—Despite the growing attention to security concerns and advances in code verification tools, many memory errors still escape testing and plague production applications with security vulnerabilities. We present RCORE, an efficient dynamic program monitoring infrastructure to perform automated security vulnerability monitoring. Our approach is to perform extensive static analysis at compile time to automatically index *program state invariants* (PSIs). At runtime, our novel dynamic analysis continuously inspects the program state and produces a report when PSI violations are found. Our technique retrofits existing applications and is designed for both offline and production runs. To avoid slowing down production applications, we can perform our dynamic analysis on idle cores to detect suspicious behavior in the background. The alerts raised by our analysis are symptoms of memory corruption or other—potentially exploitable—dangerous behavior. Our experimental evaluation confirms that RCORE can report on several classes of vulnerabilities with very low overhead.**

***Keywords*-Program State Invariants, Vulnerability Analysis, Memory Errors, Systems Security**

I. Introduction

Memory errors represent a major source of security vulnerabilities for widely deployed programs written in type-unsafe languages like C. According to the NIST's National Vulnerability Database [1], 662 memory error vulnerabilities were published in 2011 and 724 in 2012. While software engineers strive to identify memory errors and other vulnerabilities as part of the development process, dynamic vulnerability monitoring and identification in production runs is a compelling option for two important reasons.

First, many security vulnerabilities escape software testing and are only later discovered in production applications at a steady rate every year [2]. This is due to the limited power of code analysis tools and the inability to test all the possible execution scenarios effectively in offline runs. Given the large-scale deployment of today's production applications, it is no wonder that the testing surface can increase drastically in production runs, with different installations subject to very different environments and workloads. This gives a much better chance for zero-day vulnerabilities to emerge.

In addition, experience suggests that the number of unpatched vulnerabilities is substantial every year. A recent study [2] has shown that only 53% of the vulnerabilities disclosed in 2012 were patched by the end of the year. When prioritizing the known security vulnerabilities to go after becomes a necessity, a dynamic vulnerability monitoring infrastructure can provide a useful feedback to analyze the impact of those vulnerabilities in production.

Unfortunately, state-of-the-art solutions designed to detect and protect against different classes of memory errors [3]–[11] are not well-suited to be used for comprehensive vulnerability monitoring in production runs. Despite significant effort, they still incur substantial overhead and often fail to provide any meaningful feedback.

This paper presents RCORE, a new dynamic program monitoring infrastructure that continuously inspects the state of a running program and provides informative feedback about generic memory errors and potential security vulnerabilities. Our solution barely impacts running applications and retrofits existing programs and already deployed shared libraries. Our low-overhead design can completely decouple the execution of a target program and the execution of the monitor, isolating the monitoring thread on a separate core.

To detect many classes of memory errors, our approach builds on a combination of static and dynamic analysis. Static analysis is performed at compile time to embed in the final binary all the *program state invariants*, which specify inviolable safety constraints for the different components of the program state (i.e., objects, types, values). The key idea is to detect memory errors and suspicious behavior from run-time violations of the prerecorded invariants maintained in memory. To accomplish this task, our run-time monitor continuously inspects the program state in the background and reports every violation found, along with all the necessary information to track down the original problem. Our analysis is concerned with security and not space overhead, given that RAM is hardly a scarce resource nowadays.

To achieve the lowest possible overhead—at the cost of reduced precision—in production runs, our default invariants analysis strategy is fully asynchronous. This approach results in a probabilistic detection model specifically designed to detect forms of global state corruption. This category covers a significant fraction of emerging vulnerabilities, which produce latent errors or silent data corruption and may normally go undetected [12]. This trend is reflected in a growing number of exploits moving from stack-based attacks to data or heap-based attacks [13]. Our ultimate goal is to build an automated security vulnerability reporting service, similar, in spirit, to widely used remote crash reporting tools.

The contribution of this paper is threefold. First, we introduce a novel program state invariants analysis which is used as basis for our detection technique. Second, we show that our invariants analysis can be effectively used to infer both suspicious behavior that can cause memory errors and memory corruption caused by a memory error, even when the root cause is unknown. Our analysis covers the entire global program state (i.e., data, heap, and memory-mapped regions) and can detect a broad class of memory errors, including buffer overflows, dangling pointers, double or invalid frees, and uninitialized reads. Compared to existing techniques, we support detection of memory corruption caused by the libraries, application-specific memory management, and memory errors that do not spread across data structure boundaries (e.g., buffer overflow *inside* a struct). Third, we have developed a system, termed RCORE, which can reuse dedicated spare cores to perform our invariants analysis on a running program in real time. Our prototype demonstrates that our analysis can be efficiently parallelized and used in practical vulnerability monitoring scenarios. To the best of our knowledge, we are the first to support such a fine-grained vulnerability analysis and show that it can be performed in real time with very low overhead.

II. Program State Invariants

Program state invariants (PSIs) represent global safety constraints that restrain the run-time behavior of every state element in the program in an *execution-agnostic* fashion. We use the term state element (or *s-element*) to refer to typed memory objects (i.e., variables or dynamically allocated objects) and their recursively defined members (e.g., `struct` fields or array elements) indiscriminately. We consider PSIs for both pointer and nonpointer *s-elements* for programs written in type-unsafe languages like C.

Our current system supports three types of PSIs: *value-based* PSIs, *target-based* PSIs, and *type-based* PSIs. Value-based PSIs restrict the set of legal values for both pointer and nonpointer *s-elements*. Target-based PSIs specify the set of valid targets a pointer *s-element* can point to (i.e., a pointer must point to a valid *s-element* in memory). Finally, type-based PSIs restrict the set of legal types a pointer *s-element* can point to (e.g., a function pointer must point to a valid function *s-element* with a matching type).

Unlike other invariant-based techniques [14]–[20] or more general learning-based techniques [21], [22] that aim to automatically detect anomalous behavior, our invariants are execution-agnostic and solely determined from static analysis. While this strategy might miss some valid invariants that can only be determined by fine-grained dynamic monitoring, our approach eliminates the coverage problems that arise when learning invariants at runtime and results in a more conservative invariants analysis, ruling out false alarms at detection time—other techniques incorrectly raise an alert whenever a program element reports a legitimate value never observed in the training phase. In particular, when not using proactive detection of suspicious behavior like long-lived dangling or off-by-N pointers, RCORE's conservative analysis squarely meets the goal of zero false positives.

Another advantage of using static compile-time information to learn properties of the program behavior is the ability to derive restrictive and fine-grained PSIs. This immediately suggests that run-time PSI violations can be used as an accurate predictor for memory errors. The key intuition is that, when some form of arbitrary state corruption occurs, the probability of no PSI being violated is low. This is true independently of the particular memory error that caused the corruption. For example, when a global data pointer is corrupted with arbitrary data, the chance that the pointer is still pointing to an object of a valid type (as determined by static analysis) is negligible—equivalent to the probability of randomly guessing the address of a valid memory object. Similarly, a global function pointer corrupted with arbitrary data by a memory error is unlikely to still point to a valid function with a valid type (as determined by static analysis). As a result, both scenarios will allow RCORE to detect a target- or type-based PSI violation with high probability for the corrupted pointer *s-elements* and immediately generate a report on the memory errors found.

Finally, the violation patterns reported can be used to effectively classify the memory errors detected (e.g., contiguous *s-elements* with PSI violations in a buffer overflow).

III. Architecture

Our architecture comprises 5 main components: compiler driver, static instrumentation component, metadata framework, dynamic instrumentation component, and run-time analyzer. The compiler driver is designed to support static analysis and instrumentation of existing applications and libraries, integrating seamlessly with existing build systems.

The static instrumentation component—implemented on top of the LLVM compiler framework [23]—inspects the program to identify all the *s-elements* and PSIs at compile time and instruments the final binary to store the corresponding metadata. The latter are used to validate the program state against all the prerecorded invariants at runtime. For this purpose, the metadata framework provides an API to query and manage statically and dynamically created metadata. The framework is transparently linked to the program by the compiler driver during the build process.

The dynamic instrumentation component creates and destroys dynamic metadata to support uninstrumented shared libraries. This step is necessary to perform a conservative target-based analysis and avoid spurious alerts. In contrast, using only the dynamic instrumentation component without any static instrumentation as done in state-of-the-art memory allocators [13], [24], would degrade the effectiveness of our analysis, as explained later. The run-time analyzer is responsible for monitoring the behavior of the program in

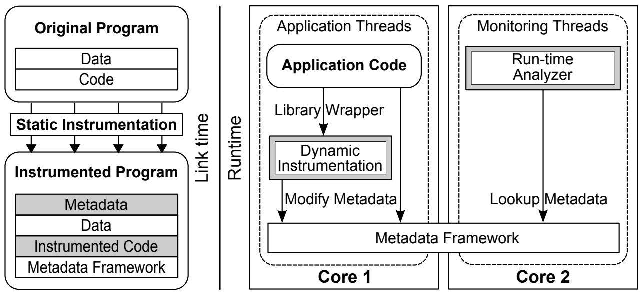


Figure 1. RCORE architecture.

real time and detecting PSI violations. At each monitoring cycle, the analyzer uses the metadata framework to introspect the program state and check annotated PSIs for each *s-element* found. Figure 1 depicts RCORE's architecture and the interactions between the components at runtime.

A. Static Instrumentation

The static instrumentation component is an LLVM link-time pass that analyzes and instruments the program and statically linked libraries to embed state metadata into the resulting LLVM bitcode. The latter is then processed by a valid LLVM back end to produce the final binary.

Our static analysis starts by extracting relocation and type information on variables and functions used in the program. These objects will become part of the embedded metadata that index all the *s-elements* and the corresponding PSIs.

To index types, we extract all the relevant type information available using the LLVM API. The language- and architecture-independent LLVM type hierarchy found in the original bitcode is extracted and then stored directly into the program using a convenient format. Our data structures use a compact tree-like representation with leaf nodes representing primitive types. For example, similar to the original LLVM type system, an array of 10 `float*` pointers is represented by 3 distinct type nodes linked together: array (`[10 x float*]`), pointer (`float*`), and primitive (`float`), respectively. Only types referring to state entities are made part of the type hierarchy.

Global variables extracted from the program represent the first important state entity. Metadata information about global variables is stored in a number of state entries (or *s-entries*) made available at runtime. Each *s-entry* contains the name, type, and address of the variable. In addition, flags are used to mark constant variables and variables that have their address taken. Size and padding information for aggregate variables are computed at runtime (for portability reasons) and stored directly in the type hierarchy. This is important to efficiently support target-based and type-based PSIs.

A similar approach is adopted for functions, but only functions that have their address taken are made part of the embedded metadata information. This is possible since functions are only used to enforce target-based and type-based PSIs and not to introspect the state of the program.

Indexing pointer casts. To enforce type-based PSIs for pointer *s-elements* we need knowledge of pointer types that are allowed for each *s-element* at runtime. Unfortunately, type-unsafe languages like C allow for arbitrary casts between different types and recording metadata on static pointer types is not sufficient for a conservative analysis. To address this problem, our static analysis extracts all the pointer casts from the original program and enriches the type hierarchy with *casts-to* links between compatible type nodes. Luckily, LLVM explicitly represents both implicit and explicit pointer casts in the intermediate representation. The `bitcast` instruction is used to represent pointer-to-pointer casts, while the `inttoptr` and the `ptrtoint` instructions are used to handle integer-to-pointer casts and vice versa.

Indexing value sets. To enforce value-based PSIs it is necessary to store metadata for the set of legal values allowed for a given *s-element* at runtime. When possible, our static instrumentation annotates each type of the type hierarchy with the set of the legal values allowed. The value set is directly stored in a type node to simplify sharing and *s-element*-level metadata management. Our current value analysis module supports basic value-set analysis (VSA) and can annotate both pointer and nonpointer *s-elements*. For pointer *s-elements*, we analyze `inttoptr` instructions and attempt to determine the set of all the legal integer values. When our conservative analysis fails, no value set is recorded but the pointer is marked as an *integer candidate*. Our simple analysis is, however, very often successful in real-world scenarios, where integer values usually refer to some special predefined constants (e.g., `SIG_DFL` and `SIG_IGN` defined for the POSIX `sigaction` system call). For nonpointer *s-elements*, our current module records metadata for enums, constants, and variables assigned to constant expressions, but it would be straightforward to incorporate more sophisticated value analyses (e.g. range analysis).

Memory management instrumentation. To index dynamically allocated memory objects, we replace all the memory management functions in the program with our own wrappers to create and destroy metadata at runtime. Our current implementation supports all the POSIX `malloc`-like and `free`-like functions, including the `mmap` family, the `mem_align` family, and shared memory functions. Each memory allocation wrapper takes, along with the original arguments, an additional parameter that describes the run-time type of the to-be-created dynamic state entry (or *ds-entry*). Each *ds-entry* provides metadata for a dynamically allocated memory object represented as a typed array and contains similar information to the one included in a *s-entry*.

To determine the run-time type of a *ds-entry* correctly, we devised a conservative type inference algorithm for our static instrumentation component. Our algorithm recursively walks through all the possible uses of the value returned by each memory allocation function and considers all the global and local variables in the caller to which the value can be possibly assigned. When the run-time type can be determined unambiguously, the type node and all the relevant casts encountered are added to the type hierarchy. This approach provides the ability to introspect dynamically allocated objects at the finest level of granularity possible at runtime, while dealing with the ambiguous cases in a conservative way by indexing all the pointer casts encountered.

Our type inference algorithm can also automatically recognize and handle application-specific memory allocation wrappers (e.g., `my_malloc`) with arbitrary levels of nesting by analyzing the interprocedural propagation of weak pointer types (i.e., `void*`). In these cases, the algorithm is repeated recursively and the final run-time type propagated throughout all the wrappers encountered. When the type cannot be correctly determined, the instrumentation component resorts to the special `void` type used to describe a block of untyped memory, which, however, hampers the ability to introspect the memory object at runtime. In our current prototype, this scenario can occur in practice with programs relying on region-based memory management implementations [25]. To deal with such schemes and improve the coverage of our runtime analysis, our static instrumentation can be instructed to locate and automatically construct typed wrappers for region-based memory allocation/deallocation functions.

B. Metadata Framework

The metadata framework provides the data structures for all the metadata entities (i.e., *types*, *functions*, *s-entries*, and *ds-entries*) and the API to manage them at runtime. The metadata API provided by the framework offers 3 primary functionalities. First, API functions are available to introspect the entire program state. For example, a callback-based mechanism is used to process *s-entries* and *ds-entries* and conveniently operate on all the *s-elements* found therein. Second, a lookup API is available to locate any metadata entity given an appropriate search key. For example, the points-to lookup API—used to check target-based PSIs—locates a target *s-entry* given a valid pointer *s-element*. Finally, the framework provides an API to create and destroy metadata at runtime. This is used in the predefined memory management wrappers included in the framework.

To achieve good performance, our wrappers store the *ds-entries* using in-band metadata. Similar to prior approaches, we use canaries [26] to detect metadata corruption due to overflows, and optionally flip the top bit of every metadata word before and after use as suggested in [24] to mitigate dangling pointers. If necessary, strategies adopted by out-of-band memory allocators [13], [24] can be incorporated in our implementation to offer additional protection at the cost of more overhead. Note that the metadata canaries are continuously checked for consistency by the run-time analyzer. This eliminates the need to determine application points to check the canaries, which hampered the effectiveness of this approach in prior work, as evidenced in [13].

To achieve loose synchronization between memory wrappers executed in the application context and run-time analyzers executed on a separate monitoring thread, our design provides a lock-free interface for metadata management operations. This avoids any lock contention overhead and guarantees a scalable implementation. Our design arranges the *ds-entries* in a singly-linked list with newly-created *ds-entries* always added to the top. With the top of the list maintained stable, this strategy offers a lock-free stack interface to the memory allocation wrappers. A push-only lock-free stack can be implemented very efficiently using compare-and-swap (CAS) primitives, avoiding extra implementation complexities to ensure scalability in face of significant push and pop contention for the top of the stack or to deal with the “ABA problem" [27]. To maintain the top of the *ds-entry* list stable, the application threads are never allowed to perform a pop operation on the stack. This is accomplished by marking the state of the corresponding *ds-entry* as “*dead*” in the memory deallocation wrappers (e.g. `free`) without actually removing the *ds-entry* from the list. This allows metadata destroy operations to be completely lock-free, using lockless *ds-entry* state updates. The monitoring thread, in turn, periodically deallocates all the dead *ds-entries* and the corresponding data. To maintain the top of the list stable, a removal operation on the top is always deferred until the next *ds-entry* is pushed onto the stack. Multiple monitoring threads can concurrently operate and check PSIs on the same *ds-entry* list using lock-based synchronization (not exposed to the application threads). Note that this does not interfere with our memory wrappers and introduces no additional lock-contention overhead for the application threads.

C. Dynamic Instrumentation

The dynamic instrumentation component is an interposition library responsible for indexing deployed uninstrumented libraries at runtime to avoid spurious alerts in our invariants analysis. The component provides three key functionalities. First, the component creates *ds-entries* for dynamically linked libraries at program initialization time. This step is necessary to enforce target-based invariants in case of application pointers pointing to text or data regions created by the dynamic linker. To address this problem, we rely on the same data structures used by the dynamic linker to introspect the address space of the program and locate all the dynamic objects that belong to uninstrumented libraries.

Our current implementation is based on the ELF binary format. To introspect dynamic objects, it is sufficient to parse the ELF header to locate the GOT and the `link_map` data structure used by the linker. For each text region found, we extract all the symbols and create a single untyped *ds-entry* (i.e., a *ds-entry* with the special type `void`) for each library function. Note that we need to index all the uninstrumented library functions, since the knowledge of whether a function can have its address taken is irretrievably lost, and so are the original symbol types. For each data region found, in turn, we create a single untyped *ds-entry* describing the region.

The second important responsibility of the dynamic instrumentation component is to keep track of dynamically loaded shared libraries at runtime and create and destroy the corresponding *ds-entries* when necessary. For this purpose, the interposition library includes wrappers for the programming interface to the dynamic linking loader provided by POSIX. We use a `dlopen` wrapper to create or update *ds-entries*—POSIX defines a reference counter to reuse existing library mappings—and a `dlclose` wrapper to destroy existing *ds-entries* when the reference counter drops to zero.

For dynamically loaded libraries, our wrappers follow the same approach adopted to index dynamically linked libraries. New *ds-entries* are similarly created for the new memory regions, and each *ds-entry* is marked as text or data depending on the particular region type considered. Finally, the dynamic instrumentation component needs to handle all the memory management functions invoked at runtime from uninstrumented libraries. For this purpose, the interposition library includes dynamic wrappers for all the memory management functions supported by the metadata framework and redirects execution to the original static wrappers accordingly. Since the type information is lost for uninstrumented libraries, the run-time type provided to the original wrappers is always `void`. This explains why it is crucial to combine static and dynamic instrumentation to handle memory management functions. Dynamic instrumentation is necessary to create metadata for all the possible dynamic memory objects and avoid proliferation of spurious alerts. At the same time, dynamic instrumentation alone would produce only untyped *ds-entries*, making it harder to reason about dynamically allocated memory blocks, a common problem in prior work [24]. In our approach, untyped memory objects hamper state introspection—as for example expected for uninstrumented libraries—and decrease the accuracy of our target-based and type-based PSIs.

D. Run-time Analyzer

The run-time analyzer is responsible for sampling the program state periodically and checking PSIs to detect any violation. The initialization code prepares all the data structures used in the analysis and transparently allocates the monitoring thread on a predefined core. Depending on the configuration given, it is possible to allocate multiple monitoring threads on the same or different cores to increase the frequency PSIs are checked. In the other direction, it is also possible to reduce the frequency of monitoring cycles to reduce CPU utilization. This allows us to trade off security and CPU utilization when power consumption is of concern. If strict backward compatibility is not required, the application could also be slightly modified to start the analysis only in face of particular events, saving monitoring cycles when the application is idle. The analyzer runs the monitoring thread in an endless loop, although the analysis can be interrupted and restarted on demand, if necessary. At each cycle, the program state is sampled to check PSIs. The analyzer cycle comprises 5 (not necessarily sequential) phases: state introspection, invariants analysis, recording, reporting, and feedback generation.

State introspection. The analyzer locates all the indexed *s-entries* (and *ds-entries*) and recursively walks through all the *s-elements* found using the functions provided by the metadata framework. All the *s-elements* that have candidate PSIs are considered for analysis. Our default strategy analyzes all the relevant *s-elements* sequentially but, depending on the threat model considered, additional policies can be used to prioritize particular state regions (e.g., heap) and check corresponding PSIs at a higher frequency.

Invariants analysis. The analysis is carried out for all the PSIs supported for any given *s-element*. First, the value of the *s-element* is atomically read and checked for value-based PSIs whenever a value set is available. If the value is not part of the value set, a violation is flagged.

We also analyze pointer *s-elements* that have been marked as *integer candidates* with no value set provided. In particular, if the pointer points anywhere in the set of reserved pages at the beginning or at the end of the address space, our analysis marks the pointer as safe. This strategy reflects the knowledge that pointers marked as *integer candidates* are typically assigned to special constants that do not reflect valid memory addresses. Although some corrupted pointer in this category may go undetected, when the pointer is dereferenced a fault will immediately be triggered. If an *integer candidate* points to an address outside the reserved range, the pointer is promoted to a regular pointer and further PSIs are normally checked for violations.

For all the pointer *s-elements* considered for further analysis, target-based PSIs are checked next. The metadata API is used to look up the *s-entry* or *ds-entry* each *s-element* points to. If no valid entry can be found or the entry refers to an object that does not have its address taken, a violation is flagged. Upon successful lookup, the target entry is recursively analyzed to determine the run-time type or types the pointer is pointing to. If the analysis fails, for example the pointer is illegitimately pointing to padding data of a `struct`, a violation is flagged. When valid target types are found, type-based PSIs are considered.

For type-based PSIs, we first check the static pointer type and determine whether it matches any of the run-time types the pointer is pointing to. When no match with the static pointer type is found, the analyzer examines the set of compatible pointer types retrieved from our *linked* type hierarchy. If no match is found, a PSI violation is flagged. When the target *s-entry* is untyped, the nature of the target is considered. If the original pointer is a data pointer and points to a *s-entry* referring to a text memory region (or vice versa), a violation is flagged.

Recording. The results of our analysis are recorded to collect fine-grained statistics on each *s-element* with annotated PSIs. For each *s-element*, we record the PSI violations found. For pointers, we also record the distribution of target types found (with the number of occurrences sampled for each type) and the corresponding memory regions (i.e., data, heap, mmap, shared memory, text).

Reporting. To be effective in different scenarios, RCORE supports policy-based detection mechanisms. Policies decide what events indicate suspicious behavior and need to be reported. RCORE supports two default detection mechanisms: *synchronous* detection and *window-based* detection. The synchronous detection mechanism simply logs all the PSI violations found. While useful in development mode, this mode of operation is not always desirable in production. Some short-lived PSI violations may be sometimes acceptable and expected in the normal execution of the program. For example, consider a pointer that is freed and then immediately set to `NULL`. The asynchronous analysis performed by the monitoring thread might sample the pointer value right after the `free` call. In this case, a dangling pointer would be immediately reported as dangerous although the pointer is dangling only for a very short period of time and never used. To address this issue and reduce the number of alerts in production, RCORE defaults to another (window-based) detection mechanism for dangling pointers. In this mode of operation a sliding window is used to collect a number of state samples over a time interval. The resulting distribution is used to enforce detection policies and log suspicious events on a per *s-element* basis. The size of the window is configurable, and so are the policies supported for each particular event. The default policy is to report only the PSI violations that occur for all the samples in a single detection window, but more sophisticated policies are possible. This simple policy is effective in real-world scenarios, allowing one to tune the number of alerts logged by simply varying the window size. Reasonably short detection windows avoid logging common dangling pointer violations and provide accurate detection for the suspicious cases.

Feedback generation. For each logged event, we generate accurate information on the PSIs violated and report all the statistics gathered on a per *s-element* basis. The detailed information provided in the feedback can help developers track and reproduce the original problem for debugging purposes. In addition, the feedback can be used to automatically classify the violations basing on the patterns observed. For example, a common pattern we have observed for the distribution of target types of a dangling pointer is `NULL`, `type x`, `target-based PSI violation`.

Debugging. RCORE includes a flexible debugging interface to support offline analysis of all the PSI violations found. Our asynchronous detection strategy provides a debug-friendly environment for offline runs, with the runtime analyzer imposing minimal disruption on the application threads and the static instrumentation preserving symbol table and stack information. To quickly locate and fix the original problem, developers can use the debugging interface to set arbitrary breakpoints and interrupt the program execution upon specific PSI violations (e.g., break on any PSI violation found for the pointer `my_ptr`). Debugging support reflects our goal of simplifying the entire vulnerability discovery and patch development-deployment lifecycle.

IV. Memory Errors Detected

Dangling pointers. RCORE supports proactive detection of memory errors derived from dangling pointers, which are explicitly recognized using PSI violations and knowledge of known heap regions. Dangling pointers can be detected immediately, even before they are actually dereferenced. This is crucial for a dynamic vulnerability monitoring infrastructure. Common are cases where even extensive dynamic analysis fails to trigger corruption caused by a vulnerable dangling pointer. For instance, the pointer may be dereferenced only in code paths that are rarely triggered during normal execution. For this reason, RCORE uses window-based detection to report all the suspicious long-lived dangling pointers, which can then be further inspected offline. In addition, arbitrary corruption caused by incorrect use of dangling pointers can be detected by PSI violations on the corrupted target region.

Off-by-one pointers. RCORE supports proactive detection of off-by-one pointers, which are often legitimately used to mark buffer boundaries. If incorrectly used, however, they can introduce overflows or indirectly cause other memory errors. RCORE's target-based analysis can explicitly recognize generic off-by-N pointers even before they can do any harm. A policy determines whether our analysis should report (immediately or using window-based detection) or ignore these cases. Similar to dangling pointers, memory corruption caused by incorrect use of these pointers is still always reported by our invariants analysis.

Overflows/underflows. RCORE supports detection of buffer overflows (and underflows) using invariants analysis to detect the resulting memory corruption occurred. Note that, in contrast to existing source-level approaches, our fine-grained analysis allows RCORE to detect arbitrary overflows, even those for buffers inside a `struct` or buffers allocated using application-specific dynamic memory allocation. In most cases, it is very easy to classify a suspicious event as a buffer overflow or underflow, depending on the patterns observed. This is reflected by a number of PSI violations reported for contiguously allocated *s-elements*.

Double and invalid frees. Like other common memory allocators, RCORE detects most double and invalid frees directly in the memory management wrappers, using in-band metadata canaries. In the remaining cases, arbitrary memory corruption caused by the illegal operation can still be detected by PSI violations on the corrupted region.

Uninitialized reads. RCORE supports probabilistic detection of uninitialized reads. Our default strategy is to start checking PSIs for *s-elements* described by a given *ds-entry* as soon as the *ds-entry* is created. Dynamically allocated *s-elements*, however, may not have been initialized yet when the analysis starts and the random garbage contained therein would likely trigger PSI violations. To address this issue, we allow a configurable grace period before introspecting new *ds-entries* in the analysis. This strategy follows the intuition that new *s-elements* that are left uninitialized for too long increase the probability of uninitialized reads and should therefore be considered for offline inspection. This strategy is effective even for reasonably short grace periods. As in other cases, memory corruption indirectly caused by uninitialized reads—for example dereferencing an uninitialized pointer and write data to an arbitrary memory region—can be detected by PSI violations on the corrupted region.

V. Evaluation

The current RCORE implementation runs on Linux, but most components can be easily ported to other operating systems and binary formats other than ELF. Our compiler driver is implemented in python in 1200 lines of code (LOC). The static instrumentation component is implemented as an LLVM pass in 6000 LOC, and supports all the standard LLVM optimizations. The metadata framework is implemented as a static library written in C in 3700 LOC. The dynamic instrumentation component and the run-time analyzer are implemented as shared libraries written in C in 800 LOC and 2300 LOC, respectively. The libraries are preloaded using platform-specific support offered by the dynamic linker (e.g., the `LD_PRELOAD` UNIX environment variable) to override the default run-time program behavior.

A. Performance

We evaluated the overhead of our solution using the C programs in the SPEC CPU2006 benchmarks. We ran our experiments on a Dell Precision workstation with two 2.27GHz Intel Xeon E5520 quad-core processors and 4GB of RAM running a 2.6.35 Linux kernel. Each core has two hyper-threads sharing the L1 and L2 cache, whereas the four cores on the same die share an 8-MB L3 cache.

We executed each experiment 11 times and reported the median. We evaluated both the overhead introduced by our static and dynamic instrumentation and the one introduced by these components and the RCORE run-time analyzer allocated on dedicated cores. Figure 2 shows the execution time of the RCORE version of our benchmarks normalized against the baseline.

The static and dynamic instrumentation components of RCORE introduce 3% run-time overhead on average (geometric mean). The whole framework in its default configuration (one run-time analyzer) introduces 8% overhead on average. The average, however, is heavily influenced by benchmarks like `perlbench` due to the massive use of dynamic memory allocations, which inevitably results in high memory management instrumentation overhead and significantly increased contention for memory bandwidth between application and monitoring threads. Encouragingly, for the majority of the benchmarks RCORE introduces a negligible overhead with a median value of only 1.5%.

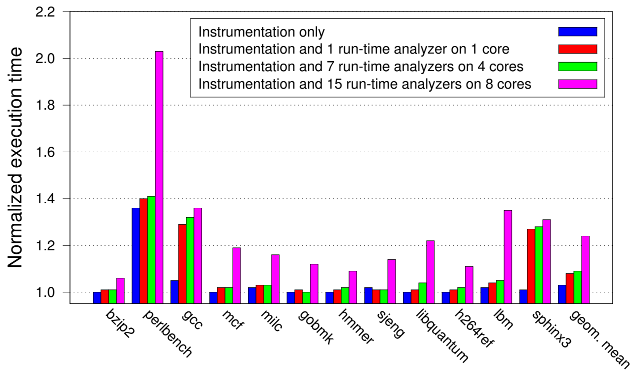
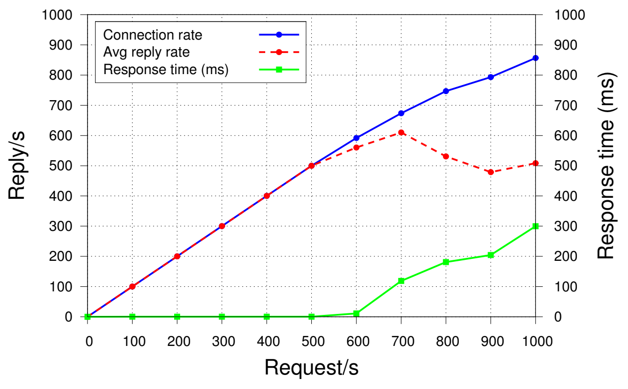


Figure 2. Run-time performance overhead introduced by RCORE for the SPEC CPU2006 benchmarks.

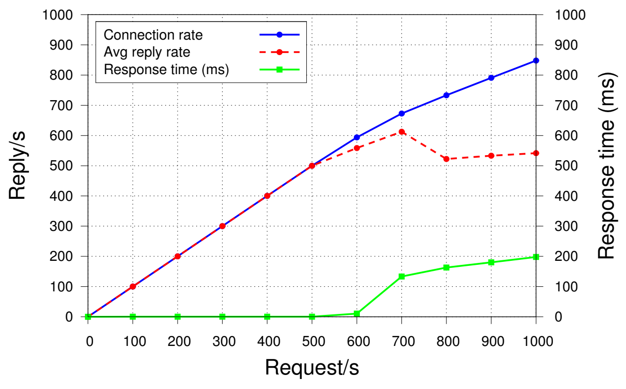
The last two bars in each benchmark show the overhead imposed by RCORE when configured with 7 and 15 run-time analyzers assigned to 4 and 8 independent cores with hyper-threading (9% and 24% on average, respectively). The significantly higher overhead introduced in the latter scenario acknowledges the impact of the increased contention for memory bandwidth caused by multiple monitoring threads scheduled on different dies with no cache shared. Our results confirm the importance of a shared cache to achieve good performance in concurrent dynamic monitoring applications, as also recognized in prior work [28].

We now compare our SPEC results with WIT [29] and Cruiser [30], two recent low-overhead solutions to detect memory errors. RCORE reports lower overheads than WIT on average, which shows an average overhead of 10% on SPEC CPU2000 benchmarks. WIT's object-level analysis for memory writes is also more coarse-grained than ours, although WIT's run-time checks follow the main application flow and are thus less probabilistic than RCORE's asynchronous detection model. RCORE reports lower average overhead than Cruiser, which shows an average overhead of 12.5% on SPEC CPU2006 Integer benchmarks in its lazy version. The overhead drops to 5% for Eager Cruiser, which, however, requires recovery techniques and may incur false positives. Cruiser's detection model is asynchronous like ours, but focuses only on heap-based buffer overflows.

Our second set of experiments evaluated the throughput and latency degradations introduced by RCORE on `nginx` [31] (version 0.8.54) and `lighttpd` [32] (version 1.4.28), two popular web servers. The web servers were independently deployed on the same Dell Precision work-



(a)



(b)

Figure 3. `nginx` benchmark: (a) Uninstrumented and (b) RCORE version.

station used above. A number of clients were deployed on different Dell workstations with a 3.33GHz Intel Core 2 Duo CPU and 4GB of RAM, each running a 32-bit 2.6.35 Linux kernel and connected to the servers through a Gbit link.

Figure 3 shows the average throughput and latency of `nginx` under different workloads (i.e., requests per second) while retrieving a 50KB file. In particular, we started with a rate of 100 req/s up to 1000 req/s, increasing the request rate by 100 req/s on each subsequent run. Each request opened a connection to download the requested file. Each run lasted for at least 75 seconds and issued as many connections as needed to match the request rate for the duration of the whole run considered. This allowed `httperf` [33], our web benchmarking tool, to collect enough evidence (i.e., samples) needed to produce statistically sound results. Figure 3(a) refers to tests performed on an unmodified version of `nginx` and represents our baseline. Figure 3(b), in contrast, refers to tests performed on the RCORE (1 run-time analyzer on a dedicated core) version of `nginx`. Figure 3(a) shows that the baseline achieved the maximum throughput at around 500 requests per second, at which time `httperf` issued 37500 connections in total. As we further increased the request rate, the server became saturated, as shown by the gradual throughput and latency degradation. We have checked that the server was the bottleneck by performing a number of additional experiments to verify that all the clients could keep up with that maximum request rate. Similarly, Figure 3(b) shows that the RCORE version of `nginx` was able to match the same maximum rate and response time of the baseline, introducing only a negligible overhead. We have performed the same set of experiments on `lighttpd` and under different workload scenarios and obtained similar results with negligible overhead. We omit the figures of such experiments due to lack of space. The overall results here outlined are encouraging and show that our approach introduces negligible overhead on the end performance of the RCORE version of the program operating at full capacity. This enables a practical and realistic deployment of our solution in production systems.

B. *Detection Accuracy*

Decoupling security checks from the main application flow guarantees low overhead but inevitably introduces a latency in the detection of PSI violations. The latency depends on the monitoring frequency and the number of run-time analyzers and cores used. Lower detection latencies are desirable for better accuracy. Higher detection latencies, on the other hand, reduce power consumption. The appropriate tradeoff can be tuned for each particular scenario considered.

Figure 4 depicts the monitoring cycle time achieved by RCORE and the resulting number of PSI violations detected when injecting into the program 100 memory corruptions with a lifetime uniformly distributed in [1, 200]ms. The values are plotted as a function of the overall CPU utilization allowed to the run-time analyzers, with one or more dedicated hyper-threads hosting a single analyzer each. At 100% CPU utilization, the default configuration (1 run-time analyzer on 1 thread) completes a monitoring cycle and checks the PSIs of a given *s-element* every 24ms. Conversely, the time elapsed between any two checks with 1 run-time analyzer at 20% CPU utilization is around 110ms. The configuration with 7 analyzers allocated on 7 threads sharing the L3 cache (700% CPU utilization) with the application achieves the lowest detection latency of 10ms. The resulting percentage of PSI violations detected in these 3 configurations is 94%, 72%, and 98%, respectively. These results have been obtained for `nginx`, but similar behavior can be observed for other programs, with the detection latency dependent on the number and the complexity of the data structures used. When compared to the simpler canary-based detection strategy used in Cruiser [30], our analysis incurs higher detection latencies, but encompasses many more memory errors than only heap-based buffer overflows. To further improve RCORE detection latency, we can increase the number of monitoring threads or instruct the run-time analyzers to focus on particular portions of the program state, for example, on those that are observed to change more often.

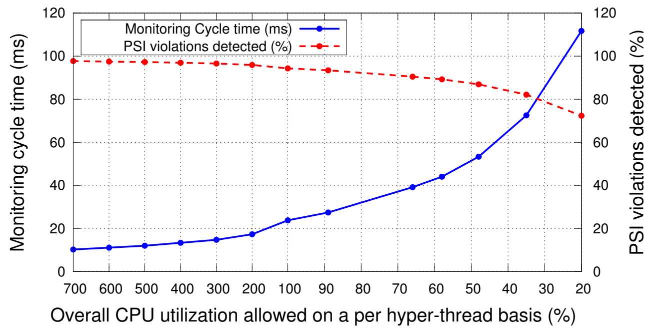


Figure 4. Monitoring cycle time and PSI violations detected in `nginx` for decreasing overall CPU utilization.

C. *Effectiveness*

To evaluate the effectiveness of RCORE in detecting memory errors using our invariants analysis, we performed two complementary experiments: (1) a feedback evaluation, which measured the accuracy achieved by RCORE during testing runs of `proftpd` version 1.3.3e and `exim` version 4.69, two well-known FTP and SMTP servers; (2) a CVE (Common Vulnerabilities and Exposures) evaluation, which assessed the ability of RCORE to detect representative memory error vulnerabilities related to `nginx` and `openssl`. Our ultimate goal is to evaluate RCORE's effectiveness at detecting real-world memory errors as PSI violations and providing useful feedback to pinpoint the original problem.

The feedback evaluation performed on `exim` allowed us to find a previously unknown, potentially exploitable, vulnerability. In particular, the vulnerability is represented by an out-of-bounds pointer, `mainlog_datestamp` (`log.c`), that is dereferenced for reading. This may potentially lead to a denial of service situation, such as, process crash or file resource exhaustion. Conversely, `proftpd`'s feedback evaluation reported an interesting long-lived dangling pointer, `capabilities` (`mod_cap.c`). Even a careful code inspection was insufficient to assess whether this dangling pointer could lead to a vulnerability, but the code should probably be better restructured to avoid problems.

We then proceeded to assess whether RCORE is able to detect real-world vulnerabilities. To this end, we selected the CVE advisory 2009-2629, which describes a buffer underflow vulnerability that affects several versions of `nginx`, including those from 0.6.x before 0.6.39, among others (our analysis was performed on `nginx` version 0.6.38), and the CVE advisory 2010-2939, which describes a double-free vulnerability affecting `openssl` version 1.0.0a.

The CVE 2009-2629 advisory states that a specially crafted HTTP request may produce memory corruption enabling the execution of arbitrary code. We selected this CVE because it was a particularly representative case of global state corruption introduced by a typical memory underflow vulnerability. In addition, `nginx` relies heavily on application-specific memory management and uses many `struct` types with buffer variables; all elements that make the detection of memory corruption hard in the general case. The execution flow that triggers the vulnerability starts with `nginx` invoking `ngx_http_init_request()` when processing network input. This function allocates a 1024-byte pool using application-specific memory allocation functions and fills the pool with a number of data structures containing several nested pointers and the parsed input $\mathcal{I}$ at the end (e.g., `ngx_table_elt_t`). The underflow causes a temporary pointer, initially pointing to $\mathcal{I}$, to traverse back up to the next '/' character encountered (`ngx_http_parse_complex_uri`). Depending on the particular memory layout at the moment of the underflow, the temporary pointer may land within the same `struct` that included the original input buffer (e.g., `ngx_str_t`), within the same pool-dedicated block, or on a different memory block. Depending on the input provided, the pointer is then used to write garbage that overrides a number of consecutive *s-elements*. In our multiple experiments with different input distributions, the observed memory corruption repeatedly triggered several PSI violations, given the significant number of pointers corrupted. As a result, RCORE was able to detect the corruption in all our tests, no matter where the temporary pointer initially landed. Existing approaches would have failed to detect the corruption in the general case. We were also positively impressed by the accuracy of our invariants analysis. In particular, our type-based invariants were extremely accurate in detecting type violations for all the corrupted function pointers, given the small fraction of *s-elements* referring to the same given function type.

Conversely, the CVE 2010-2939 advisory states that a specially crafted private key may allow context-dependent attackers to execute arbitrary code due to a double-free vulnerability in the function `ssl3_get_key_exchange` of `openssl` version 1.0.0a. In our experiments, RCORE repeatedly detected the vulnerability in the memory management wrappers using in-band metadata canaries. To simulate the scenario of a new valid memory block overriding the memory location of the original canary with a legal canary value, we disabled all our checks in the memory management wrappers and only checked for PSI violations instead. When the allocator happened to allocate a new memory block in the same memory region as the old block's metadata, the unchecked double free corrupted arbitrary data in the new block. Similarly to what was observed for `nginx`, our experiments in this scenario promptly reported type- and target-based PSI violations on the corrupted data.

VI. Limitations

RCORE is primarily targeted at reporting on known or unknown vulnerabilities during normal in-the-field execution. Due to the probabilistic nature of our asynchronous detection model (crucial to achieve low overhead), however, we can make no claim that RCORE can identify *all* the short-lived vulnerabilities or attacks that affect the global program state. Attacks, in particular, can only be detected under the following conditions. First, memory corruption induced by the attacker must trigger some PSI violations. This is often the case when the attacker cannot make strong assumptions on the layout of the corrupted region. In this scenario, RCORE is very likely to identify PSI violations, especially for pointers corrupted with arbitrary data. On the other hand, even if the attacker can reliably craft a request that produces no PSI violation, his exploitation power is clearly reduced. For example, our PSI analysis would only allow an attacker to corrupt a function pointer with the address of a function of the same type.

Second, the lifetime of the corruption introduced should be no shorter than a monitoring cycle. To evade detection, the attacker may be able to execute arbitrary code shortly after corrupting critical data and quickly perform recovery actions. In our experience, while generally practical for stack smashing and other short-lived attacks, this strategy cannot be taken for granted for attacks that exploit global state corruption. Our claim is also supported by other similar (but less generic) asynchronous detection models which have been successfully applied to heap-spraying [34] and heap-based buffer overflow [30] attacks.

In our future work, we intend to improve our attack detection accuracy by reducing RCORE's monitoring cycle and further investigate the different security-performance tradeoffs, e.g., by switching to a more deterministic "stop-the-world" detection model under particular conditions.

VII. Related Work

Memory errors represent a major category of vulnerabilities and have received much attention in recent years. Bounds checking is a largely explored solution devised to address common memory errors in C programs, but traditional approaches [8]–[10] suffer from significant overhead. More recent bounds checkers have used efficient checks and static analysis [3], [6] to achieve better performance but are still unsuitable for large-scale adoption. More widespread adoption has been gained by StackGuard [35], which uses "canaries" before the return address of a function to detect buffer-overflow errors. Other techniques [36], [37] have used a shadow stack to separate the return address and other sensitive data from buffer variables that are subject to overflow. More recent approaches, in contrast, are specific to heap-based memory errors. Some suggest a particular memory allocator design [13], [24], [38], others use canaries to detect heap-based overflows [26], [39]. ValueGuard [39] instruments the original code to add canaries for both global and local variables. LibsafePlus [40] detects overflows that occur in particular unsafe C library functions (e.g., `strcpy`). This is done by analyzing debugging information and instrumenting the code to describe ranges for local, global, and dynamically allocated buffers. The metadata collected, however, is coarse grained and only used to perform range checking. In a similar direction, MEDS [11] uses a basic low-level type system to perform run-time detection of memory errors, but requires software dynamic translation incurring extremely high overhead. Like ours, other approaches have used the general idea of enforcing static analysis results at runtime. Control-flow integrity [4] computes the program control-flow graph and prevents deviations from it at runtime. Similarly, Castro et al. [7] present an approach to enforce data-flow integrity at runtime using a precomputed data-flow graph.

WIT [29] is a low-overhead solution that uses static analysis to determine the set of objects that can be written by each instruction in the program and instruments the code to enforce write integrity at runtime. Albeit static, their points-to analysis presents similarities with our target-based PSI analysis. Their checks, however, are always performed at the object level and subject to the precision of static analysis to identify accurate object sets. In contrast, our invariants analysis is fine-grained and generalizes their approach with generic program invariants. The approach we propose is more radical. Our static analysis extracts as much information as possible from the program and enforces all the PSIs found at runtime. In addition, WIT cannot support out-of-bound reads without incurring additional overhead.

None of the approaches examined is general and fine-grained enough to support several classes of memory errors, with low overhead. Our PSI analysis can be used to detect arbitrary memory corruption, even when the source of the corruption is unknown. For example, RCORE can also detect hardware memory errors when the resulting corruption leads to PSI violations. Moreover, we support fine-grained analysis of both static and dynamically allocated objects, including introspection of `structs` and objects managed by custom memory allocators. None of the approaches considered can support either. Finally, while RCORE was designed to operate in a fully asynchronous fashion, we believe our invariants analysis can be used in different contexts as a generic state checking mechanism, as also demonstrated by our prior work in the context of live update [41].

We conclude by briefly surveying a number of relevant multicore security applications. He et al. [42] describe dynamic multicore-based program monitoring and compare the performance of their compiler-driven optimizations with instrumentation-based monitoring. Ruwase et al. [43] show how to efficiently parallelize dynamic information flow tracking with several threads running on different cores. Aftandilian et al. [44] propose asynchronous assertions to inexpensively evaluate heavyweight programmer-provided checks concurrently to the execution of the program.

Other approaches [45], [46] explore parallel execution of program variants to detect attacks from divergent behavior. Finally, Cruiser [30] is a low-overhead solution for concurrent heap buffer overflow monitoring. Their work is similar in spirit to ours, but they focus only on a particular class of memory errors using a detection mechanism based on canaries. Our invariants analysis, in contrast, is much more general and targeted toward several classes of memory errors. Their implementation, however, includes a very efficient lock-free dynamic memory allocator that maintains out-of-band metadata, which could also be incorporated in RCORE if using out-of-band metadata is required.

VIII. Conclusion

Current approaches that aim to detect memory error vulnerabilities are either specific to particular categories of memory errors or incur significant overhead, which hinders their widespread adoption in vulnerability monitoring scenarios in production. Despite claiming backward compatibility, existing solutions make also strong assumptions on the nature of the program under analysis, for example that no application-specific memory management is used.

In this paper, we presented RCORE, a low-overhead dynamic program monitoring infrastructure that can leverage available cores to continuously inspect running programs and report on a broad class of memory errors. RCORE uses extensive static analysis to extract as many PSIs as possible from a given program and make them available at runtime for fine-grained invariants analysis. RCORE covers all the standard C features and explicitly supports application-specific memory management not to lower the accuracy of the results at invariants checking time. Our investigation demonstrates that common memory errors can all be mapped to PSI violations and classified to provide an informative feedback to the developers. Our invariants analysis can be used to detect both dangerous behavior (e.g., long-lived dangling pointers) and memory corruption. In the latter case, our dynamic analysis concentrates on the effect of the corruption rather than on the cause, enabling probabilistic detection of arbitrary memory errors, even when the cause is unknown or not directly controlled. As a result, RCORE can seamlessly detect memory corruptions in the program state caused by the libraries or by arbitrary hardware memory errors.

Acknowledgment

We would like to thank the anonymous reviewers for their comments. This work has been supported by European Research Council under ERC Advanced Grant 2008 - R3S3.

References

[1] NIST, "National vulnerability database," http://nvd.nist.gov.

[2] IBM Security X-Force, "Mid-Year trend and risk report," http://www-935.ibm.com/services/us/iss/xforce/trendreports, 2012.

[3] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors," in *Proc. of the 18th USENIX Security Symp.*, 2009, pp. 51–66.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. on Inf. and System Security*, vol. 13, no. 1, pp. 1–40, 2009.

[5] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proc. of the 14th USENIX Security Symp.*, 2005, p. 17.

[6] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "PAriCheck: an efficient pointer arithmetic checker for C programs," in *Proc. of the Fifth ACM Symp. on Inf., Computer and Commun. Security*, 2010, pp. 145–156.

[7] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proc. of the Seventh USENIX Symp. on Oper. Systems Design and Impl.*, 2006, pp. 147–160.

[8] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective memory protection using dynamic tainting," in *Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Eng.*, 2007, pp. 284–292.

[9] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *Proc. of the 28th Int'l Conf. on Software Eng.*, 2006, pp. 162–171.

[10] O. Rowase and M. S. Lam, "A practical dynamic buffer overflow detector," in *Proc. of the 11th Network and Distributed System Security Symp.*, 2004, pp. 159–169.

[11] J. D. Hiser, C. L. Coleman, M. Co, and J. W. Davidson, "MEDS: the memory error detection system," in *Proc. of the First Int'l Symp. on Engineering Secure Software and Systems*, 2009, pp. 164–179.

[12] P. Fonseca, C. Li, and R. Rodrigues, "Finding complex concurrency bugs in large multi-threaded applications," in *Proc. of the Sixth European Conf. on Computer Systems*, 2011, pp. 215–228.

[13] G. Novark and E. D. Berger, "DieHarder: securing the heap," in *Proc. of the 17th ACM Conf. on Computer and Commun. Security*, 2010, pp. 573–584.

[14] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proc. of the 21st Int'l Conf. on Software Eng.*, 1999, pp. 213–224.

[15] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proc. of the 24th Int'l Conf. on Software Eng.*, 2002, pp. 291–301.

[16] 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 *Proc. of the 37th Annual IEEE/ACM Int'l Symp. on Microarchitecture*, 2004, pp. 269–280.

[17] M. Dimitrov and H. Zhou, "Unified architectural support for soft-error protection or software bug detection," in *Proc. of the 16th Int'l Conf. on Parallel Architecture and Compilation Techniques*, 2007, pp. 73–82.

[18] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: an approach for the anomaly-based detection of state violations in web applications," in *Proc. of the 10th Int'l Conf. on Recent advances in intrusion detection*, 2007, pp. 63–86.

[19] S. V. Adve, V. S. Adve, and Y. Zhou, "Using likely program invariants to detect hardware errors," in *Proc. of the IEEE Int'l Conf. on Dependable Systems and Networks*, 2008, pp. 70–79.

[20] K. Pattabiraman, G. P. Saggese, D. Chen, Z. T. Kalbarczyk, and R. K. Iyer, "Automated derivation of application-specific error detectors using dynamic analysis," *IEEE Trans. Dep. Secure Comput.*, vol. 8, no. 5, pp. 640–655, 2011.

[21] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer, "Using generalization and characterization techniques in the anomaly-based detection of web attacks," in *Proc. of the 13th Network and Distributed System Security Symp.*, 2006.

[22] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer, "Exploiting execution context for the detection of anomalous system calls," in *Proc. of the 10th Int'l Conf. on Recent Advances in Intrusion Detection*, 2007, pp. 1–20.

[23] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proc. of the Int'l Symp. on Code Generation and Optimization*, 2004, p. 75.

[24] P. Akritidis, "Cling: a memory allocator to mitigate dangling pointers," in *Proc. of the 19th USENIX Security Symp.*, 2010, p. 12.

[25] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Reconsidering custom memory allocation," in *Proc. of the 17th ACM Conf. on Object-oriented Programming, Systems, Languages, and Applications*, 2002, pp. 1–12.

[26] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur, "Runtime detection of heap-based overflows," in *Proc. of the 17th USENIX Systems Admin. Conf.*, 2003, pp. 51–60.

[27] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," in *Proc. of the 16th Symp. on Parallelism in Algorithms and Architectures*, 2004, pp. 206–215.

[28] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley, "A concurrent dynamic analysis framework for multicore hardware," in *Proc. of the 24th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Appilcations*, 2009, pp. 155–174.

[29] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proc. of the IEEE Symp. on Security and Privacy*, 2008, pp. 263–277.

[30] Q. Zeng, D. Wu, and P. Liu, "Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures," in *Proc. of the 32nd ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 2011, pp. 367–377.

[31] "nginx," http://nginx.org.

[32] "lighttpd," http://www.lighttpd.net.

[33] "httperf," http://www.hpl.hp.com/research/linux/httperf.

[34] P. Ratanaworabhan, B. Livshits, and B. Zorn, "NOZZLE: a defense against heap-spraying code injection attacks," in *Proc. of the 18th USENIX Security Symp.*, 2009, pp. 169–186.

[35] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattle, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. of the Seventh USENIX Security Symp.*, 1998, p. 5.

[36] T. Chiueh and F. Hsu, "RAD: a compile-time solution to buffer overflow attacks," in *Proc. of the 21st IEEE Int'l Conf. on Distr. Computing Systems*, 2001, pp. 409–417.

[37] Y. Younan, D. Pozza, F. Piessens, and W. Joosen, "Extended protection against stack smashing attacks without performance loss," in *Proc. of the 22nd Annual Computer Security Appl. Conf.*, 2006, pp. 429–438.

[38] E. D. Berger and B. Zorn, "DieHard: probabilistic memory safety for unsafe languages," in *Proc. of the 27th ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 2006, pp. 158–168.

[39] S. Van Acker, N. Nikiforakis, P. Philippaerts, Y. Younan, and F. Piessens, "ValueGuard: protection of native applications against data-only buffer overflows," in *Proc. of the Sixth Int'l Conf. on Inf. Systems Security*, 2010, pp. 156–170.

[40] K. Avijit, P. Gupta, and D. Gupta, "TIED, LibsafePlus: tools for runtime buffer overflow protection," in *Proc. of the 13th USENIX Security Symp.*, 2004, p. 4.

[41] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Safe and automatic live update for operating systems," in *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2013, pp. 279–292.

[42] G. He and A. Zhai, "Efficient dynamic program monitoring on multi-core systems," *J. of Systems Architecture*, pp. 121–133, 2011.

[43] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan, "Parallelizing dynamic information flow tracking," in *Proc. of the 20th Symp. on Parallelism in Algorithms and Architectures*, 2008, pp. 35–45.

[44] E. E. Aftandilian, S. Z. Guyer, M. Vechev, and E. Yahav, "Asynchronous assertions," in *Proc. of the 26th ACM Conf. on Object-oriented Programming, Systems, Languages, and Applications*, 2011, pp. 275–288.

[45] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proc. of the Fourth European Conf. on Computer Systems*, 2009, pp. 33–46.

[46] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz, "Multi-variant program execution: Using multi-core systems to defuse Buffer-Overflow vulnerabilities," in *Proc. of the 2008 Int'l Conf. on Complex, Intelligent and Software Intensive Systems*, 2008, pp. 843–848.