Chapter 11

Related Work

In this chapter, we describe work related to our runtime monitoring approach for preventing security violations and for finding programming errors. We organize our survey of related work by first exploring runtime monitoring approaches in increasing order of refinement, beginning with security-oriented approaches that only target known vulnerabilities, progressing into approaches that enforce memory safety and type safety. We then touch upon some approaches that supplement dynamic checks with static analysis, and error-detection approaches that are purely static.

11.1 Security-Oriented Approaches

With software security becoming a growing concern, a number of runtime-monitoring approaches have been developed with the goal of preventing known methods of attacks from succeeding. Most of these approaches have focused on buffer overruns, which is the most common vulnerability to be found in programs, and stack smashing, which is probably the easiest way to exploit a buffer-overrun vulnerability.

StackGuard [Cow+98, Cow+00] is a security tool that instruments a program to place a “canary” word next to the return address in the activation record when entering a new function. Prior to returning from the function, the canary value is checked to see if it has changed; if so, the return address may have been tampered with. PointGuard [Cow+00] is an extension of StackGuard that uses the same canary approach to detect tampering of function
pointers and longjmp buffers. The approach has a low runtime overhead, but the adjacent-canary approach assumes that the mechanism for overwriting the vulnerable location is a “continuous overflow” that overwrites a contiguous block of memory; the protection can be circumvented by a “direct write” method — e.g., via a format-string vulnerability — to modify the return address without changing the canary value.

To address this shortcoming, they tried using a fine-grained memory-protection mechanism called MemGuard [Cow+98] to write-protect the return address in the activation record; however, the overhead incurred was exorbitant. Other approaches for protecting the return address have also addressed this concern. StackShield [Sta00] copies the return address to an array, the Global Ret Stack, declared elsewhere in memory; the return value is retrieved from the Global Ret Stack prior to returning from a function. This approach effectively shifts the sensitive ‘return-address’ location from the activation record to the Global Ret Stack, which is still prone to tampering — i.e., an attack could be effected by overwriting the Global Ret Stack rather than the activation record. The Return Address Defender (RAD) [Chi+01] addresses this shortcoming by protecting the memory containing the return addresses — in their case, called the Return Address Repository (RAR) — in one of two ways:

1. MineZone RAD surrounds the RAR with buffers that are declared read-only. This prevents the RAR from being written by a continuous overflow, but not by a direct write.

2. Read-Only RAD declares the RAR itself to be read-only, thus preventing it from being corrupted via direct writes. However, this introduces a significant runtime overhead, since the RAR needs to be un-protected and re-protected at the start of each function to push the return address onto the RAR.

To achieve both better protection and performance, SmashGuard [Özd+02] proposes storing the return addresses in a hardware stack.
Libverify and Libsafe [Bar+00] are libraries that respectively check for stack smashing and buffer overruns. Libverify implements the StackGuard canary approach by adding instrumentation at link time rather than compile time. Libsafe contains instrumented versions of library functions that are normally prone to buffer overruns (like `strcpy`). The libsafe versions of these functions check that writes to a buffer do not exceed the bounds of the stack frame containing the buffer. This prevents library functions from overflows into the return address, but does not prevent them from overflows into other sensitive locations within the same stack frame, like function pointers, nor does it prevent overflows that are not caused by a library function.

These approaches are, by and large, \textit{ad hoc} solutions to security vulnerabilities, and only address specific forms of attack that are known and well-understood. Because most attacks take advantage of weaknesses in the C language and its weak enforcement of memory and type safety, there is a trend towards developing stronger versions of C to prevent \textit{any} erroneous behavior from happening.

### 11.2 Memory and Type Safety

The lack of memory-safety enforcement is arguably the weakest aspect of the C language: memory-safety errors are easy to induce, can be difficult to detect and diagnose, and are most vulnerable to attacks. For this reason, memory safety has been the focus of many approaches to finding errors and preventing attacks.

#### 11.2.1 Fat Pointers

Fat pointers have been used to check for memory safety violations in bcc [Ken83], Integral-C [Ros86], Saber-C [Kau+88], and rtcc [Ste92]. These were all designed for use in a debugging setting, and incurred high runtime overheads. Further, they only detected spatial access errors and not temporal access errors.

Safe-C [Aus+94] was the first approach to systematically account for temporal access errors. This was done by associating each storage allocation (stack frame or `malloc` block)
with a unique ID (called a “capability”), and augmenting the fat pointer to record the capability of the intended target. They also used a runtime optimization to skip redundant checks, and had a 2–6× runtime overhead. Keen et al. [Kee+02a] proposed using hardware to improve this performance, with a hardware accelerated table to speed up the temporal checks, and a hardware reduction queue to detect redundant checks dynamically.

Guarding [Pat+97] is an approach that performs spatial and temporal access checks similar to Safe-C, but it decouples the access checks from the original program, and executes these checks in a shadow process separately from the original program — possibly in parallel on a multi-processor, to improve runtime performance.

Cyclone [Jim+02] is version of the C language that enforces memory-safety at runtime. The language distinguishes between never-NULL pointers, regular pointers, and fat pointers, and instruments uses of the latter two with runtime checks for null dereference and spatial access errors respectively. To prevent temporal access errors, Cyclone requires the use of a garbage collector or a region-based memory-management scheme [Gro+02]. Due to the need to convert C pointers into the appropriate kind of Cyclone pointer, and to adapt to the region-based memory management, porting existing C programs into Cyclone becomes a significant barrier to its widespread use.

CCured [Nec+02, Con+03] is a system that enforces memory safety and limited type safety (distinguishing only between pointers and non-pointers) in C programs. It uses fat pointers to detect spatial access errors, and garbage collection to prevent temporal access errors. To make this approach efficient, it restricts some of the flexibility of C, and so disallows certain valid C operations (like storing the address of a stack variable on the heap, and dereferencing a pointer that has been cast to/from an integer). It may also reports some false positives, such as when accessing open arrays declared with size 1 (see Section 3.1, page 26), or assigning between pointers declared to point to different-sized underlying array types. To reduce the overhead of runtime checks, CCured uses a static type-inference scheme to classify pointers into safe, sequence, and dynamic pointers, so that runtime checks can be reduced for sequence pointers and eliminated for safe pointers. The goal of their type
inference is thus similar to that of our Extended Points-To and Type-Flow analyses, and the inference was able to improve CCured’s runtime overhead from 6–20× to 1–2× [Nec+02].

A major complication with the fat-pointer approaches is in interacting with uninstrumented libraries. Due to the change in the pointer representation, fat pointers must be translated to regular pointers before being passed to uninstrumented functions — an exercise that requires manual effort [Har+03]. To address this concern, Jones and Kelly [Jon+97] enforce memory safety by storing the set of valid pointer targets in a separate data structure (in their implementation, a splay tree), and check each pointer operation (including pointer arithmetic) to ensure that it remains within the bounds of its valid target. This allows the pointer representation to remain unchanged, which lets instrumented modules interoperate with uninstrumented ones, but their runtime overhead is high (5–6× slowdown), and they restrict certain unorthodox operations like computing an intermediate pointer value more than one byte past the end of an array.

CRED [Ruw+04] extends the Jones and Kelly approach to allow pointers to point beyond one past the end of an array. When a pointer value is computed that points beyond one past the end of an array, an additional level of indirection is introduced to associate the pointer with a reference to its intended target. While this enables CRED to handle a larger set of programs than Jones and Kelly, there remain cases that are not safely handled by CRED, such as dereferencing a pointer value cast from an integer that has undergone integer arithmetic. The overhead of CRED is similar to Jones and Kelly, and quite high (up to 12× slowdown), but restricting their approach to check only char pointer dereferences was found to decrease runtime overhead significantly (to 1.3–2.3× slowdown). They argue that since most buffer-overflow attacks exploit string operations (that operate via char pointers), restricting the checks to char-pointer operations does not reduce the protection of CRED.

Compared to the Jones and Kelly approach and CRED, our tagged memory approach has the benefit of interoperability with uninstrumented modules while maintaining a low runtime overhead, without restricting the language, and without limiting checks to char
pointers. In fact, the char-pointer-only checking is an option that could be adopted by the MSE to further reduce our runtime overhead.

### 11.2.2 Tagged Memory

The tagged memory approach sacrifices completeness for efficiency and practicality: while it cannot guarantee detecting all memory safety errors, it can be much more flexible than the fat-pointers approach while still being able to detect most errors. It has been effectively employed in a number of tools.

Purify \([Has+92]\), is a dynamic debugging tool for detecting memory-safety errors, memory leaks, and uninitialized memory accesses. It tags memory with two bits for each byte of memory: one bit to indicate whether the byte is allocated, and one bit to indicate whether the byte is initialized. It instruments object code, and has a fairly high overhead (about 15× slowdown) which is acceptable for a debugging tool, but impractical for use in deployed code.

Valgrind \([Net+03]\) performs similar checks, but it interprets the executable binary on a "synthetic CPU", and thus incurs a higher overhead (about 40× slowdown). The memcheck component of Valgrind associates each byte of memory with one valid-address bit to mark allocated memory and eight valid-value bits to detect uses of uninitialized memory.

Hobbes \([Bur+03]\) is another interpreter for binary code; it includes runtime type-checking that is very similar to that performed by the RTC. Each byte of memory is shadowed by a one-byte tag, which encodes its scalar type along with an initialized flag, and an invariant flag indicating whether the type of the byte is permitted to change. The interpreter incurs a 40× slowdown, and the type-checker an additional 100× slowdown.

These approaches all operate on binary code, which gives them the advantage of not requiring source code, but a disadvantage of being platform dependent, and being unable to take advantage of source-level information to improve runtime performance or coverage with static analysis. In our source-level approach, we were able to use static analysis to achieve significantly better runtime overheads.
11.2.3 Runtime Type Checking

Besides the RTC and Hobbes, there have been few other attempts to check type safety in C programs at runtime. The Saber-C [Kau+88] programming environment includes a rich interpreter-based debugging facility that, among other things, checks for both memory safety and type safety. Fail-Safe C [Oiw+02] is an approach to fully enforce memory and type safety at runtime for the full ANSI C. Memory safety is checked using fat pointers, while type safety is checked by maintaining data structures describing each (aggregate) type declared in the program, and using those data structures to resolve dereferences that have been affected by a type cast. Due to the change in data representation, they, like CCured, also require a complicated mechanism for interfacing with uninstrumented libraries, which also introduces some runtime overhead [Sue+03].

Runtime type checking is more prevalent in functional-style languages, and in languages with higher-order type systems. Despite the significant differences in the language structure and type system between these languages and C, there is some related work that is relevant to the RTC, in particular to optimizing runtime performance. For dynamically-typed languages like LISP and Scheme, Henglein [Hen92] proposed an efficient type-inference algorithm very similar in spirit to our flow-insensitive Type-Flow Analysis. Soft Typing [Car+91] is a combination of static and dynamic typing, where a program is first statically type-checked as much as possible, before dynamic type-checking is applied to the parts of the program that did not pass the static type-checker. Chailloux et al. [Cha+97] argued that, although runtime tags are not necessary for ML programs to run correctly, they can be useful for debugging, as well as to aid the garbage collector. They described an approach that stores type information for the heap in a mirror space, which allowed their implementation of a stop-and-copy garbage collector to be space efficient.

11.2.4 Other Runtime Monitoring Ideas

DynamoRIO [Dyn] is an approach to add runtime-monitoring checks on-the-fly (i.e., while the program is running), and to use dynamic optimization techniques to improve
performance. *Program shepherding* [Kir+02] is a specific application built on top of this infrastructure for enforcing security policies at runtime; this approach is able to guarantee that a given runtime check can never be circumvented.

There has been some recent work that extends dynamic approaches to detect potential errors that are not necessarily exercised during execution. Haugh and Bishop [Hau+03] focus on library functions that are known to be vulnerable to buffer overruns, like `strcpy`, and report a potential error if, for example, the size of the source buffer in a call to `strcpy` is greater than the size of the destination buffer, even though no buffer overrun actually occurs on the given execution. Larson and Austin [Lar+03] describe a similar but more general and more precise approach. They track more runtime information (such as the maximum potential string length in a given buffer) and handle more operations (beyond the vulnerable library functions), allowing them to detect more errors and fewer false positives.

### 11.3 Eliminating Array-Bounds Checks

Another relevant area that has been extensively researched is the elimination of runtime array-range checks. Early work includes Harrison [Har77], who relied on the presence of structured loops to infer range information, and Welsh [Wel78], who made use of explicit subrange type declarations in Pascal to optimize range checks. Markstein *et al.* [Mar+82] described using code motion and common subexpression elimination to optimize range checks, approaches later refined by Asuru [Asu92], Gupta [Gup93], and Kolte and Wolfe [Kol+95]. Bodik *et al.* [Bod+00] described an efficient analysis that allows array-bounds checks in Java programs to be eliminated on-demand. Luján *et al.* [Luj+02] explore ways to allow bounds-checking elimination strategies to be effective in the multi-threaded setting of Java. Most of the approaches studied have dealt with languages in which array accesses are more tightly controlled, while our pointer-range analysis had to contend with complications arising from pointer arithmetic and type casts in C.
11.4 Static Error Detection

The goal of the SAFECode [Kow+02, Dhu+03] project is to statically validate that memory safety errors can never occur, thus avoiding the need for runtime checks. In essence, the idea amounts to restricting the set of valid programs to be those that pass their static checks. An example restriction is that array indices must be affine in relation to the array sizes; this property is required because of the limitations of range analysis, but is non-trivial to state, and might not be easy for a programmer to adhere to.

Static error-detection techniques analyze a program without executing it to find potential errors or security holes. The main benefit of static techniques is that they can detect errors in portions of the program that are infrequently executed. Unfortunately, precise static techniques are expensive, and thus do not scale to large programs. In order for static approaches to scale, either the user must supply annotations (as in LCLint/Splint [Eva96, Lar+01] and ESC [Det+98]), or a less precise analysis must be performed (which may lead to missing some potential errors or to reporting false positives [Wag+00, Dor+01]), or the scope of the analysis must be limited, either by checking only certain paths (like PREFIX [Bus+00]), or simplifying the properties to be checked [Bal+01, Ash+02, Liv+03, Che+04].