StackArmor: Comprehensive Protection from Stack-based Memory Error Vulnerabilities for Binaries

Xi Chen* Asia Slowinska† Dennis Andriesse† Herbert Bos§ Cristiano Giuffrida§
Department of Computer Science
VU University Amsterdam, The Netherlands
*x.chen@vu.nl, †{asia,da.andriesse}@few.vu.nl, §{herbertb,giuffrida}@cs.vu.nl

***Abstract*—*StackArmor* is a comprehensive protection technique for stack-based memory error vulnerabilities in binaries. It relies on binary analysis and rewriting strategies to drastically reduce the uniquely high spatial and temporal memory predictability of traditional call stack organizations. Unlike prior solutions, *StackArmor* can protect against arbitrary stack-based attacks, requires no access to the source code, and offers a policy-driven protection strategy that allows end users to tune the security-performance tradeoff according to their needs. We present an implementation of *StackArmor* for x86_64 Linux and provide a detailed experimental analysis of our prototype on popular server programs and standard benchmarks (SPEC CPU2006). Our results demonstrate that *StackArmor* offers better security than prior binary- and source-level approaches, at the cost of only modest performance and memory overhead even with full protection.**

I. Introduction

While common defenses like W⊕X, canaries, and traditional ASLR prevent naïve return address overflows and code injection attacks, they have done little to eliminate stack-based attacks altogether. Mainly, the complexity of the attacks has increased as attackers resort to advanced techniques like Return-Oriented Programming (ROP) [69]. Likewise, they exploit the stack's predictable layout to disclose useful information, stored in current, previous, or reused stack frames [24]. We conclude that, despite all efforts, the stack remains a hugely attractive target for attackers, mainly because it is an exploit-friendly contiguous mapping with spatial and temporal allocation locality that is *entirely* predictable—obviating even the need for "feng shui" strategies on the heap [74].

In this paper, we address the problem at its root by completely abandoning the idea of a linearly growing stack. We statically rewrite binaries to isolate and fully randomize the locations of stack frames and individual stack buffers, countering both spatial attacks like overflows and temporal attacks like stack-based use-after-frees.

While ours is an extreme solution that provides more comprehensive protection than prior solutions, we are not the first to argue for better stack defenses. Existing approaches include compiler extensions [10], [11], shadow stacks [18], [26], [28], [44], [62], [65], [70], [80], Control-Flow Integrity (CFI) [8], [34], [82], and binary rewriting to add buffer protection [72], but they either rely on source code and leave binaries at the mercy of attackers, or offer only very limited protection. Specifically, there is currently no stack protection technique for binaries that mitigates all of the following attack vectors: (i) buffer overwrites and overreads within a stack frame, (ii) buffer overwrites and overreads across stack frames, (iii) stack-based use-after-frees, (iv) uninitialized reads (in reused stack frames). As a result, stack attacks are still rampant. Attackers use them both to divert the control flow (and, e.g., start off a ROP chain) and for memory disclosures [24].

Information leakage and buffer overflow attacks, in particular, are greatly helped by the predictability of the stack layout. Although the start is typically randomized, the stack itself grows in an entirely predictable fashion, making the disclosure of canaries, return addresses, or data pointers of previous stack frames as simple as leaking uninitialized data or exploiting buffer overreads. The same applies to exploits modifying data in another stack frame. For example, randomization between stack frames would have stopped recent high-profile attacks on Asterisk [38], Xen [39], Kerberos [36], and MS Office [37].

Our focus on binaries is neither academic nor fundamental, but important in practice: the adoption of advanced security measures in popular compilers is slow. Compiler maintainers are conservative and wont to reject options that incur significant overhead. The `-fstack-protector-strong` option in `gcc` is a case in point: it had to be tailored to a very narrow threat model for performance reasons. As most vendors simply use common compilers like `gcc`, any measure not added to it for performance reasons will not make it into their products. Unless they apply the defenses at the binary level, users cannot decide for themselves to sacrifice some performance for better security.

Contributions We introduce *StackArmor*, a novel stack protection technique that shields binaries from all of the above attacks. To provide comprehensive protection, *StackArmor* relies on static analysis enabled by state-of-the-art binary analysis tools—which provide the necessary program abstractions, such as functions and their control-flow graphs. Our static analysis is also supported by information on the location and size of stack objects, for example provided by debug symbols (similar to prior binary-level protection techniques [44]) or dynamic reverse engineering techniques [53], [71]. *StackArmor* can also operate in complete absence of these, by gracefully reducing its (intra-frame) protection guarantees. Using binary rewriting to instrument call and return instructions, *StackArmor* provides tailored protection based on application-specific performance and security requirements. In full protection mode, *StackArmor* relies on a combination of randomization, isolation, and secure

NDSS '15, 8-11 February 2015, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2015.23248

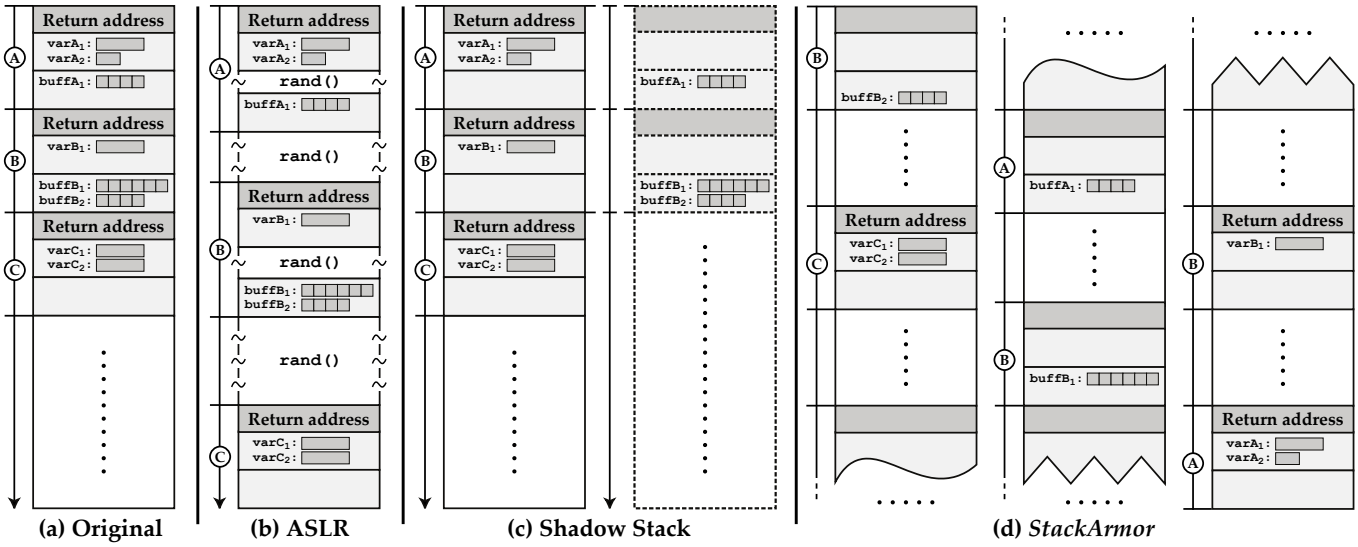

Fig. 1. Comparison of different stack protection techniques.

allocation techniques to create the illusion that *all* the stack frames and the individual stack buffers are drawn from a fully randomized space with no spatial or temporal predictability guarantees. Unlike all the existing solutions, this strategy can comprehensively protect against *arbitrary* stack-based attacks.

To summarize, our contributions are:

- We present *StackArmor*, a novel stack protection technique which combines inter- and intra-frame defenses to stop arbitrary spatial and temporal attacks.
- We present an implementation of *StackArmor* for x86_64 Linux. Ours is the first system that provides such comprehensive stack protection for binaries.
- We provide a detailed experimental evaluation of our prototype, and show that it achieves a modest performance and physical memory overhead of 5% and +3 MB, respectively, on average, on single-threaded server programs, while scaling well even to heavily threaded server programs (28% and +112 MB with 100 worker threads, on average) with full protection.

Outline: The remainder of the paper is organized as follows. Section II provides background information on the various classes of stack-based memory error attacks and compares the protection guarantees offered by *StackArmor* with those of prior techniques. Section III and Section IV present the design and implementation of *StackArmor* and discuss the limitations of our current prototype. Section V presents experimental results to assess the viability and effectiveness of our stack protection technique. Finally, Section VI surveys related work and Section VII concludes the paper.

II. Threat Model

StackArmor prevents memory error attacks exploiting spatial and temporal locality of reference on the stack, i.e., *spatial* and *temporal* attacks, respectively. This section briefly elaborates on both classes of attacks and discusses the limitations of existing stack protection techniques.

A. Spatial Attacks

Spatial attacks exploit memory errors to access data outside the prescribed buffer bounds. Well-known memory error examples include stack-based buffer overflows and underflows. Attackers exploit them to corrupt memory objects with malicious buffer writes, or leak secrets through unintended reads. Attacks can target both control data, e.g., return addresses or function pointers, and noncontrol data, e.g., variables storing user privilege levels.

To access a target object, an attack first estimates its address and next obtains a pointer to the target location via a vulnerable buffer. It either exploits a vulnerable buffer and a target object located in the same stack frame (*intra-frame* attack) or in different ones (*inter-frame* attack). In a traditional stack organization, both stack frames and per-frame objects are contiguously allocated in memory, so the attack can safely rely on the predictability of the relative distance between the buffer and the target object.

B. Temporal Attacks

Temporal attacks exploit memory errors to access data outside the prescribed object lifetime. Such attacks rely on predictable memory reuse to read/write data from a newly allocated object via a reference to a deallocated object or, conversely, read data from a deallocated object via a reference to a newly allocated object. Memory errors originating these attacks are commonly referred to as *use-after-free* and *uninitialized read* errors, respectively. They can be successfully exploited to corrupt or leak both control and noncontrol data.

On the stack, temporal attacks exploit erroneous memory accesses into deallocated stack frames (via dangling pointers), or into uninitialized stack variables containing old data. In a traditional stack organization, stack frames are allocated and deallocated in a predetermined order, so an attack can determine which two objects overlap across stack frame allocations and corrupt/leak the intended data. In this scenario, the attack relies on the predictability of stack frame reuse induced by stack memory allocation.

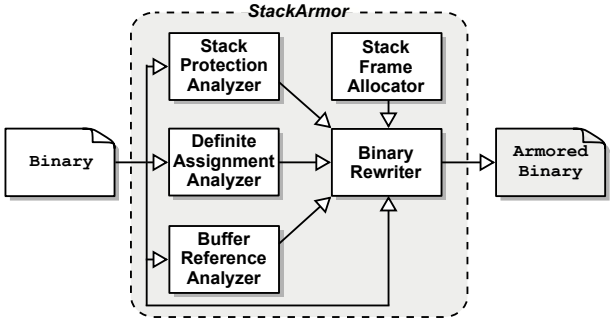

Fig. 2. High-level overview of *StackArmor*.

C. Defenses

Figure 1 compares the stack organization adopted by different protection techniques to counter attacks.

Modern ASLR techniques introduce random gaps between stack frames [18], [33] and between buffer and nonbuffer stack objects [33]—separating and permuting them in two adjacent per-frame regions (Figure 1b). This strategy alone does not change the order of the stack frames nor does it isolate vulnerable stack buffers, exposing the stack to guessing or spraying—spatial and temporal—attacks. This problem is exacerbated by the gaps being limited in size for practical reasons and often statically determined for efficiency reasons [33], resulting in even poorer randomization entropy, stack frame reuse unpredictability, and resilience to information leakage attacks.

Modern shadow stack techniques [18], [20], [44], [80], in turn, isolate the vulnerable stack buffers on a separate, but contiguous, shadow stack (Figure 1c). This strategy alone does not prevent buffers from attacking each other in a predictable way in intra- and inter-frame spatial attacks nor does it attempt to protect against temporal attacks.

StackArmor, finally, completely disrupts the traditional stack organization, creating the illusion that stack frames and vulnerable buffers are neither temporally nor spatially adjacent in memory, but randomly drawn and isolated from one another (Figure 1d). This strategy prevents *all* the spatial and temporal attacks considered.

III. StackArmor

Figure 2 illustrates the overall *StackArmor* architecture. It consists of three *analysis* modules, a binary *rewriter*, and a secure *allocator*. The analysis modules provide support for the binary rewriting, while the allocator is employed to ensure an unpredictable allocation of stack frames. In this section, we describe the design of *StackArmor* and we defer the implementation details until Section IV.

The three analysis modules statically analyze each function found in the binary and determine what protection measures it requires. The *Stack Protection* (SP) analyzer conservatively decides which functions are not provably safe from spatial or use-after-free attacks, so need randomized (and isolated) stack frames. For this purpose, the analysis pinpoints functions that compute pointers to local variables.

Functions that are not assigned randomized stack frames still require protection from uninitialized reads. To this end, the *Definite Assignment* (DA) lists functions which are not provably safe from uninitialized variables. When such a function is called later, *StackArmor* zero-initializes its relevant stack objects, effectively creating the illusion that the stack frame has been allocated from a random pool of zero-initialized frames and preventing potential errors from being exploited.

The *Buffer Reference* (BR) analyzer identifies stack buffers (and their references) that are provably safe to relocate. This strategy is used to isolate potentially vulnerable stack buffers from the original stack and prevent intra-frame spatial attacks. To this end, the BR analyzer relies on information about location and size of all the per-function stack objects, for example provided by debug symbols or dynamic reverse engineering techniques [53], [71].

Next, *StackArmor* combines the results of the analyses and the *Binary Rewriter* instruments all functions that cannot be conservatively proven safe. It creates a new stack frame for each function call and for each stack buffer, while the *Stack Frame Allocator* ensures at runtime that the frames are allocated in an unpredictable manner. After statically rewriting the binary, the resulting (*armored*) binary can run natively.

A. Stack Protection Analyzer

The *SP analyzer* employs static analysis to conservatively identify functions that cannot be proven safe from spatial and use-after-free attacks, so require stack protection. It classifies as *SP-unsafe* all functions that compute pointers to local variables, i.e., (i) have stack-allocated buffers, (ii) call `alloca` or (iii) contain stack variables that have their address taken. Our algorithm is inspired by the `-fstack-protector-strong` option in `gcc` [7], which uses similar analyses—at the source level—to identify functions prone to buffer overflows. One key difference is that our strategy is more generally tailored to locating *any* uses (and possibly leaks) of pointers into stack objects, allowing our analyzer to also identify functions prone to use-after-free attacks. Another difference is that operating at the binary level raises more challenges since stack accesses are mediated by the stack (or frame) pointer, generally subject to aliasing.

To address this challenge, the SP analyzer overapproximates the conditions above using a data-flow analysis over the Control-Flow Graph (CFG) of every function. In *SP-safe* functions, *StackArmor* allows references to stack objects only via the stack (or frame) pointer and a constant offset. More specifically, for every function, the SP analyzer performs a forward analysis of its CFG and marks the function as *SP-unsafe* if any of the following *SP-safety* rules hold:

1) The stack is accessed through the stack (or frame) pointer and an offset stored in another register.
2) The stack (or frame) pointer or derived pointers are stored into registers or memory outside the function's prologue and epilogue.
3) The stack (or frame) pointer is manipulated outside the function's prologue and epilogue.

Summarizing, (1) detects when a stack buffer is accessed in its local function, (2) prohibits implicit accesses to stack

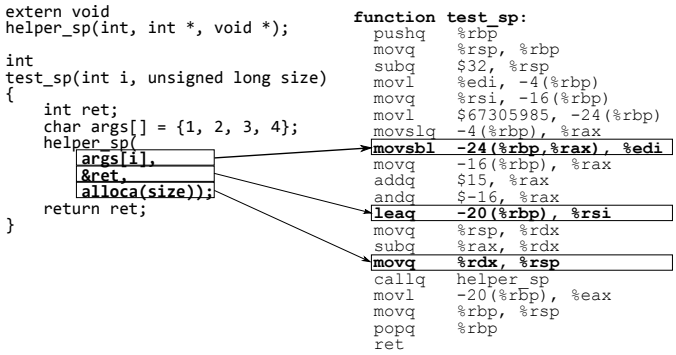

Fig. 3. A sample *SP-unsafe* function that violates all the three *SP-safety* rules imposed by the BR analyzer.

variables, by confirming that their pointers are never stored or passed to callees, and (3) detects `alloca` invocations (and possibly other unsafe idioms). The example function in Figure 3 violates all the three *SP-safety* rules and is classified as *SP-unsafe*[1]. To read `args[i]`, the function accesses the stack through `%rbp` and `%rax`—violating (1). The second rule is violated when the function computes the address of `ret` and stores the resulting `%rbp`-derived pointer to `%rsi`. Finally, the invocation of `alloca` causes a manipulation of the `%rsp` register, which violates (3).

While seemingly very conservative, our analysis closely matches the behavior of modern compilers, which typically generate very simple (and efficient) stack-accessing instructions for functions that maintain no pointers into the stack. As shown in Section V-C, on average, our analysis classified 80% of functions across all the SPEC CPU2006 benchmarks as *SP-safe* (geometric mean).

B. Definite Assignment Analyzer

To determine the functions (and objects) that require protection from uninitialized reads, the *DA analyzer* relies on static analysis to conservatively identify all the *DA-unsafe* objects, i.e., stack objects that cannot be proven as initialized before they are first read. Our strategy is inspired by similar source-level analyses employed in safe languages to implement zero initialization semantics [46]. An important challenge when operating at the binary level is that object boundaries are no longer exposed in the code in any obvious way. Another challenge is that aliasing problems may generally prevent the analyzer from unambiguously mapping all the accesses to stack objects.

To address these challenges, the DA analyzer relies on two key observations. First, the functions that require ad-hoc uninitialized read protection are only those that have been marked as *SP-safe*—since the others are protected using randomization and isolation. All the *SP-safe* functions, in turn, have no buffers, pointers into the stack, or stack-accessing instructions that our simple data-flow analysis cannot map into a constant stack frame offset. As a consequence, these variables are not initialized in other functions, which drastically simplifies our definite assignment analysis, essentially reducing it to a basic intra-procedural data-flow analysis [43]. Second, once constant stack frame offsets are available for each load and store stack-accessing instruction, our analysis can simply operate at the byte rather than at the object level.

[1]All the assembly snippets presented in the paper have been generated with `clang` 3.3.

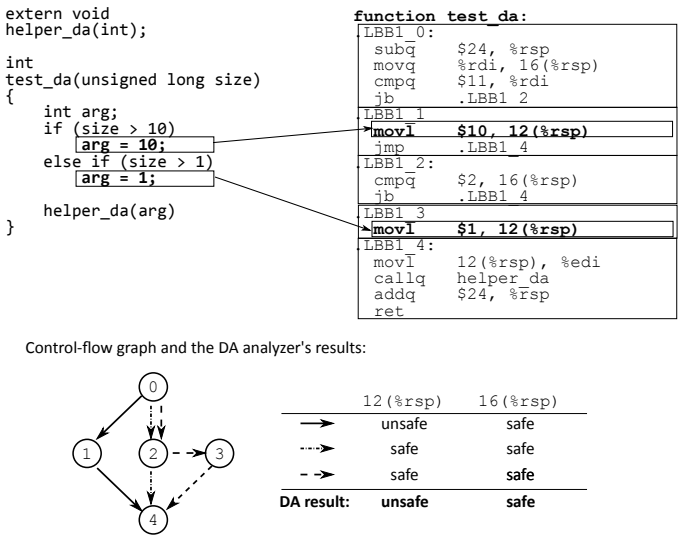

Fig. 4. A sample *DA-unsafe* function—on the CFG path marked with solid arrows, the analyzer cannot prove that `12(%rsp)` (containing the `arg` variable) is initialized.

To determine functions and stack variables that require protection (and thus zero initialization), the DA analyzer proceeds as follows. For every function, it traverses its CFG in depth-first fashion and maintains a per-path tag map to keep track of the bytes in the stack frame that have been read or written to in the current path. For every path, a first write-before-read event causes the DA analyzer to mark the target bytes as *path-safe* and a first read-before-write event causes the DA analyzer to mark the target bytes as *path-unsafe*. If the traversal reaches an unresolved control transfer or a function call, it marks all the bytes that are not marked at all yet as path-unsafe. At the end, all the bytes in the stack frame (and the function itself) that have been marked as path-unsafe at least once are marked as *DA-unsafe*, thus requiring uninitialized read protection. The example function in Figure 4 is *DA-unsafe*, since the analyzer cannot prove that on each CFG path, the stack location `12(%rsp)` (containing the `arg` variable) is written before it is read.

C. Buffer Reference Analyzer

For each function, the BR analyzer determines which stack buffers can be *safely* isolated in separate frames, i.e., while making sure that all references to these buffers are detected and relocated as well. The isolation serves as a protection against intra-frame spatial memory corruption attacks.

To this end, the BR analyzer performs an intra-procedural static analysis to unambiguously map all the instructions taking stack addresses. *StackArmor* can safely isolate a buffer only if it proves that none of its references are ever used to access other memory regions. Even though the BR analyzer relies on the available information on the location and size of all the stack objects (as provided by debug symbols or dynamic reverse engineering techniques [53], [71]), the mapping poses significant challenges. First, the stack (or frame) pointer is subject to aliasing. Another difficulty is that unlike source-level

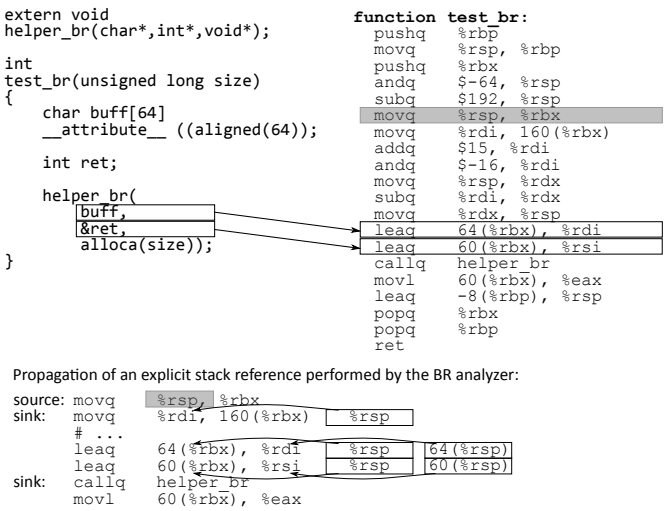

Fig. 5. A sample function with an ambiguous stack reference—the `%rbx` base pointer, derived from `%rsp`, is used to access two separate objects on the stack: the `ret` variable located at `64(%rbx)` and the `buff` object at `60(%rbx)`.

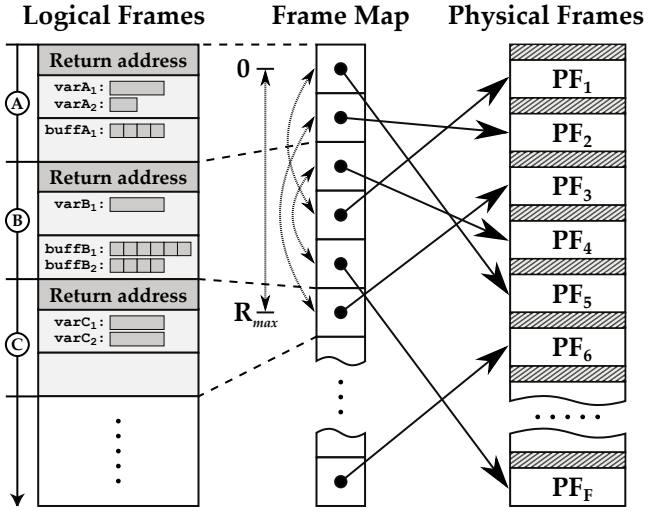

Fig. 6. *StackArmor*'s stack frame allocation strategy.

solutions [10], we cannot assume that the relative layout of independent objects in memory is undefined. At the binary level, references to stack objects are inherently ambiguous—due to intra-procedural compiler optimizations, a reference to an object could be later used to access a completely different object inside a function.

Figure 5 illustrates the problem. In this example, `clang` reserves a dedicated *base pointer* (`%rbx`) to access stack objects. The function prologue sets up the pointer to point to the bottom of fixed-size portion of the stack, i.e., excluding stack space dedicated to Variable-Length Arrays—or *VLAs*. In our example, this behavior is induced by the presence of stack alignment for the `buff` object and the VLA allocated on the stack using `alloca` [64]. `gcc` handles this situation in a similar way, mediating the necessary accesses to stack objects with a dedicated Dynamic Realigned Argument Pointer (*DRAP*) register (typically `%r10`) [49]. Using an additional register to access the stack causes a stack reference (i.e., `movq %rsp, %rbx`) to be used to access distinct stack objects (i.e., `buff` and `ret`) before calling `helper_br`. As detailed in the figure, the BR analyzer detects the ambiguity and refuses to remap the stack references for the given function.

The algorithm implemented in the BR analyzer operates in two steps. First, for each function, it identifies and propagates all the explicit stack references down the CFG. The mechanism is inspired by the way prior techniques [73], [75] use constant propagation to discover targets of indirect calls. Next, the analysis verifies that each reference targets a single stack object.

In the first step, the BR analyzer runs an intra-procedural flow-sensitive static data-flow tracking analysis [30] to determine where stack addresses are dereferenced, stored to memory, or escape the current function. It takes two types of taint seeds present in the instructions of the CFG: constants and explicit stack references, i.e., operands of the form `rsp+offset` or `rbp-offset`, where `offset` is an immediate value. It considers each reference in turn and treats this reference and all the constants as tainted. With this setup, for each control-flow path, it propagates the tainted values down the CFG, in a depth-first fashion and builds up expressions representing the computed values. When the propagation reaches a *sink* instruction, it labels the propagated reference as follows:

1) If the sink is an instruction accessing memory at an address tainted by the reference:
   – If the address evaluates to a single stack location, the analysis locates the target stack object, labels the reference accordingly, and continues the propagation;
   – Otherwise, e.g., if the address contains an unresolved index, the analysis labels the reference as `unknown` and stops.
2) If the sink is an instruction storing a value tainted by the reference to memory, the analysis conservatively assumes that it is a valid pointer and proceeds as in (1).
3) If the sink is a `call` instruction and a register holds a value tainted by the reference, the analysis conservatively assumes that this register may contain a valid pointer and proceeds as in (1).

Once the propagation completes, the analysis verifies that across all the paths, each reference is labeled with exactly one *known* object (i.e., a buffer). If this check fails, the analysis conservatively reports no buffers for the current function. If each reference can be successfully mapped into a single stack buffer, in turn, the analysis reports on all the buffers and all the stack-referencing instructions that reference those buffers. Note that, since our analysis is fully conservative, it may occasionally fail to isolate some buffers (see Section V) but it also allows for no false positives—which would otherwise result in instrumentation-induced undefined behavior—in practice.

D. Stack Frame Allocator

Figure 6 depicts the allocation strategy adopted in our stack frame allocator. For each thread call stack, the allocator maintains a pool of F contiguous physical frames (PFs), all preallocated in a random region of the virtual address space. Each physical frame consists of D data pages surrounded by 1 guard (nonmapped) page to isolate frames from one another. To map individual logical frames into one or more physical

stack frames of $D+1$ pages, our allocator relies on a frame map of F preallocated entries, each initialized with a pointer to a physical frame. At runtime, the frame map is managed similarly to a stack using a dedicated index. Allocating a frame entails fetching the next entry and decrementing the index, which is again incremented at deallocation time.

To ensure that the relative distance between physical frames is unpredictable, the entries in the frame map are initialized with a random permutation of the physical stack frames. This static randomization strategy efficiently protects against spatial attacks, but is alone insufficient for temporal attacks, since the entries in the frame map can still be predictably reused across consecutive calls—e.g., in a loop. To protect against temporal attacks, our allocator performs in-place frame map randomization, swapping the next entry with the entry located at a random offset $R \in [1; R_{max}]$ before allocating a new frame. R is computed using a global counter and a random number provided by the new `rdrand` x86 instruction. This in-place randomization strategy eliminates the need for free lists and efficiently satisfies all our requirements.

In addition, to minimize physical memory consumption—instrumented stack frames are allocated at page granularity with low reuse—and comply to the restrictions on the maximum number of (guarded) virtual memory areas (VMAs) imposed by the operating system—i.e., 65,535 on stock Linux—our allocator retains fine-grained control over the memory pages pre-allocated in the stack frame pool using predetermined *soft* limits. In particular, while the parameter F establishes the maximum number of active stack frames, its soft limit F_{SL} determines how many frames are immediately made available to each application thread. The other $F-F_{SL}$ frames are initially all mapped as consecutive inaccessible pages—and thus accounted as a single VMA by the operating system. Every time a thread exhausts its currently available physical frames, our allocator doubles the value of the soft limit F_{SL} and remaps the new physical frames correctly. This strategy is crucial to increase the number of guard pages—and thus decrease the number of VMAs available to the program—only when strictly necessary.

A similar adaptive strategy is used to manage the individual physical stack frames. The soft limit D_{SL} determines how many data pages are immediately made available to each stack frame. The other $D-D_{SL}$ pages are initially all mapped as inaccessible. In rare cases in which more space is needed by the current stack frame, a user-level page fault handler (i.e., signal handler intercepting `SIGSEGV` signals) is used to increase D_{SL} on demand using the same exponential growth strategy described earlier. The soft limit D_{SL} is restored at its initial value only at stack frame deallocation time. Restoring the original soft limit also entails returning the extra memory pages to the operating system (i.e., using POSIX's `MADV_DONTNEED`, similar to [9]) and remapping them as inaccessible. This strategy is crucial to minimize physical memory consumption in (rare) cases of functions using a large amount of stack space and being potentially assigned by our allocator a different (random) physical stack frame at each invocation.

Our allocator allows users to fine-tune the individual configuration parameters, but also supplies carefully chosen default values that we found effective in common real-world programs. The parameter R_{max} controls the entropy of our in-place randomization strategy, but may also increase the *StackArmor*-induced memory usage—due to more resident physical frames—and performance overhead—due to poorer data cache locality. By default, *StackArmor* opts for the maximum-entropy configuration $R_{max}=F_{SL}$ in nonthreaded programs while gradually decreasing the per-thread value of R_{max} (and F_{SL} accordingly) with the number of active threads in multithreaded programs (-5% for each newly created thread, with a minimum $R_{max}=F_{SL}=128$). This strategy is crucial to strictly bound physical memory consumption in heavily threaded programs. We evaluate the performance and memory impact of this choice in Section V.

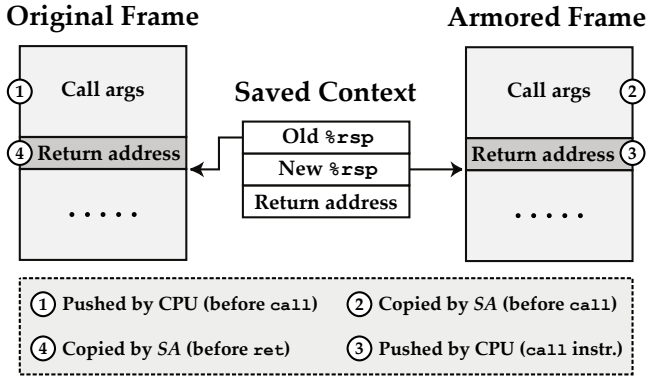

Fig. 7. *StackArmor*'s call site instrumentation strategy.

The maximum number of active stack frames F and its soft limit F_{SL}, in turn, default to $16,384$ and $1,024$ (respectively), values that do not aggressively reduce the maximum number of VMAs available to the program—i.e., 39% reduction on stock Linux for a program with 100 active threads within the soft limit—but can elastically adapt to programs with fairly deep instrumented call stacks. The number of per-frame data pages D and its soft limit D_{SL}, finally, default to the OS-specified maximum stack size for the program—i.e., 2,048 pages on stock Linux—and $0.1 \cdot D$ (respectively), values that yield a conservative stack allocation strategy while providing strong physical memory consumption guarantees.

E. Binary Rewriter

The binary rewriter relies on the information provided by the analysis modules to guide the instrumentation process. First, it instruments all the call sites invoking *SP-unsafe* functions allowing each caller to set up a new randomized and isolated (i.e., *armored*) stack frame for the callee's execution. The latter is functional to protect the callee against inter-frame spatial attacks and temporal attacks. Figure 7 depicts our call site instrumentation strategy. Before the `call` instruction, the rewriter requests the allocator to allocate the armored stack frame, copies over all the call arguments already pushed into the stack, and redirects the stack pointer `%rsp` to the new frame. The return address, the old `%rsp`, and the new `%rsp` are all saved in a dedicated *context* (maintained on a separate stack) for later use. After the `call` instruction, the callee runs protected with the armored stack and eventually returns. The rewriter instruments the return site (`ret` instruction) to deallocate the armored stack frame, restore the old `%rsp`, and push the saved (and *trusted*) return address into the stack to allow execution to return in a consistent way. While our call site instrumentation strategy is

necessary to correctly copy caller-specified arguments—whose number may change across call sites, due to variadic calls—into the armored stack frame, it may also complicate stack management when caller and callee cannot be statically paired with one another. Indirect calls, for example, need to be conservatively instrumented since their target may be *SP-unsafe*—albeit not statically known. An instrumented indirect call with a *SP-safe* callee, however, would cause execution to return from a call site with the `%rsp` still pointing into the armored stack frame. To address this problem, the rewriter instruments all the instructions following a call site which contains indirect calls, library calls, and other special idioms—e.g., `setjmp`—to restore the original stack and allow the caller to resume execution consistently. Our `setjmp` instrumentation also checks if control returned from a `longjmp` invocation and garbage collects all the deeper (and thus no longer needed) physical stack frames in that case. The complementary situation—an uninstrumented call site with an instrumented *SP-unsafe* callee—is also possible, for example when dealing with uninstrumented libraries calling program-specified callbacks. To detect (and simply ignore) this situation, the return site instrumentation checks if the current `%rsp` is lower than the new `%rsp` in the most recent saved context. Our instrumentation strategy can efficiently handle all the caller-callee combinations—i.e., instrumented caller and callee, uninstrumented caller and callee, instrumented caller and uninstrumented callee, and uninstrumented caller and instrumented callee—in a conservative way, allowing unrestricted use of shared libraries and arbitrary optimizations driven by our static analyzers.

To protect against intra-frame spatial attacks, in turn, the rewriter instruments all the *SP-unsafe* functions with buffers reported by the BR analyzer. In particular, it first instruments the entry site (i.e., after the function prologue) to relocate each buffer reported in a new stack frame provided by the allocator. Second, it remaps all the stack-referencing instructions reported by the BR analyzer to reference the corresponding buffers in their own independent frames. Such frames are later garbage collected when the main armored frame is deallocated.

To protect the remaining *SP-safe* functions against uninitialized reads, finally, the rewriter instruments the necessary entry sites to zero-initialize all the stack regions reported by the *DA analyzer*. To implement efficient zero initialization semantics, the rewriter also coalesces multiple `bzero` writes into the same memory word (8 bytes).

IV. Implementation

We implemented *StackArmor* to instrument 64-bit ELF binaries for the Linux x86_64 platform, but our prototype is easily portable to other UNIX systems. As *StackArmor* is built on top of PEBIL [52], it performs static instrumentation, i.e., it inserts additional code and data into an executable, and generates a new *armored* binary with permanent modifications. We first discuss the requirements for the disassembly process imposed by *StackArmor* and then we present some details of the instrumentation. We conclude with limitations of the current implementation.

A. Binary Disassembly and Analysis

StackArmor's static analyses require information about instructions, basic blocks, CFGs, and functions present in a binary. While we do not claim any contributions in the area of binary disassembly and we use existing tools, we show that *StackArmor* relies only on the correctness of disassembly and not its completeness. It is designed to cope with incomplete information—it gracefully reduces security guarantees without breaking the binary. Finally, we discuss what other assumptions *StackArmor* makes.

Incomplete disassembly. In principle, it is not feasible to fully disassemble arbitrary stripped x86 binaries statically [73], [78]. Due to indirect control flows and interleaving code and data, disassembly can be imprecise, so we get possibly incomplete information on instructions, basic blocks, CFGs, and functions. At the same time, binary disassembly is subject to active research [12], [13], [83] and our experience with standard tools that support both symbol and non-symbol based CFG reconstruction (Dyninst [16]) shows that lack of symbols is hardly a concern in practice if 100% accuracy is not strictly required (on SPEC benchmarks we missed only $1/12357$ functions and $7/325173$ basic blocks and edges due to lack of debug information).

Unresolved jumps: The first consequence of incomplete disassembly are unresolved `jmp` instructions, i.e., indirect jumps whose targets remain unknown. All the static analyses behave in a conservative way: the SP analyzer classifies a function with an unresolved jump as *SP-unsafe*, so the DA analyzer does not even consider it and the BR analyzer labels all its buffers as `unknown`. The rewriter, in turn, sets up a new and isolated stack frame for this function. If the rewriter misses and does not instrument a return instruction, *StackArmor* handles this case with no trouble, as we discussed in Section III-E.

Unresolved calls: StackArmor also needs to deal with unresolved indirect `call` instructions. Similarly to unresolved jumps, the three static analyses report conservative results. An unresolved `call` does not, however, influence the instrumentation of the caller function.

Missing functions: Since *StackArmor* is not aware of functions missed due to incomplete disassembly, it simply does not analyze or instrument them. As discussed in Section III-E, the binary rewriter ascertains that the binary works well even if an instrumented function calls an uninstrumented one, and the other way round.

Summarizing, in the presence of incomplete disassembly, *StackArmor* always errs on the safe side. If necessary, it just excludes a function from analysis and protection, so it executes as in the vanilla version of the binary.

Stack pointers and function prologue. The three *StackArmor*'s static analyses (Sections III-A-III-C) consider explicit stack references, i.e., instruction operands of the form `rsp+offset` or `rbp-offset`. While this definition of an explicit stack reference assumes the special role of the `rsp` and `rbp` registers, *StackArmor*'s actual implementation does not rely on `rbp` containing the base pointer. If, due to optimizations, `rbp` does not point to the beginning of the frame, nothing bad happens—the analyses are limited to the references derived from the `rsp` register. To detect and examine function prologues, *StackArmor* follows the ABI for x86_64/Linux [55]. Observe that `rsp`, however, is a *sacred* register, whose value is required by the important `push`, `pop`, `call`, and `ret` instructions. Thus, in practice, there is no reason why a program would ever use

`rsp` for anything else than the current stack position and we can therefore safely rely on its usage.

Function arguments. As we said in Section III-E, before a `call` instruction, the binary rewriter copies over all the call arguments already pushed into the stack. It examines the basic block containing the `call` instruction and checks how many bytes to transfer. Our argument-copying strategy poses very little restriction on the calling convention: (a) stack-based argument passing is done in a single basic block, (b) arguments are callee-owned. Even though it is possible to extend our prototype to an even more general solution as proposed in [12], [13], we are not aware of any calling convention violating (a)-(b)[2]. Our implementation was correct in all the server programs and standard benchmarks we tested.

B. Instrumentation

Our binary rewriter is implemented on top of PEBIL [52], an efficient binary instrumentation tool for Linux. PEBIL can place hooks in arbitrary binary points to call a predetermined handler enclosed in a shared library, with the instrumentation automatically saving and restoring registers to create a consistent execution context.

For our purposes, we extended the original PEBIL tool in three ways. First, we allowed only registers effectively used in *StackArmor*'s handler—which does not rely on external library calls on the instrumentation path—to be saved and restored at each invocation to minimize context switching costs at function entry/exit time. Second, we implemented support for handlers enclosed in a static library—injected by our rewriter into the binary—eliminating the costs associated with indirect PLT calls on the instrumentation path. This change enabled *StackArmor*'s stack frame allocator to be entirely implemented and compiled into a static library with efficient position-dependent code. Finally, we allowed PEBIL to access Thread-Local Storage (TLS), where our instrumentation stores references to per-thread metadata and stack frames managed by our allocator. Our current implementation completely allocates/deallocates per-thread data structures at thread creation/destruction time by default, but adopting more efficient pooling strategies—not necessary for our test programs—is straightforward.

One limitation of PEBIL is that it relies on debug symbols to generate a list of functions present in a binary and their CFGs. This is, however, not a limitation of *StackArmor*, but of the tool we used for familiarity. As we discussed in Section IV-A, *StackArmor* is designed to handle incomplete disassembly while gracefully reducing its security guarantees, without breaking the binary. At the same time, recent research shows how to disassemble stripped binaries [13], [83], as well as locate functions and resolve (at least some) indirect control flows with the aim of generating CFGs [12]. Thus, *StackArmor*'s prototype could be ported to other binary rewriting frameworks, such as Dyninst [16], which can operate even in the absence of relocation and debug symbol information.

[2]The tail call optimization violates (b) but even that does not require copying back callee-owned arguments given that the callee returns directly to the caller of the caller.

C. Limitations

The main limitation of the current *StackArmor* implementation—inherited from PEBIL—is the inability to support C++-style exceptions. This limitation—addressable with additional effort—did not prevent our prototype from running all the popular server applications and benchmarks considered in our evaluation.

V. Evaluation

We evaluated *StackArmor* on a workstation equipped with an Intel i7-4770K CPU clocked at 3.90 GHz, a 256 KB per-core cache, an 8 MB shared cache, and 8 GB of DDR3-1600 RAM. We ran all our tests on an Ubuntu 12.10 installation running Linux kernel 3.12 (x86_64).

For our evaluation, we selected lighttpd (v1.4.28)—a popular web server—vsftpd (v1.1.0)—a popular FTP server—the OpenSSH Daemon (v3.5)—a popular SSH server—and exim (v4.69)—a popular email server. To benchmark lighttpd, we relied on the Apache benchmark [1] configured to issue 25,000 requests with 10 concurrent connections and 10 request/connection. To benchmark vsftpd, we relied on the pyftpbench benchmark [3] configured to open 100 connections and request 100 1 KB-sized files per connection. To benchmark OpenSSH and exim, finally, we relied on the OpenSSH test suite and a homegrown script repeatedly launching the `sendemail` program [4], respectively. To stress our *StackArmor* prototype in memory-intensive scenarios and better investigate the performance-security tradeoffs, we also considered all the C programs in the SPEC CPU2006 benchmarks. To guide our BR analyzer, finally, we generated the necessary information on the location and size of stack objects from debug symbols. This allowed our BR analyzer to identify (and isolate) 90.7% of the buffers on average across all our programs. We ran all our experiments 11 times—while checking that the CPUs were fully loaded throughout our tests—and reported the median.

Our evaluation answers 4 key questions: (i) *Security*: Is *StackArmor* effective in protecting against both spatial and temporal stack-based attacks? (ii) *Performance*: Does *StackArmor* yield acceptable run-time overhead across all the configurations supported? (iii) *Memory usage*: How much memory does *StackArmor* require? *Multithreading support*: Does *StackArmor* perform and scale well in multithreaded programs?

A. Security Against Spatial Attacks

To evaluate the security guarantees offered by *StackArmor* against spatial attacks, we measured the *attack surface* reduction induced by our protection techniques. Our definition of attack surface quantifies both the number of vulnerable targets (i.e., stack-allocated objects) and the number of offenders (e.g., stack-allocated buffers) in intra-frame and inter-frame attack scenarios.

Intra-frame attack surface reduction. The intra-frame attack surface $S_{\text{intra}}(f)$ of a given function f quantifies the extent to which the N_f objects—noncontrol and control data including the return address—allocated in f's stack frame are exposed to buffer overflow/underflow attacks using any of the B_f stack-allocated buffers in the same frame during the execution. More formally:

TABLE I. MEAN ATTACK SURFACE REDUCTION FOR ALL THE FUNCTIONS WITH STACK-ALLOCATED BUFFERS.

	Intra-frame		Inter-frame	
	Shadowing (Source)	*StackArmor*	Shadowing (Source)	*StackArmor*
lighttpd	100.0%	100.0%	99.0%	99.9%
exim	96.1%	96.8%	97.2%	99.9%
OpenSSH	93.2%	94.4%	94.0%	99.9%
vsftpd	100.0%	100.0%	99.6%	99.9%
SPEC$_{gm}$	91.5%	95.95%	94.6%	99.9%

$$S_{\text{intra}}(f) = \sum_{i=1}^{B_f} \sum_{j=1}^{N_f} \text{canAttack}(i,\ j)\ ?\ 1 : 0$$

When our BR analysis achieves full coverage, *StackArmor*'s protection strategy reduces the original intra-frame attack surface $B_f \cdot N_f$ (induced by a traditional stack organization) to 0 (no buffer can predictably attack other intra-frame objects). In general, however, the intra-frame attack surface reduction is subject to the precision of our BR analysis. Table I shows the mean intra-frame attack surface reduction across all the functions with stack-allocated buffers, also comparing against traditional shadow stack techniques in the ideal case—source-level, with all the buffers remapped.

As shown in the table, *StackArmor* yields a high attack surface reduction across all our test programs—94.4% worst-case reduction for OpenSSH—even successfully isolating all the stack-allocated buffers for lighttpd and vsftpd. Encouragingly, the reduction is comparable and even higher than traditional shadow stack techniques, which typically rely on source code to implement a precise BR analysis strategy but also fail to prevent individual per-frame buffers from attacking one another.

Inter-frame attack surface reduction. To provide a practical definition of inter-frame attack surface, we consider the extent to which an attacker can overflow into the stack frame of all the possible callers of every given function f—an attacker could potentially overflow into any active stack frame, but the spatial predictability guarantees are progressively reduced as we move higher in the call stack and consider all the possible caller-callee combinations. A hypothetical attacker that reliably predicts the caller g of the currently executing function f can potentially rely on all the buffers in f to overflow into any of the objects in the stack frame of g—an attack model completely defeated by *StackArmor*'s protection strategy. In a more generic attack model, in turn, the inter-frame attack surface $S_{\text{inter}}(f)$ of a given function f is subject to the probability p_k of the stack frame of the caller k—i.e., a function in the set of the C_f callers of f—being active on the call stack before that of f. More formally:

$$S_{\text{inter}}(f) = \sum_{i=k}^{C_f} \left[p_k \cdot \sum_{i=1}^{B_f} \sum_{j=1}^{N_k} \text{canAttack}(i,\ j)\ ?\ 1 : 0 \right]$$

To concretely compute $S_{\text{inter}}(f)$ and its reduction induced by *StackArmor* for our test programs, we assume p_k to be a uniform distribution (for simplicity), that is, $p_k = 1/C_f$ for a traditional stack organization and $p_k = 1/R_{max}$ for *StackArmor*, with the swap size R_{max} set to $1,024$ in our experiments. To accurately identify the set of C_f callers for every given f, we performed static callgraph analysis of our test programs using the LLVM compiler framework [50]. Our implementation relies on data structure analysis [51], an efficient context-sensitive and field-sensitive points-to analysis to conservatively analyze function pointers used in indirect calls. Table I reports our findings, comparing the mean inter-frame attack surface reduction across all the functions with stack-allocated buffers induced by *StackArmor* against the reduction induced by traditional (source-level) shadow stack techniques.

As the table shows, *StackArmor* yields a very high inter-frame attack surface reduction across all our test programs—99.9% in all the cases—and consistently higher than traditional source-level shadow stack techniques. The higher reduction reported compared to our intra-frame analysis highlights the effectiveness of *StackArmor* in greatly increasing the randomization entropy in probabilistic attack models. Even when compared to prior source-level stack randomization strategies [18], [47] that introduce random gaps between objects, *StackArmor* yields much stronger randomization guarantees, given that logically contiguous objects (and frames) are guaranteed to be physically nonadjacent in memory by construction.

B. Security Against Temporal Attacks

To evaluate the security guarantees offered by *StackArmor* against temporal attacks, we analyzed the unpredictability of stack frame reuse. To this end, we measured the effectiveness of *StackArmor* in generating a seemingly random sequence of physical stack frame addresses at runtime. For each of the benchmarked programs, we evaluated the randomness of such sequence observed under four configurations: (1) Baseline, (2) *StackArmor* with $R_{max} = 0$, (3) ASLR, and (4) *StackArmor* with $R_{max} = F_{SL} = 1,024$. Our ASLR implementation dynamically generates random inter-frame gaps $g \in [0; 40\ KB]$ using the `rdrand` instruction. This strategy already yields higher entropy than modern ASLR techniques [18], [33], which allow deterministic [18] or periodic [33] stack frame reuse in loops, or use statically generated random inter-frame gaps [33].

To measure randomness of a sequence of stack frames, we conducted a nonparametric hypothesis test for randomness (Bartels' rank test [14]), where we assumed as a null hypothesis that the sequence is generated randomly. For the first two configurations, p-values are consistently lower than $1.9e-7$, while always lower than $9.3e-3$ for ASLR. Summarizing, in the first three configurations, we can reject the null hypothesis at the significance level $\alpha = 0.01$, which confirms highly predictable stack frame reuse. In contrast, *StackArmor*'s default configuration ($R_{max} = F_{SL}$) reported high p-values $\in [0.37; 0.90]$, meaning that we cannot reject the null hypothesis. This result confirms the randomness of the sequence generated by *StackArmor*, yielding truly unpredictable stack frame reuse and strong protection against temporal attacks.

C. Performance

StackArmor's protection strategy introduces run-time performance overhead due to the costs associated with random stack frame allocation—for the logical stack frames and

TABLE II. *StackArmor*-INDUCED BENCHMARK RUN TIME NORMALIZED AGAINST THE BASELINE.

	Basic	+Intra-frame	+UZero
lighttpd	1.06x	1.07x	1.10x
exim	1.01x	1.04x	1.05x
OpenSSH	1.00x	1.01x	1.01x
vsftpd	1.00x	1.01x	1.04x
SPEC_{gm}	1.16x	1.22x	1.28x

the individual per-frame buffers—and zero initialization—for uninitialized (and nonrandomized) per-frame objects. Table II isolates these costs in different configurations, depicting the resulting *StackArmor*-induced benchmark run time normalized against the baseline. The *Basic* column reflects the behavior of *StackArmor* when only mapping the necessary logical stack frames into random physical frames maintained by our allocator. This configuration ultimately results in a basic inter-frame (and return address) protection strategy—16% overhead on SPEC (geometric mean) but generally low overhead (6% in the worst case) across our server programs. The *+Intra-frame* column reflects *StackArmor*'s behavior when also enabling intra-frame protection. As shown in the table, isolating the individual per-frame buffers in independent stack frames only marginally affects the run-time overhead—+6% worst-case overhead increase compared to *Basic* on SPEC (geometric mean). The *+UZero* column, finally, reflects *StackArmor*'s default configuration, with intra-frame+inter-frame protection and zero initialization. As shown in the table, despite the elimination of unnecessary initializers driven by our definite assignment analysis, zero initialization does introduce extra costs, yielding the final overhead of 28% for SPEC (geometric mean) and 10% (in the worst case) across our server programs.

To better investigate the factors contributing to the performance overhead, we also measured the number of cycles and instructions required to complete the SPEC benchmarks across the configurations above and two extra (synthetic) configurations: (i) *Rewriter only*, which isolates the cost to jump into *StackArmor*'s handler and back; (ii) *Rewriter+Allocator*, which isolates the additional stack frame allocation/deallocation costs. Figure 8 presents a number of interesting findings. First, cycles and instructions yield very similar overhead distributions across all the benchmarks, demonstrating that, despite *StackArmor*'s *"locality-unfriendly"* design for securing the stack, the reported overheads mainly stem from new instructions added by our instrumentation rather than poorer cache locality—we observed marginal differences in L1 and L2 code/data cache misses across the different configurations. Second, the *Rewriter only* configuration shows that binary rewriting costs—e.g., saving/restoring registers—dominate the overhead, with the allocation costs (*Rewriter+Allocator* configuration) ranking as a close second. Encouragingly, our optimizations resulted in all the other costs proving generally less significant. For comparison, more naïve protection strategies seeking to instrument *all* the functions with (i) randomized stack frames or (ii) full-coverage zero initialization would result in a worst-case overhead of 719% or 14,700% (perlbench), respectively. Finally, our reported overhead results are significantly skewed by heavily recursive benchmarks such as perlbench—the geometric mean would typically experience a 10% reduction without the latter.

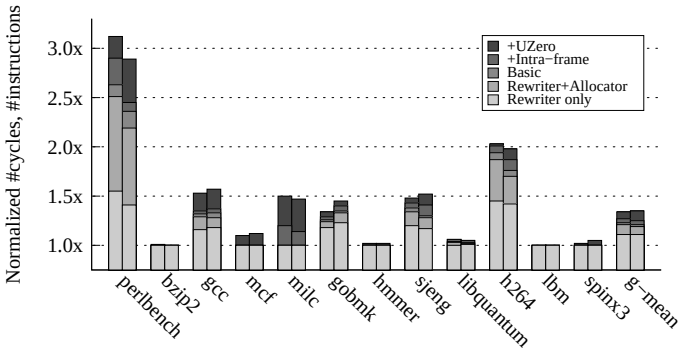

Fig. 8. *StackArmor*-induced SPEC performance overhead.

To understand the impact of our static analyses on the run-time behavior, we also allowed *StackArmor* to report detailed statistics on instrumented functions and call stacks during the execution of our SPEC benchmarks. Table III presents our findings. The first four grouped columns report the total number of functions for each benchmark and those marked as *SP-* or *DA-unsafe*—with their respective ratios compared to *SP-* and *DA-safe* functions detailed in the second three grouped columns. The last two grouped columns, in turn, detail the maximum stack depth (maximum number of physical stack frames allocated by *StackArmor* at any point in time during the execution) and the maximum frame size (maximum size measured across all the physical stack frames allocated by *StackArmor* during the execution).

Our results demonstrate that our static analyzers are effective in significantly reducing the instrumentation overhead. In particular, our SP analyzer reported only 20% of the functions as unsafe on average (geometric mean). Encouragingly, this comes close to the 16% (geometric mean) of the functions reported as unsafe by similar source-level analyses—i.e., `-fstack-protector-strong`, with results reported in the *SP-unsafe (Source)* column—despite operating at the binary level. In addition, over all the *SP-unsafe* functions reported, our BR analyzer was able to correctly identify 92.8% of the total number of buffers found (geometric mean). Our DA analyzer, in turn, reported 52% of the functions as unsafe, while instructing our binary rewriter to zero-initialize only 42% of the stack objects analyzed in those functions on average. The impact of the SP analyzer is particularly evident when examining stack-related statistics. With a small number of *SP-unsafe* functions, only a small fraction of function calls is instrumented and thus assigned a new physical stack frame by *StackArmor*. This is reflected in a worst-case maximum depth value of only 394 frames (perlbench), which never caused our allocator to dynamically increase the soft limit F_{SL} in our default configuration $F_{SL} = 1,024$. The vast majority of function calls, in turn, runs completely uninstrumented, thus efficiently reusing the physical stack frame naturally inherited from the caller. This is reflected in the relatively large values reported for the maximum stack frame size (10.4 KB, geometric mean). Such values never cause our allocator's user-level page fault handler to dynamically increase the soft limit D_{SL} in our default configuration $D_{SL} = 0.1 \cdot D = 205$, while preserving adequate memory usage guarantees (Section V-D).

TABLE III. *StackArmor*-REPORTED STATISTICS FOR THE SPEC CPU2006 BENCHMARKS.

	Functions (#)				Functions (Ratio)			Stack	
	Total	SP-unsafe (Source)	SP-unsafe (*StackArmor*)	DA-unsafe	SP-unsafe (Source)	SP-unsafe (*StackArmor*)	DA-unsafe	Max depth	Max frame (KB)
perlbench	1,885	397	409	1,227	0.21x	0.22x	0.65x	394	8.7
bzip2	112	18	22	48	0.16x	0.20x	0.42x	4	10.2
gcc	5,630	846	972	3,499	0.15x	0.17x	0.62x	48	54.2
mcf	33	1	4	17	0.03x	0.12x	0.52x	80	82.8
milc	244	73	77	91	0.30x	0.32x	0.37x	9	80.5
gobmk	2,690	335	337	1,351	0.12x	0.13x	0.50x	5	5.5
hmmer	548	118	121	324	0.22x	0.22x	0.59x	1	2.8
sjeng	153	41	42	73	0.27x	0.27x	0.48x	2	1.1
libquantum	127	27	31	56	0.21x	0.24x	0.44x	1	2.0
h264ref	599	101	105	391	0.17x	0.18x	0.65x	4	2.7
lbm	29	5	8	14	0.17x	0.28x	0.48x	30	26.3
sphinx3	380	57	59	218	0.15x	0.16x	0.57x	4	21.4
SPEC$_{gm}$	332	54	66	172	0.16x	0.20x	0.52x	9	10.4

Compared to prior stack-based solutions, our reported overheads are generally higher than source-level stack randomization strategies [18], [33]—which, however, offer poorer entropy and isolation guarantees—and comparable to traditional shadow stack strategies [18], [20], [26], [28], [44], [62], [70], [77], [80]—which, however, offer a narrower protection model. In the latter case, a direct benchmark comparison is plausible with the SPEC (INT2000) results reported in [70], which evaluate the impact of a binary translation-based return address shadowing strategy. The latter yields 17% overhead (geometric mean), comparable to our SPEC overhead for *Basic*, which, in turn, also includes inter-frame protection. The small performance difference shows the effectiveness of our stack protection analysis in eliminating the need for pervasive stack instrumentation. Overall, we believe *StackArmor* provides realistic performance for real-world programs, supporting a much more comprehensive strategy than prior solutions in the default configuration, while allowing users to tune the security-performance tradeoff according to their needs.

D. Memory Usage

StackArmor's stack frame allocation strategy translates to higher virtual and physical memory usage. Since virtual memory is a plentiful resource in modern (x86_64) systems, we focus our analysis on the latter. For this purpose, Figure 9 depicts the resident set size (RSS) increase for varying values of the maximum swap size R_{max} during the execution of our benchmarks. To thoroughly evaluate the impact of R_{max}, we configured *StackArmor* with the default number of maximum stack frames and no soft limit (i.e., $F = F_{SL} = 16,384$), allowing arbitrary values of R_{max} during the execution.

The $R_{max}=0$ configuration reflects *StackArmor*'s behavior when in-place frame map randomization is disabled. In this scenario, the RSS increase is only caused by internal fragmentation, since stack frames are allocated at page granularity (0.2-0.5 MB increase across our server programs). When in-place frame map randomization is enabled, in turn, our allocation strategy progressively reduces stack frame reuse, resulting in RSS linearly increasing with R_{max}. For very large values of R_{max} (e.g., $R_{max} = 10,000$), the allocator unrestrictedly draws new stack frames from a large random pool of $10,000$ frames—initially all nonresident in memory—yielding very low frame reuse and thus very high RSS increase (0.7-118.5 MB across our server programs). The differences reported across programs acknowledge variations in the distributions of functions with stack-allocated buffers instrumented by *StackArmor*. For example, a program continuously calling a function f with many in-frame buffers may rapidly circle through all the *StackArmor*'s physical stack frames, resulting in all the F frames being resident in memory. This was nearly the case for some long-running SPEC benchmarks, leading to a consistent worst-case RSS increase for SPEC (1.6-195.1 MB geometric mean).

Note, that while *StackArmor*'s protection strategy exposes an evident randomization entropy-RSS tradeoff—controlled by R_{max}—which generally results in higher RSS than prior stack-based solutions, our default configuration $R_{max}=1,024$—which already provides reasonably high entropy, as earlier experiments demonstrated—results in a worst-case RSS increase for SPEC of only 22 MB (geometric mean).

E. Multithreading Support

To evaluate *StackArmor*'s ability to perform and scale well in multithreaded programs, we selected 3 additional server programs which rely on worker threads to process client requests: Apache httpd (v2.2.23, *mpm_worker_module*)—a popular web server—MySQL (v5.1.65)—a popular database server—and Memcached (v1.4.20)—a popular memory caching server. To evaluate Apache httpd, we relied on the Apache benchmark [1] configured to issue 25,000 requests with T concurrent connections and 10 request/connection. To evaluate MySQL, we relied on the Sysbench OLTP benchmark [5] configured to issue 10,000 transactions using a read-write workload and T concurrent connections. To evaluate Memcached, we relied on the memslap benchmark [2] configured to issue 1,000,000 operations with a T concurrency level. We configured each server program with T worker threads to match the concurrency level induced by the corresponding benchmarks and evaluated *StackArmor*'s performance and memory usage impact for increasing values of $T=[1;100]$.

To evaluate the performance impact, we measured the *StackArmor*-induced benchmark run time normalized against the baseline, using our default configuration at full protection (intra-frame+inter-frame protection and zero initialization enabled).

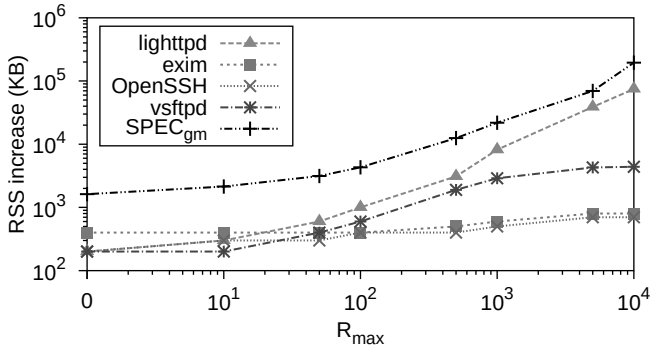

Fig. 9. *StackArmor*-induced RSS increase.

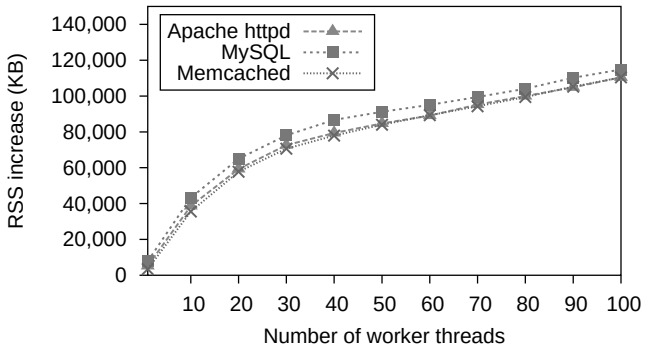

Fig. 10. *StackArmor*-induced RSS increase in multithreaded programs.

Our results reported a nearly constant run-time overhead for increasing values of T across all our programs—i.e., 29-33% (Apache httpd), 35-37% (MySQL), and 13-15% (Memcached). The constant run-time overhead acknowledges the marginal impact introduced by per-thread metadata allocation (and deallocation) performed by our instrumentation, given that common server programs already implement their own thread pooling strategies with worker threads rarely created (or destroyed) on the fast path. In addition, despite its extensive use of C++ code—which may generally result in a larger number of (instrumented) indirect calls due to polymorphic behavior—MySQL encouragingly reported comparable run-time overhead to that of the other server programs.

The generally higher run-time overhead of our multithreaded server programs compared to our nonthreaded test programs, in turn, acknowledges the extra TLS-accessing costs incurred by *StackArmor* to implement a thread-safe allocation and instrumentation strategy. Nevertheless, such costs are strictly bounded even in the heavily threaded $T = 100$ configuration, demonstrating that our design can scale efficiently to a large and uncommon number of threads—Apache httpd, MySQL, and Memcached run with only 25, 16, and 4 threads (respectively) in their default configuration.

To evaluate the memory usage impact, we measured the *StackArmor*-induced RSS increase compared to the baseline, using our default configuration at full protection. Figure 10 depicts our results for an increasing number of threads T. As shown in the figure, the RSS increase grows linearly with the number of worker threads in all our server programs, acknowledging the RSS impact induced by the individual per-thread stack frame pools maintained by our allocator. Nevertheless, *StackArmor*'s ability to adapt R_{max} to the number of active threads results in a gentle slope and a RSS increase of only 115 MB in the worst case (MySQL, with $T = 100$). This confirms that *StackArmor* can scale fairly well even to heavily threaded programs, while preserving reasonable randomization entropy guarantees at runtime—$R_{max} = \{299, 474, 878\}$ in the default configuration of Apache httpd, MySQL, and Memcached (respectively).

VI. Related Work

Protection from stack-based vulnerabilities. Early stack protection systems typically rely on canary values guarding all the per-frame objects [45] or only the return address [31]—a technique still in use in modern compilers [7]—to verify the stack frame integrity upon function exit. Unlike *StackArmor*, canary-based techniques can only protect from spatial (smashing) attacks and are extremely susceptible to information leakage. Shadow stack techniques isolate the return address [26], [28], [62], [70], [77] or all the per-frame nonbuffers [18], [20], [44], [80] from potentially vulnerable per-frame buffers to limit the exploitation power of buffer overflows/underflows. Unlike *StackArmor*, such techniques are ineffective against temporal attacks and typically limited to *buffer-to-nonbuffer* spatial attacks. Traditional Address Space Layout Randomization (ASLR) techniques [6], [17], [76] randomize the base address of the stack to make the address of the individual stack objects unpredictable. This strategy is insufficient to disrupt attacks that solely rely on the relative distance or reuse between stack objects. To overcome this limitation, fine-grained ASLR techniques [18], [33] introduce small random gaps between stack frames and between buffer and nonbuffer stack objects [33]. Unlike *StackArmor*, this strategy alone provides no isolation, offers limited entropy to defend against sophisticated (e.g., guessing or spraying) attacks, and is susceptible to information leakage when using static gaps [33]. Finally, while sharing many of the abstractions used in prior ASLR and shadow stack techniques—including isolation, randomization, and buffer detection—*StackArmor* is the first comprehensive stack protection technique for binaries.

Protection from generic memory errors. Generic memory error protection systems have been the subject of a vast body of research in the last decade. Unlike *StackArmor*, many popular techniques—including data flow integrity [23], write integrity testing [10], bounds checkers [11], [40], [81], and memory safe environments [35], [41], [42], [48], [56]—all require access to the source code or recompilation. At the binary level, many techniques such as Control-Flow Integrity (CFI) [8], [82], Instruction Set Randomization (ISR) [61], ROP protection systems [25], [60], and Dynamic Taint Analysis (DTA) [29], [58] can stop various control-flow diversions, but, unlike *StackArmor*, offer no protection against corruption of noncontrol data. In addition, techniques like DTA [29], [58], multivariant execution [32], [66], [67], and memory error checking tools such as Valgrind [57] and Dr. Memory [21] typically incur performance overheads of an order of magnitude or more. Finally, recent binary-level protection systems such as *BinArmor* [72] rely on dynamic data structure reverse engineering techniques to instrument legacy binaries, with protection and accuracy guarantees subject to the coverage of the dynamic analysis. In

addition, *BinArmor* [72] is tailored to buffer overflows and, unlike *StackArmor*, cannot address temporal attacks.

Protection from temporal attacks. Prior temporal attack protection systems focus on both use-after-free and uninitialized read vulnerabilities. Systems in the former category are generally targeted towards heap-based vulnerabilities, with techniques ranging from garbage collection [19], [63] to secure allocation [9], [15], [42], [54], [59] and dynamic memory checking [21], [22], [57], [68]. The first two classes are not directly applicable to the stack, although *StackArmor* does borrow ideas from prior secure heap allocator designs. Similar to many secure allocators [15], [54], [59], *StackArmor* enforces a fully randomized allocation strategy, with a sparse page layout [59] and a single object per page(s) [54] to enforce probabilistic memory safety. Unlike type-safe allocation strategies [9], [42], in turn, *StackArmor* does not allow type-safe memory reuse because doing so is more expensive and makes uninitialized stack reads more predictable. Compared to prior designs, however, *StackArmor*'s allocator is much simpler (frames are preallocated) and more efficient (the self-managing frame map eliminates expensive bookkeeping and lookups), thanks to the inherently bounded and dynamic nature of the stack. Dynamic memory checking tools [21], [22], [57], [68], in turn, can generally operate on the stack, offer pseudo-deterministic detection guarantees, and also often detect uninitialized reads [21], [57], [79]. Unlike *StackArmor*, however, they typically incur very high overhead. Other more lightweight uninitialized read detection techniques exist, but they generally sacrifice precision for performance [15], [47]. While *StackArmor* can only protect (and not detect) against uninitialized reads, it thwarts arbitrary attacks using a combination of probabilistic (randomization) and deterministic (zero initialization) strategies. The latter, in turn, is directly comparable to secure deallocation [27], which zeroes out memory blocks at deallocation time. Even though this approach can also protect against nonreuse-based temporal violations, it is very expensive on the stack [27]. *StackArmor* exploits the allocation context to reduce the number of initializations (and thus the overhead) using definite assignment analysis—implemented at the binary level in contrast to prior source-level strategies [35], [41], [42], [46], [48].

VII. Conclusion

Nearly two decades after the first stack smashing attack, performance concerns still induce modern compilers to ship with weak stack protection mechanisms, ultimately resulting in binaries being left at the mercy of the attackers. This paper presented *StackArmor*, a more comprehensive stack protection technique which offers a practical solution to this problem. Unlike prior systems, *StackArmor* can efficiently protect against arbitrary spatial and temporal stack-based attacks, operates entirely at the binary level, and supports policy-driven defenses to allow end users to tune the performance-security tradeoff. To fulfill its goals, *StackArmor* abandons the traditional stack organization and relies on a combination of randomization, isolation, and zero initialization—efficiently balanced using static analysis—to create the illusion that stack objects are drawn from a fully randomized space. Our experimental results confirm that *StackArmor* is practical, efficient, and provides more comprehensive protection than all the prior binary- and source-level solutions.

Acknowledgment

The authors would like to thank the anonymous reviewers for their comments. This work has been supported by the Rosetta project funded by ERC (ERC Starting Grant #258108) and by the Re-Cover project funded by NWO.

References

[1] "Apache benchmark," http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] "memslap," http://docs.libmemcached.org/bin/memslap.html.

[3] "pyftpdlib," https://code.google.com/p/pyftpdlib.

[4] "SendEmail," http://caspian.dotconf.net/menu/Software/SendEmail.

[5] "SysBench," http://sysbench.sourceforge.net.

[6] "ASLR: Leopard versus Vista," http://blog.laconicsecurity.com/2008/01/aslr-leopard-versus-vista.html, 2008.

[7] "Switching from "-fstack-protector" to "-fstack-protector-strong" in Fedora 20," http://fedorahosted.org/fesco/ticket/1128, 2013.

[8] 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, 2009.

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

[10] 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.

[11] 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.

[12] K. Anand, M. Smithson, A. Kotha, K. Elwazeer, and R. Barua, "Decompilation to compiler high IR in a binary rewriter," University of Maryland, Tech. Rep., 2010.

[13] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *Proc. of the 8th ACM European Conference on Computer Systems*, ser. EuroSys'13, 2013.

[14] R. Bartels, "The rank version of von Neumann's ratio test for randomness," *J. Am. Statist. Assoc.*, vol. 77, no. 377, 1982.

[15] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2006.

[16] A. R. Bernat and B. P. Miller, "Anywhere , Any-Time Binary Instrumentation," in *Proc. of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, ser. PASTE'11, 2011.

[17] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a board range of memory error exploits," in *Proc. of the 12th USENIX Security Symp.*, 2003.

[18] 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.

[19] H.-J. Boehm, "Bounding space usage of conservative garbage collectors," in *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 2002.

[20] P. Broadwell, M. Harren, and N. Sastry, "Scrash: A system for generating secure crash information," in *Proc. of the 12th USENIX Security Symp.*, 2003.

[21] D. Bruening and Q. Zhao, "Practical memory checking with Dr. Memory," in *Proc. of the Ninth Int'l Symp. on Code Generation and Optimization*, 2011.

[22] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proc. of the 2012 Int'l Symp. on Software Testing and Analysis*, 2012.

[23] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '06, 2006.

[24] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proc. of the Second Asia-Pacific Workshop on Systems*, 2011.

[25] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "Ropecker: A generic and practical approach for defending against ROP attacks," in *Proc. of the 21th Annual Network and Distributed System Security Symp.*, 2014.

[26] T.-C. Chiueh and F.-H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *Proc. of the 21st Int'l Conf. on Distributed Computing Systems*, 2001.

[27] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation," in *Proc. of the 14th USENIX Security Symp.*, 2005.

[28] M. L. Corliss, E. C. Lewis, and A. Roth, "Using DISE to protect return addresses from attack," in *Proc. of the Workshop on Architectural Support for Security and Anti-Virus*, 2004.

[29] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *Proc. of the 20th ACM Symp. on Oper. Systems Prin.*, 2005.

[30] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static detection of vulnerabilities in x86 executables," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006.

[31] 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.

[32] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity," in *Proc. of the 15th USENIX Security Symp.*, 2006.

[33] A. K. Cristiano Giuffrida and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proc. of the 21st USENIX Security Symp.*, 2012.

[34] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *Proc. of the 35th IEEE Symp. on Security and Privacy*, 2014.

[35] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *Proc. of the 21st ACM Symp. on Oper. Systems Prin.*, 2007.

[36] CVE-2008-0063, "Kerberos stack-based information leak," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-0063, 2008.

[37] CVE-2010-0262, "Microsoft office arbitrary code execution vulnerability due to uninitialized stack variable," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-0262, 2010.

[38] CVE-2012-5976, "Asterisk stack-based buffer overflow," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5976, 2013.

[39] CVE-2013-4368, "Xen stack-based information leak," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4368, 2013.

[40] 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.

[41] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: Enforcing alias analysis for weakly typed languages," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2006.

[42] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory safety without runtime checks or garbage collection," in *Proc. of the ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems*, 2003.

[43] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory safety without garbage collection for embedded applications," *ACM Trans. Embed. Comput. Syst.*, 2005.

[44] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proc. of the Seventh Symp. on Operating Systems Design and Implementation*, 2006.

[45] H. Etoh and K. Yoda, "ProPolice - Improved stack smashing attack detection," *IPSJ SIGNotes Computer Security*, vol. 14, 2001.

[46] N. G. Fruja, "The correctness of the definite assignment analysis in C#," in *Proc. of the Second Int'l Workshop on .NET Technologies*, 2004.

[47] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum, "Practical automated vulnerability monitoring using program state invariants," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2013.

[48] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proc. of the USENIX Annual Technical Conf.*, 2002.

[49] Y. Joey, "Merge stack alignment branch," https://gcc.gnu.org/ml/gcc-patches/2008-04/msg00349.html, 2008.

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

[51] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2007.

[52] M. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for Linux," in *Proc. of the Int'l Symp. on Performance Analysis of Systems and Software*, 2010.

[53] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proc. of the 17th Annual Network and Distributed System Security Symposium*, ser. NDSS'10, 2010.

[54] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn, "Archipelago: Trading address space for reliability and security," in *Proc. of the 13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[55] M. Matz, J. Hubi, A. Jaeger, and M. Mitchell, "System V Application Binary Interface. AMD64 Architecture Processor Supplement." 2013.

[56] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 2002.

[57] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. of the Third Int'l ACM SIGPLAN Conf. on Virtual Execution Environments*, 2007.

[58] J. Newsome and D. Song, "Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software," in *Proc. of the Network and Distributed Systems Security Symposium*, ser. NDSS'05, 2005.

[59] G. Novark and E. D. Berger, "DieHarder: Securing the heap," in *Proc. of the 17th ACM Conf. on Computer and Commun. Security*, 2010.

[60] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proc. of the 22nd USENIX Security Symp.*, 2013.

[61] G. Portokalidis and A. D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," in *Proc. of the 26th Annual Computer Security Applications Conf.*, 2010.

[62] M. Prasad and T. cker Chiueh, "A binary rewriting defense against stack-based buffer overflow attacks," in *Proc. of the USENIX Annual Technical Conf.*, 2003.

[63] J. Rafkind, A. Wick, J. Regehr, and M. Flatt, "Precise garbage collection for C," in *Proc. of the Eighth Int'l Symp. on Memory management*, 2009.

[64] C. Rosier, "Support for dynamic stack realignment + VLAs for x86," http://lists.cs.uiuc.edu/pipermail/llvm-commits/Week-of-Mon-20120702/146062.html, 2014.

[65] B. G. Roth and E. H. Spafford, "Implicit buffer overflow protection using memory segregation," in *Proc. of the Sixth Int'l Conf. on Availability, Reliability and Security*, 2011.

[66] 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 Int'l Conf. on Complex, Intelligent and Software Intensive Systems*, 2008.

[67] 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.

[68] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *Proc. of the USENIX Annual Technical Conf.*, 2012.

[69] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. of the 14th ACM Conf. on Computer and Communications Security*, 2007.

[70] S. Sinnadurai, Q. Zhao, and W.-F. Wong, "Transparent runtime shadow stack: Protection against malicious return address modifications," University of Singapore, Tech. Rep., 2004.

[71] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Proc. of the Network and Distributed System Symp.*, 2011.

[72] A. Slowinska, T. Stancescu, and H. Bos, "Body Armor for binaries: Preventing buffer overflows without recompilation," in *Proc. of USENIX Annual Technical Conf.*, 2012.

[73] M. Smithson, K. Anand, and A. Kotha, "Binary rewriting without relocation information," University of Maryland, Tech. Rep. November, 2010.

[74] A. Sotirov, "Heap feng shui in JavaScript," in *Black Hat Europe*, 2007.

[75] B. D. Sutter, B. D. Bus, and K. D. Bosschere, "On the static analysis of indirect control transfers in binaries," ser. PDPTA'00, 2000.

[76] P. Team, "Overall description of the PaX project," http://pax.grsecurity.net/docs/pax.txt, 2008.

[77] Vendicator, "Stack Shield: A "stack smashing" technique protection tool for Linux," http://www.angelfire.com/sk/stackshield/info.html, 2001.

[78] R. Wartell, Y. Zhou, and K. Hamlen, "Differentiating code from data in x86 binaries," in *Proc. of the 2011 European Conference on Machine Learning and Knowledge Discovery in Databases*, ser. ECML PKDD'11, 2011.

[79] D. Ye, Y. Sui, and J. Xue, "Accelerating dynamic detection of uses of undefined values with static value-flow analysis," in *Proc. of the 11th Int'l Symp. on Code Generation and Optimization*, 2014.

[80] 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 Applications Conf.*, 2006.

[81] 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.

[82] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proc. of the 34th IEEE Symp. on Security and Privacy*, 2013.

[83] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proc. of the 22nd USENIX Security Symposium*, 2013.