SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities

Alyssa Milburn
Vrije Universiteit Amsterdam
amilburn@zall.org

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

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

***Abstract*—Usage of uninitialized values remains a common error in C/C++ code. This results not only in undefined and generally undesired behavior, but is also a cause of information disclosure and other security vulnerabilities. Existing solutions for mitigating such errors are not used in practice as they are either limited in scope (for example, only protecting the heap), or incur high runtime overhead.**

In this paper, we propose SafeInit, a practical protection system which hardens applications against such undefined behavior by guaranteeing initialization of *all* values on the heap and stack, every time they are allocated or come into scope. Doing so provides comprehensive protection against this class of vulnerabilities in generic programs, including both information disclosure and re-use/logic vulnerabilities.

We show that, with carefully designed compiler optimizations, our implementation achieves sufficiently low overhead (<5% for typical server applications and SPEC CPU2006) to serve as a standard hardening protection in practical settings. Moreover, we show that we can effortlessly apply it to harden non-standard code, such as the Linux kernel, with low runtime overhead.

I. Introduction

The use of uninitialized memory in C/C++ programs introduce vulnerabilities that are popular among attackers to manipulate a program's control flow or to disclose information. In addition to the obvious issue of revealing sensitive data, the exposure of metadata has become a more prominent problem in recent years, since information disclosure increasingly becomes an essential prelude to successful exploits (e.g., to circumvent ASLR or other hardening methods) [56]. Unfortunately, concerns about the performance overhead have made compiler writers reluctant to adopt strong mitigations against this attack vector.

Languages such as Java and C# ensure the *definite assignment* of variables, requiring them to be initialized on all possible paths of execution. Unfortunately, C and C++ do not enforce this property. As a result, the vast body of existing C/C++ code, which includes many runtimes and libraries for safer languages, is potentially vulnerable to uninitialized read attacks. Today's compiler warnings and static analysis tools flag only a small subset of these uninitialized reads, and worse, the common occurrence of false positives in the warnings means that programmers often ignore them altogether. Given the growing popularity of uninitialized reads in real-world exploits [21], [53], [8], [50], the current lack of comprehensive protection is concerning. In this paper, we show that automatic initialization of *all* values on the heap and stack at allocation time is possible with minimal performance penalties.

Worryingly, C/C++ compilers can even introduce new vulnerabilities when taking advantage of the fact that reading uninitialized memory is 'undefined behavior'; the optimizations applied by modern compilers can remove sanity checks or other code in such circumstances [63]. Worse, recent research [44] has also shown that many programmers are unaware of these dangerous consequences. For C/C++ programs running in production systems, there are few options for preventing attacks that exploit uninitialized read errors. Solutions such as valgrind [54] and MemorySanitizer [58], in widespread use during the development process, are much too expensive for use in production systems, even when using complex data-flow analysis to reduce this overhead [65].

Clearing memory: The obvious mitigation for this problem is to always clear memory. For instance, Chow et al. [11] proposed to clear all memory at deallocation time. However, they only obtained acceptable overhead for heap allocations – not for the high frequency allocations and deallocations on the stack. Moreover, the solution fails to address the problem of undefined behavior. The PaX project [49] offers a limited but very practical solution in the form of Linux kernel patches which protect against common uninitialized value errors, including gcc plugins. Recently, and concurrent to our work, UniSan [38] proposed more comprehensive protection than this against a narrower threat—information disclosure from the Linux kernel—using data-flow analysis to initialize memory and variables which might be disclosed to an attacker. While both of these solutions provide acceptable overhead, neither provide a complete solution for uninitialized values, and are currently applicable to only the Linux kernel.

In this paper, we describe a comprehensive and practical solution for mitigating these errors in generic programs, by adapting the toolchain to ensure that all stack and heap allocations are *always* initialized. SafeInit is implemented at a compiler level, where low-overhead static analysis and optimizations are available, and can be enabled using a single compiler flag. We show that the overhead can be reduced to acceptable levels, by applying a set of carefully designed

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23183

optimizations; for example, these more than halve the overhead of SafeInit on SPEC CINT2006 from 8% down to <4% (with the remaining overhead largely due to excessively complex code, which can be resolved using minimal-effort annotations).

Summarizing, our contributions are:

- We propose SafeInit, a compiler-based solution which – together with a hardened allocator – automatically mitigates uninitialized value reads by ensuring initialization, both on the stack and on the heap.
- We present optimizations which reduce the typical overhead of our solution to minimal levels (<5%), and are straightforward to implement in modern compilers.
- We discuss our prototype implementation of SafeInit, based on clang and LLVM, and show that it can be applied to the majority of real-world C/C++ applications without any additional manual effort.
- We evaluate our work on CPU-intensive (including SPEC CPU2006) and I/O intensive (server) applications, as well as the Linux kernel, and verify that real-world vulnerabilities are successfully mitigated.

In summary, we argue that SafeInit provides a comprehensive and practical solution to a serious real-world problem, show that it provides significant advantages compared to existing techniques and tools, and demonstrate that it offers acceptable levels of overhead. We believe this system is sufficiently practical to make it useful in production systems, with overhead below the levels typically demanded for industry adoption [60], and hope to see it become a standard ingredient of the hardening transformations offered by modern compilers.

II. Threat Model

Uninitialized read errors occur when a variable, or memory, is used without having first been initialized. This can occur after a stack variable comes into scope, or after heap memory has been allocated. We consider an attacker seeking to exploit any of the vulnerabilities caused by such reads of uninitialized values, including information disclosure and use of unintended values (such as function pointers). We assume that such potential attackers have a copy of all binaries in use, and are thus aware of details such as the exact stack layout chosen by the compiler.

We assume the program has been already hardened against other classes of vulnerabilities using existing (e.g., memory safety) defenses. Although mitigating uninitialized value vulnerabilities can probabilistically mitigate some vulnerabilities caused by other temporal (such as pointer use-after-free) and spatial errors (such as out-of-bounds reads), there are existing low-impact solutions – such as baggy bounds checking [2] – which provide superior defenses against such attacks, and we do not consider them in our threat model.

We also only consider C/C++ code. Extending this work to similar languages should be possible (as shown, for example, by existing compiler functionality for local variables in Fortran), but in particular, custom assembly-language routines fall outside the scope of our work.

III. Background

Memory is constantly reallocated, and thus reused, in almost all applications. On the stack, function activation frames contain data from previous function calls; on the heap, allocations contain data from previously-freed allocations. Issues with uninitialized data arise when such data is not overwritten before being used, extending the lifetime of the old data beyond the point of the new allocation.

Many variables are clearly initialized before they are used; as an example, consider an integer counter used only in a for loop, which is explicitly assigned a new value for every iteration of the loop. We can trivially see that such a variable is always initialized before it is used.

On the other hand, the initialization state of a variable which is only used if a complicated conditional is true may depend itself on other conditionals, resolving of which would require executing large portions of the program – or at least extensive optimization and analysis.

Memory may also only be partially initialized; structures and union types in C are often deliberately incompletely initialized, and for simplicity or performance reasons, arrays are often allocated with larger sizes than (initially) necessary to store their contents.

In practice, reuse of memory is not only common, but also desirable for performance reasons [17]. When it is unclear whether a variable will be initialized before it is used, the only practical and safe approach is to initialize it in all cases.

A. Sensitive data disclosure

The most obvious danger of information disclosure due to uninitialized data is the disclosure of directly sensitive data, such as encryption keys, passwords, configuration information and the contents of confidential files. Chow et al. have discussed [10] that data lifetimes can last far longer than we would expect, and that many unintentional copies of data may be made.

In fact, even when all copies of such data are apparently explicitly cleared, problems persist. Many programs call `memset` to clear sensitive data; unfortunately, if the data is no longer valid and thus no longer used after that point, such calls can be (and are) optimized away by compilers. Common 'workarounds' which attempt to hide these calls from compiler analysis are often optimized by ever-improving compiler analysis passes, and alternative functions (such as `memset_s` and `explicit_bzero`) which compilers are prohibited from optimizing away are not yet commonly available.

If the use of uninitialized data in a program is not directly influenced by untrusted input, it is tempting to conclude that the security consequences of these classes of issues is otherwise minimal. However, experience has shown that a wide range of potential attack vectors must be considered, and this varied attack surface means that all uninitialized data vulnerabilities should be taken seriously.

One illustrative example was a vulnerability [29] in the PostScript font rendering on Windows, caused by failure to initialize a temporary buffer which could be read by font bytecode. By providing a font which rendered glyphs based

on the contents of this buffer, JavaScript in a web browser could disclose memory by reading back rendered pixels.

Similarly, information disclosure from kernels to userspace programs, or from hypervisors to guest virtual machines, is a common and serious issue [8]. Containers and virtual machines running code from untrusted parties, or as a vital layer of sandboxing from untrusted software such as JavaScript in web browsers, are now a standard component of many systems. As such, even code such as device driver interfaces and emulated devices must be free of security issues.

B. Bypassing security defenses

Even where software does not make use of any seemingly 'sensitive' data, or such data is sufficiently isolated to avoid the possibility of it being disclosed due to uninitialized data issues, many modern software defenses depend on the secrecy of sensitive *meta*data, and so information disclosure is still a critical flaw. Stack 'canaries' provide an obvious example; their protection relies on the canary value remaining secret.

Defenses such as address space layout randomization (ASLR) [57] depend on the secrecy of pointers in general, and since this is typically done by randomizing only one base address, an attacker only needs to disclose a single pointer to defeat the protection entirely. Such pointers may be code, stack or heap pointers, and these are typically stored on both the stack and the heap, so uninitialized value errors provide a rich source of the pointer disclosures required to thwart such information hiding.

Heap allocators may disclose heap metadata if they store it inline, while other allocators are careful to maintain this separately. Some allocators re-use memory almost immediately, which is important in modern high-performance allocators which maintain per-thread pools. Notably, often disclosing just a single byte of such metadata is enough to significantly reduce security guarantees; disclosing 8 bits of a pointer can significantly reduce ASLR entropy [55].

Another example is LLVM's SafeStack defense, based on work by Kuznetsov et al. [32], which attempts to provide protection against arbitrary memory write vulnerabilities to the stack, by hiding the pointer to the 'safe' (machine) stack, and moving unsafe variables to a separate stack. However, at the time of writing, it can be trivially defeated by disclosing the contents of an uninitialized stack buffer; these contents are within the bounds of a stack variable, and so remain on the 'safe' stack, but almost always contain safe stack pointers.

C. Software exploitation

Other vulnerabilities caused by uninitialized data allow attackers to directly hijack control flow. For example, virtual function calls may be made using uninitialized local variables which are meant to contain (pointers to) C++ objects. Potential attackers can often influence the contents of the stack or heap by causing specific computation or a specific set of calls to be made [40]; their ability to control the contents of a specific part of memory depends on factors which seem difficult to predict, such as the results of the compiler's stack slot allocation algorithm, but which can often easily be determined by an attacker with access to the compiled binary.

This is far from a new problem; Microsoft described an arbitrary write vulnerability due to an uninitialized stack variable in Microsoft Excel in 2008 [45], and in 2010, Kees Cook disclosed [12] an arbitrary Linux kernel memory write vulnerability caused by an uninitialized structure on the stack.

A common mistake is to fail to initialize variables or buffers on the execution path taken when an error is encountered. For example, Samba had a vulnerability[61] caused by failure to check the error value returned by a function before using a pointer value which was only initialized in the error-free path. Similarly, a bug in Microsoft's XML parser[1] made a virtual function call using a pointer stored in a local variable which was not initialized on all execution paths. By 'spraying' the stack with pointers using JavaScript, attackers could control the contents of the memory where the variable was stored, and exploit this vulnerability from within a web browser.

It is clear that all of these vulnerabilities must be taken seriously, and that preventing information disclosure addresses only a subset of uninitialized value vulnerabilities.

D. Detection tools

Rather than attempting to mitigate uninitialized value errors, there are a variety of tools which attempt to *detect* them during the development process, allowing them to be manually corrected by the programmer. Compiler warnings and static analysis tools are limited by their nature [33], and often themselves contain bugs [59]. Reporting all values which *may* be uninitialized is counter-productive, since when presented with more than a handful of false positives, programmers will start to simply ignore the warnings. This means that any useful tool is forced to err on the side of caution in reporting; the authors of one commercial static analysis tool, Coverity, state that [6] "when forced to choose between more bugs or fewer false positives we typically choose the latter".

More fundamentally, compiler warnings and detection tools only report problems, rather than fixing them. This can lead to incorrect and dangerous fixes, as illustrated by the 2008 Debian OpenSSL flaw [66]; a patch 'fixing' a valgrind warning about uninitialized data also removed the code which added entropy during key generation, leading to the widespread generation and use of predictable keys.

E. Stack variables

A function stack frame contains copies of local variables which are either too large to be stored in registers, or which have their addresses taken (for example, to be passed to another function). The stack also generally contains spilled copies of other local variables and compiler-generated temporary variables, as well as function arguments, frame pointers and return addresses. Given the constant re-use of stack memory, these frames provide a rich source of sensitive data.

Modern compilers use sophisticated algorithms for register and stack frame allocation [42], and both temporaries and variables with non-overlapping lifetimes can be assigned the same portion of a stack frame (or registers). This reduces memory usage and improves cache locality, but means that even clearing registers and stack frames before/after a function call is insufficient to avoid all potential uninitialized variables.

```
main() {
  int x;
  printf("%d", x);
}
```

(a) C code

```
define @main() {
  %x = alloca i32, align 4
  %0 = load i32, i32* %x
  call @printf(..., i32 %0)
}
```

(b) LLVM bitcode, before mem2reg

```
define @main() {
  call @printf(..., i32 undef)
}
```

(c) LLVM bitcode, after mem2reg

Fig. 1. LLVM transforms uninitialized reads into `undef` values early in the optimization process; later passes cannot recover the information removed in (c)

```
for (i = 0; i < n; ++i) {
  int x;
  if (i == 0) {
    x = getSecretValue();
  }
  doSomething(x);
}
```

Fig. 2. Memory re-use for local variables in a loop; doSomething will be passed the secret value in *all* loop iterations, not just the first

Even when stack coloring is disabled, and every local variable is allocated a separate portion of an initialized stack, uninitialized variables can still be a problem. Memory re-use in a function stack frame also occurs inside loops, such as in Figure 2; here, a variable is initialized during the first iteration of a loop, but not in later iterations.

F. Undefined behavior

Undefined behavior [63] occurs when a C/C++ program fails to follow the rules imposed by the language. Most importantly in the context of our discussion, this is the case when code reads uninitialized stack variables, or even uninitialized heap allocations. The C/C++ standards state that permissible consequences of undefined behavior includes the compiler's code generation "ignoring the situation completely with unpredictable results", but many programmers are unaware [44] of the fact that these consequences have more serious consequences for their compiled binaries than simply producing code which will read potentially uninitialized data.

This is not a merely theoretical problem, but a serious practical issue; to enable the maximum number of optimizations, especially in code which may be expanded from templates and macros and eventually largely be discarded as unreachable, modern compiler transforms (such as those used by LLVM [34]) take advantage of this undefined behavior on a large scale. Unfortunately, such transformations may interpret undefined values (and thus, also uninitialized values) as *any* value which makes optimizations more convenient, even if this makes program logic inconsistent. These situations often only become apparent after other compiler transformations have already been applied, and *cannot* be detected by dynamic analysis tools, since they rely on the machine code which has been generated after this process.

Even a very basic level of compiler optimizations will cause problems with such code. For example, Figure 1 shows clang/LLVM generating `undef` values due to an uninitialized local variable. This is caused by the `mem2reg` pass, which converts local variables to SSA form; this transformation is a prerequisite for almost all other compiler optimization passes or analyses. This illustrates why hardening transformations must be run *before* any other optimizations, and most importantly, why they must be performed in the compiler itself.

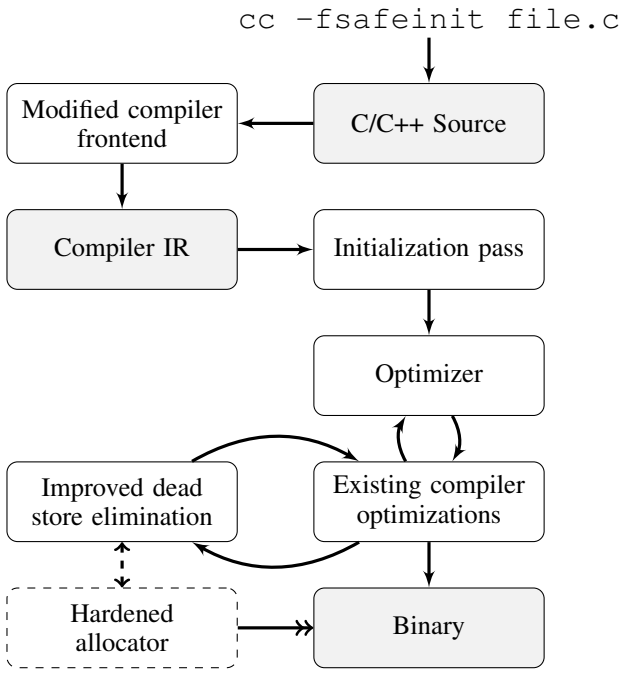

Fig. 3. High-level overview of SafeInit

This also limits the analysis available to such transformations; any analysis must be performed on the initial, unoptimized code from the frontend. This not only reduces the accuracy of any analysis, but also has a serious impact on performance; these problems are particularly troublesome for interprocedural analysis, and in any case, other functions may be unavailable until link-time optimization (LTO) is performed. Attempting to delay all optimization until LTO time has a severe impact on compilation time, making it far less practical.

IV. Overview

SafeInit mitigates uninitialized value problems by forcing the initialization of both heap allocations (after their allocations) and all stack variables (whenever they come into scope). This is done by modifying the compiler to insert initialization calls directly at all such points.

In order to provide both practical and comprehensive security, this instrumentation must be done within the compiler itself. SafeInit can be enabled by simply passing an additional hardening flag during the compilation process. As can be seen

in Figure 3, this enables an additional compiler pass which adds the necessary initialization.

A naive initialization approach would lead to excessive runtime overhead, and an important element of our system is a customized hardened allocator. This is able to avoid initialization in many cases by taking advantage of extra information, combined with our compiler instrumentation.

Finally, the SafeInit optimizer provides non-invasive transformations and optimizations which we run alongside existing compiler optimizations (themselves modified where necessary), as well as the final component, an extension of existing 'dead store elimination' optimizations. These build on top of our initialization pass and allocator, performing more extensive removal of unnecessary initialization code, demonstrating that the runtime overhead of our solution can be minimized.

Perhaps most importantly, SafeInit is practical to implement in modern compilers. Our system requires minimal changes and is non-invasive; no new analyses are required, and the extended optimizations we propose are not specific to SafeInit. Our design is also compatible with recent developments such as ThinLTO [28], where later optimization passes may not have access to the IR/bitcode of called functions.

V. Mitigating uninitialized values

A. Initialization pass

SafeInit initializes all local variables before their first use, treating the point at which variables come into scope (for example, in a loop) as a newly-allocated variable. We propose inserting initialization code at all such points where a stack variable comes into scope; the necessary scope information is provided by the compiler frontend.

Specifically, SafeInit's stack hardening pass modifies the compiler's intermediate representation (IR) of the code being compiled and inserts a store instruction (ideally, a `memset` builtin/intrinsic) after every variable comes into scope; other optimizations can later remove or simplify these. This clears all of the memory allocated for the variable, including any padding within a structure, or between array elements.

B. Hardened allocator

SafeInit's hardened allocator ensures that all newly-allocated memory is cleared to zero before being returned to the application. We do this in the allocator for safety – we override *all* heap allocation functions to ensure these hardened allocator functions are always used – as well as to improve performance by taking advantage of the extra information available to the allocator.

Importantly, the compiler is aware that our hardened allocator is in use; any code using allocated memory is no longer making use of undefined behavior, and cannot be modified or removed by the compiler.

All memory pages allocated by the operating system kernel are already cleared to zero, and so allocators can take advantage of this and avoid clearing such pages. Although the overhead of keeping track of this for small allocations is excessive, and so small allocations must always be cleared at allocation time, large allocations are at least several pages in size, and often allocated by using mmap directly.

```
char err_msg[MAX_MSG];
...
if (error) {
  setErrMessage(err_msg);
  printf(err_msg);
  return error;
}
```

Fig. 4. Typical code using error message buffer; the buffer need not be initialized unless the branch here is taken.

Modern operating systems also provide support for clearing regions of such memory directly (such as MADV_DONTNEED on Linux) by releasing the underlying pages; while this comes with potential performance downsides [31], it is already used by modern allocators to minimize memory usage, and is ideal for our needs. By ensuring that large allocations are always released, we can ensure they will be cleared even if they are reused for another allocation, without incurring any performance penalty for clearing areas of memory which will not be used.

Our allocator also exports non-initializing variants of allocation functions; the requested memory is not zeroed when these are called, but the gap between the requested allocation size and the true allocated size is *always* cleared. An application may later make a `realloc` call which can lead to the re-use of this space, and keeping track of individual allocation sizes consumes more memory and leads to excessive runtime overhead (we observed overheads of $>5\%$).

C. Optimizer

Our optimizer design provides several efficient and practical optimizations which improve the performance of SafeInit while being efficient and non-invasive. The primary goal of the optimizer is to make simple changes which will allow the many other standard optimizations available in modern compilers to remove any unnecessary initializations. We hope that SafeInit will become a standard hardening technique, and so it needs to be as practical as possible; in particular, we need to avoid adding complex or invasive analysis.

1) Sinking stores: Ideally, stores to local variables should be as close as possible to their uses. This is important for cache locality, and for minimizing the memory usage of stack frames; minimizing the lifetime of a variable allows stack coloring algorithms to allocate stack frame space more efficiently.

Our optimizer attempts to move our initialization stores to the dominating point of the uses of a variable. Importantly, this also avoids unnecessary initialization; variables which are unused in certain paths need only be initialized in the paths where they are used. A common example is where variables are only used in error paths, such as with the code in Figure 4. This code path is not executed during normal execution, and we do not need to initialize the buffer until we reach a path in which it will be used.

If this dominating point of the uses of a variable is reachable from itself, and it does not go out of scope when

```
int buf[50];
for (int i = 0; i < 50; ++i)
  buf[i] = 1;
```

Fig. 5. Example of initialization using a loop; 'buf' is fully initialized but this code cannot be converted into a `memset`.

```
sprintf (t3, "%s%s;", t1, t2);
strcpy (t1, t3);
```

Fig. 6. `t3` is a 'safe' string buffer (from gcc in SPEC CPU2006) which does not need initialization

following this execution path, then it is not an appropriate place for initialization; this typically occurs if the first stores to a variable are inside a loop. To resolve this, we instead use an initialization point which also dominates all the predecessor basic blocks of such dominating points.

2) Detecting initialization: We propose detection of typical code which initialize arrays (or portions of them), which allows other compiler optimizations to remove or shorten previous stores which are revealed to be overwritten.

Typical compiler optimizations perform this only for individual store instructions, or intrinsics such as `memset`. While modern compiler transforms attempt to convert some loops to `memset` calls [18], this is only possible if a single-byte (or in some cases, two-byte) pattern is used. This is insufficient for many common cases, such as initializing an array of (four-byte) integer values to the value '1', as shown in Figure 5. Our design detects such code, treating these loops as if they were single stores which cover the entire range they initialize.

3) String buffers: Buffers which are used to store C-style null-terminated strings are often only used in a 'safe' manner, where the data in memory beyond the null-terminator is never used. We propose a low-cost check which catches simple cases, such as that in Figure 6; buffers which are only passed to known C library string functions (such as `strcpy` and `strlen`) are 'safe'. When initializing with zero, only the first byte of such buffers must be initialized.

Compilers already know about and detect such 'built-in' string functions, so we can take advantage of their existing infrastructure to detect these functions; there is no need to add annotations. Where the optimizer can prove that the string is always initialized, the initialization can later be removed entirely; however, this often only becomes clear after further optimizations have been applied.

D. Dead Store Elimination

To minimize the performance cost of initialization, SafeInit also includes a variety of improved optimization passes. These are more complex than our other optimizations, and may not always be necessary to obtain low overhead. However, they resolve real situations which we found to introduce unnecessary overhead when using our hardening passes.

In particular, we need so-called 'dead store elimination' (DSE) optimizations, a standard class of compiler optimizations [4] which remove stores which are always overwritten by another store without ever being read. We propose DSE-style optimizations which are particularly appropriate for removing initializations; existing optimizations are often ill-suited to this task, since these situations occur less frequently in other code, and so are less of a priority for compiler development.

```
row *r = malloc(sizeof(row));
r->row_num = 0;
r->length = 0;
r->user_word = NULL;
```

Fig. 7. Example of removed zero stores (from 'espresso'); the memory returned from `malloc` is already cleared with zero values.

```
int buf[n];
memset(buf, 0, n);
memset(buf, 1, n);
```

Fig. 8. Example of an unnecessary non-constant-length store; the first `memset` can always be removed.

Only relatively simple DSE optimizations are available in current compilers, generally limited to only statically-sized stores within a single basic block. However, this is an active area of compiler development, and as we will later discuss, support for these forms of complex DSE is slowly being introduced in mainstream compilers. The optimizations we present here serve to demonstrate demonstrate the importance of this work and the potential performance improvements which are possible from more intensive optimization.

1) Heap clearing: Since all heap allocations are guaranteed to be initialized to zero, our compiler can remove any zero stores to freshly-allocated heap memory (treating all allocation functions as equivalent to `calloc`); an example is shown in Figure 7. Similarly, any loads from freshly-allocated memory are known to be zero (rather than being undefined behavior), and we can replace them with a constant value.

If memory is *fully* initialized to a non-zero value, then our optimizer can also rewrite the allocator call to an alternative allocation function which skips any potential (unnecessary) initialization; however, we want to be sure that *only* these instances are left uninitialized by our custom allocator.

2) Non-constant-length store removal: Dead Store Elimination is generally only performed when the stores are of a known constant length; we propose transforming stores with non-constant lengths, which is important to remove unnecessary initializations of both dynamic stack allocations and heap allocations. The simplest such situations are when an entire existing store is overwritten, such as the code in Figure 8. Our DSE also removes stores of a non-constant size, such as some of our initialization stores, when they are entirely overwritten by later stores.

3) Cross-block DSE: Our optimizer also performs Dead Store Elimination across multiple basic blocks. This is an active area of improvement for existing compilers, but is far more relevant when universal initialization is introduced, and is necessary to enable many of the optimizations below.

We need to remove both stores which are partially or completely overwritten by other stores (standard DSE) as well

```
int result_buf[BUF_SIZE];
return shared_func(data, result_buf);
```

Fig. 9. Example of write-only/'scratch' buffer; initialization is unnecessary if `shared_func` only writes to the pointer provided as the second argument.

as stores which are never used (for example, because the value goes out of scope before being used), and while we are primarily concerned about `memset`, we also remove normal stores. New opportunities for rewriting heap functions may also be revealed during this process, and our optimizer also applies these optimizations here.

4) Non-constant-length store shortening: To enable other optimizations, particularly involving string or other library functions, we also attempt to shorten stores by non-constant lengths. For example, if the first `x` bytes of an array are initialized, then we may be able to shorten an earlier initialization by `x` bytes, assuming that the value of `x` does not change between the stores. However, the compiler must either be able to prove that `x` is never larger than the size of the array, or add an additional (and potentially expensive) bounds check.

In practice, writing beyond the bounds of an array is undefined behavior, and existing compiler optimizations take advantage of this to make assumptions. If execution is always guaranteed to reach the second store after it has reached the first, the compiler can assume that the second store does not write beyond the size of the array, and thus that the first store may always be shortened.

The conservative approach proposed by our design fails to remove some stores which, in practice, are always safe. As we discuss in implementation, this turned out to be a serious limitation. The performance overhead of this optimization also means that it is only worthwhile on relatively large stores; we only apply it for stack allocations beyond a fixed size.

5) Write-only buffers: Sometimes, memory is allocated, but never written to. Removing unused local variables is known to be an important optimization [27], but typically interprocedural analysis has been unnecessary. A typical example is shown in Figure 9, where a function requires a memory buffer as an argument for storing results, but the caller never reads from this buffer, simply discarding the content. Our initialization further complicates this, by adding a new unnecessary write to initialize such buffers.

If the called function never reads from the buffer, then the entire buffer is unnecessary. One approach is to clone such functions and remove the arguments in these cases, enabling removal of the stores. However, this can dramatically increase code size; inlining or cloning can be very expensive, and our design aims to remain practical by avoiding the need for any additional interprocedural analysis. Instead, we annotate allocations and function (pointer) arguments which are only written to. If we can then show that portions of memory are only stored to, and not read, then all the stores can be removed.

VI. Implementation

We implemented a prototype of SafeInit by extending the clang compiler, and the LLVM compiler framework [35]. As discussed, the dead store optimizations which are vital for acceptable performance are an active area of development, so we based our work on a recent pre-release version of the code (LLVM revision 269558, from mid-May 2016).

A. Initialization pass

We implemented stack clearing as an LLVM pass, which we run before *any* other optimization pass is allowed to run – mostly importantly, before `mem2reg`, which will introduce `undef` values when an uninitialized stack variable is read.

Local variables in LLVM are defined by using the `alloca` instruction; our pass performs initialization by adding a call to the LLVM `memset` intrinsic after each of these instructions. This guarantees that the entire allocation is cleared, and are transformed into store instructions where appropriate.

B. Hardened allocator

We implemented our hardened allocator by modifying `tcmalloc`, a modern high-performance heap allocator [22].

The underlying pages for the allocator are obtained using `mmap` or `sbrk`, and are guaranteed to initially be be zero. We force the use of MADV_DONTNEED (or equivalent) when freeing any such allocations, and so large heap allocations are always zero, and need not be initialized. The performance overhead of tracking the initialization status of smaller allocations is excessive, so we simply clear all other heap allocations to zero before the allocator returns a pointer.

We also modified LLVM to treat reads from newly-allocated memory as returning zero, rather than `undef`, when SafeInit is enabled. As discussed, this is vital for avoiding the unpredictable consequences of undefined values.

C. Optimizer

We implemented our proposed sinking stores optimization for stack initialization by moving our inserted `memset` calls to the dominating point of all uses of the `alloca` (ignoring those which do not actually use the variable, such as casts or debug intrinsics). When compiling with optimizations enabled, clang will emit 'lifetime' markers which indicate the points at which local variables come into scope; we modified clang to emit appropriate lifetime markers in all circumstances, and insert the initialization after these points.

The `alloca` instructions corresponding to local variables are placed in the first basic block, which is necessary for the majority of LLVM optimizations to function correctly, and for stack coloring to be performed. However, dynamic allocation of stack space within a function may not be in the first block (such as when an `alloca` call is made from C/C++ code); in these circumstances, we have to also ensure that initialization is not performed before the allocation takes place.

We implemented initialization detection optimization by adding a new intrinsic function, 'initialized', which has the same store-killing side effects as `memset`, but is ignored by code generation. By extending components such as LLVM's loop idiom detection to generate this new intrinsic where replacing code with a `memset` is not possible, we allow other existing optimization passes to take advantage of this information without the need to modify them individually.

D. Dead Store Elimination

We implemented the other optimizations described above by extending existing LLVM code, keeping our changes minimal where possible. Our implementation of write-only buffers made use of the patch in D18714 (since merged), which added the basic framework for a `writeonly` attribute.

We also based our implementation of cross-block Dead Store Elimination on the (rejected) patch in D13363. Due to performance regressions, we disable this cross-block DSE for small stores (≤ 8 bytes); we also extended this code to support removing `memset`, and shortening such stores.

Our prototype currently only applies non-constant shortening to `memset` calls which overwrite an entire object, and requires that they be at least 64 bytes for the efficiency reasons discussed above. LLVM's limited support for range analysis severely limits the current optimization opportunities for such shortening, since in the majority of cases we are unable to prove accesses are safe without performing our own analysis.

Since our goal is to show techniques which are practical to implement without needing additional analysis, we limited ourselves to the typical analyses which are used by existing in-tree code in such circumstances. These include checking known bits, and making use of the 'scalar evolution' code for loop variables. In turn, these limitations remove opportunities for library call optimizations; we found that even our optimizations for string functions are of limited usefulness (outside of artificial micro-benchmarks) due to the effect of these safety checks.

E. Frame clearing

To put our evaluation into context, we also implemented an alternative compiler hardening pass which clears the portion of each frame reserved for local variables in every function prologue. The performance of this frame clearing provides an estimate of the lower bound for these naive approaches; we apply our normal stack hardening pass to protect non-static (dynamically-allocated) local variables.

This improves performance compared to simply clearing all frames, since we do not clear space reserved for other purposes such as spilled registers (although our optimized clearing code sometimes clears part of this space, for alignment reasons). This approach also fails to provide guarantees for overlapping or re-used variables within the function; any changes to resolve these (such as disabling stack coloring to avoid overlapping variables) resulted in significantly worse performance.

VII. Detection

Our hardened toolchain can also be combined with a modern high-performance multi-variant execution system such as [30] to provide a detection tool, inspired by DieHard [5]. We compile multiple versions of the same application, initializing memory to different values in each variant. This allows us to perform high-performance detection of the majority of uses of uninitialized values, including those which would typically be removed by compiler optimizations or only stored in registers, without the false positives resulting from 'harmless' memory reads which do not affect the output. Example usage can be seen in Figure 11.

```
int deny_access;
if (deny_access) {
  printf("Access denied.");
  return 0;
}
printf("Access granted.");
```

Fig. 10. (Simplified) example of an uninitialized read which is optimized away by existing compiler transforms; in this example, the code in the branch is typically removed entirely.

```
$ var-cc -O2 example.c
$ multivar ./example-v0 ./example-v1
! SYSCALL MISMATCH for variants v0 and v1
 ~ 0: write(1, "Access granted.", 15)
 ~ 1: write(1, "Access denied.", 14)
 == Backtrace ==
  ip = 7271a9620 __write+16
  ...
  ip = 727120de9 _IO_printf+153
  ip = 4007ce check_access+366
```

Fig. 11. Example of an uninitialized read being detected, using optimized builds of the code in Figure 10; since there is no uninitialized memory usage in the optimized binaries, tools such as valgrind fail to detect such cases.

Filling memory with a constant value is much faster than using random values, so we fill all uninitialized bytes of memory in each variant with the same constant. Some optimizations are no longer possible when using non-zero values; in particular, we need to clear all heap memory, since the zero pages returned from the kernel are no longer appropriate.

However, multi-variant systems do not necessarily require synchronization (they need not run variants in 'lockstep'); system calls need only be executed for one of the variants, the so-called 'leader'. Since our hardening has already mitigated potential security issues, there is no need to run the variants in lockstep. We initialize the values of the leader process with zero, allowing it to run ahead of the other variants, which reduces the overall runtime impact of this slower initialization.

VIII. Evaluation

Our benchmarks were run on a (4-core) Intel i7-3770 with 8GB of RAM, running (64-bit) Ubuntu 14.04.1. CPU frequency scaling was disabled, and hyperthreading enabled. Transparent Huge Pages were turned off, due to their extremely unpredictable effect on runtime and memory usage – this is a commonly-recommended vendor configuration, and although it has a negative effect on some benchmarks, it does not appear to meaningfully change our overhead figures.

Our baseline configuration is an unmodified version of clang/LLVM, using an unmodified version of `tcmalloc`. As well as comparing this to SafeInit, we also present results for the naive approach, which simply applies our initialization pass without any of our proposed optimizations, using a hardened allocator which simply zeroes all allocations. We do make use of a modified compiler which performs local variable initialization and ensures that safety is maintained; for

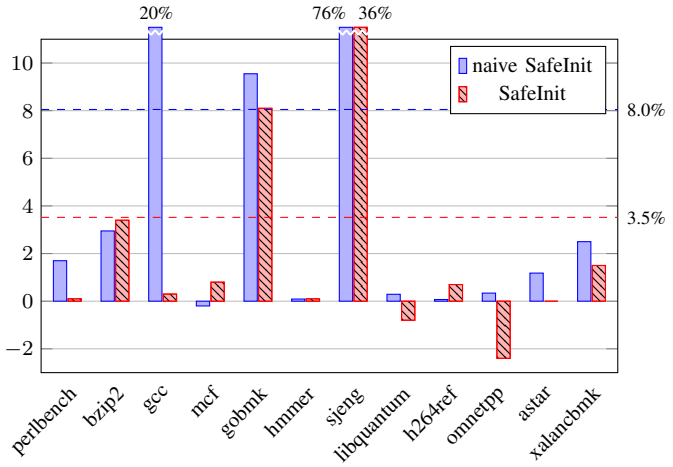

Fig. 12. SPEC CINT2006, runtime overhead (%) when hardening with SafeInit

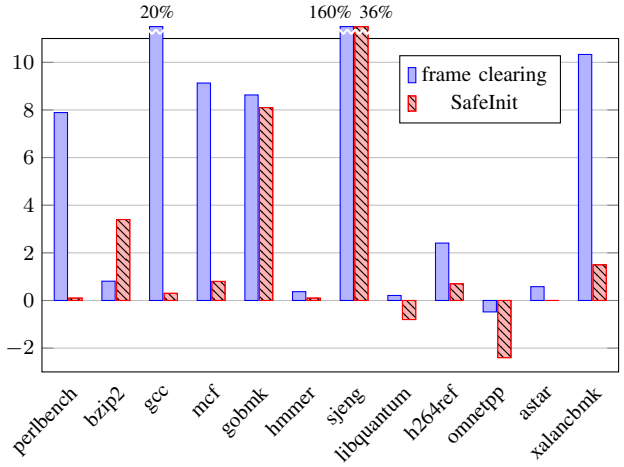

Fig. 13. SPEC CINT2006, runtime overhead (%) of frame initialization

example, zero is propagated from heap allocation sites, rather than `undef`.

A. SPEC CPU2006

We built all C/C++ benchmarks in SPEC CPU2006 using LTO and -O3, except the specrand test benchmark. We present overhead figures for the median of 3 runs, using the reference dataset. The runtime overhead of applying SafeInit to CINT2006 is shown in Figure 12.

SafeInit incurs a low performance cost for many benchmarks, even without our optimizer; these are generally CPU-bound and/or make their allocations only at startup. For example, `mcf` already uses `calloc` for allocating heap memory, and does not make significant use of the stack. However, other benchmarks experience significant runtime overhead; the (geometric) mean of the runtime overhead is 8%, when applied without our optimizer.

Applying our optimizer reduces the overhead for the remaining benchmarks significantly, as expected, resulting in average overhead for CINT2006 of 3.5% compared to our

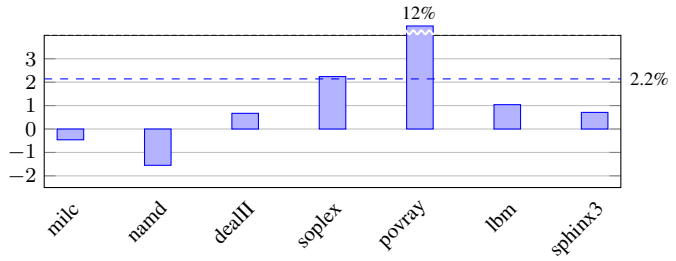

Fig. 14. SPEC CFP2006, runtime overhead (%) for SafeInit

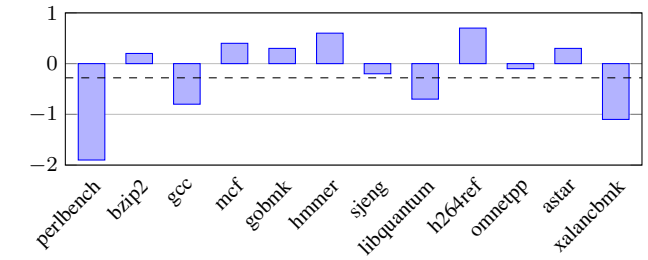

Fig. 15. SPEC CINT2006, runtime overhead (%) of SafeInit's optimizer without hardening applied

baseline compiler. Results for CFP2006 are similar, as shown in Figure 14, with an average overhead of 2.2%.

Table I provides details of the number of allocas (representing the number of local variables, plus occasional copies of arguments or dynamic allocations) for each benchmark. Many initializations are transformed or removed during optimization, but the table contains the number of initializations which are still represented as a `memset` in the final post-optimization bitcode, both with and without our customised optimizations. Note that these may still be converted into stores during code generation, and that although the optimizer often fails to remove initializations entirely, it can still obtain performance benefits due to transforming, moving or shortening them. The table also provides the (stripped) binary size; in many cases, the impact of the initialization makes no difference to the final binary size whatsoever, and in the worst case it is minimal.

When our optimizer is enabled without hardening, only a minimal performance improvement of around 0.3% is seen, as shown in Figure 15. The highest overhead is for `h264ref`; the impact of this overhead can also be seen in Figure 12 above. We do not believe the regressions seen here are fundamental, but in any case, our other overhead figures include the effect of any small regressions. If necessary in practice, any problematic individual optimizations could be adjusted or simply disabled when compiling without hardening.

The mean overhead when using our optimizer as the baseline is 3.8%; the primary contributors to this difference are small performance regressions of `perlbench` and `xalancbmk`, and a reduced performance improvement for `omnetpp`. The latter has a >1.5% performance improvement against both baselines in any case, thanks to the combination of our optimizations and improved cache behavior.

Figure 13 presents the performance overhead of our frame initialization pass, combined with our hardened allocator.

TABLE I. SPEC CINT2006 DETAILS. #INITS IS THE NUMBER OF LARGE INITIALIZATIONS LEFT AFTER EXISTING COMPILER OPTIMIZATIONS AND OUR OPTIMIZER HAVE RUN, RESPECTIVELY. SIZE IS THE (STRIPPED) BINARY SIZE.

Benchmark	#allocas	#inits (naive)	#inits (opt)	size (baseline)	size (naive)	size (optimizer)
astar	790	7	4	43736	43736 (0%)	43736 (0%)
bzip2	679	23	20	80488	84584 (5.1%)	84584 (5.1%)
gcc	31551	650	596	4108712	4133288 (0.6%)	4120992 (0.3%)
gobmk	17039	325	300	3554640	3566928 (0.3%)	3566928 (0.3%)
h264ref	4229	122	122	630664	638856 (1.3%)	638856 (1.3%)
hmmer	3333	19	18	189592	189592 (0%)	189592 (0%)
libquantum	567	3	2	31336	31336 (0%)	31336 (0%)
mcf	184	1	1	19040	19040 (0%)	19040 (0%)
omnetpp	7638	110	110	806712	810808 (0.5%)	814904 (1%)
perlbench	12327	175	167	1272584	1284872 (1%)	1280792 (1%)
sjeng	770	61	48	133976	133976 (0%)	133976 (0%)
xalancbmk	92396	1701	1302	3871528	3908392 (1%)	3892008 (0.5%)

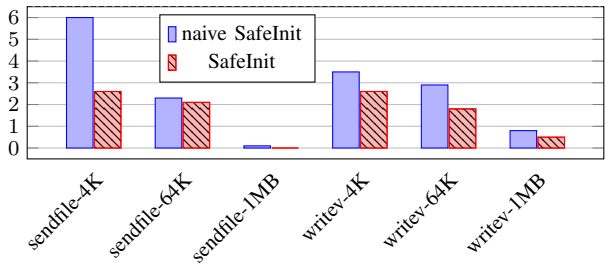

Fig. 16. requests/sec overhead (%) for hardening lighttpd

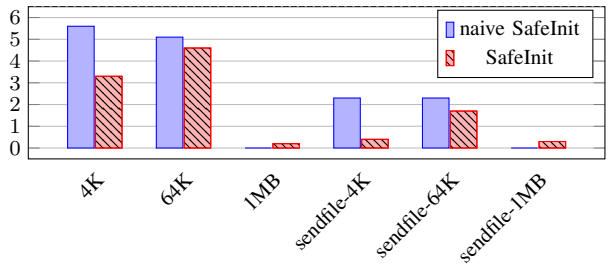

Fig. 17. requests/sec overhead (%) for hardening nginx

Given the average overhead of 13.5%, it is clear that such frame-based initialization without the benefit of compiler optimizations is too slow. Despite this simpler approach offering considerably less safety, only `bzip2` gains significant performance benefit from these reduced guarantees.

We also investigated another approach for weakening guarantees to improve performance, by increasing the lifetimes of variables inside loops so they would only be initialized once, before the loop. The impact of this on stack coloring and register allocation resulted in worse performance for almost all benchmarks (and average overhead for CINT2006 of >5%).

B. Servers

We evaluated the overhead of SafeInit for less computationally-intensive tasks by using two modern high-performance web servers, nginx (1.10.1) and lighttpd (1.4.41). We built the web servers using LTO and -O3. Since they are I/O bound when used on our 1gbps network interface, we benchmarked them using the loopback interface. This is an extreme scenario; in practice, the overhead of SafeInit is insignificant for these servers.

We used apachebench to repeatedly download 4Kb, 64Kb and 1MB files, for a period of 30 seconds. We enabled pipelining, used 8 concurrent workers, and used CPU affinity to reserve a CPU core for apachebench. We measured the overhead for the median number of requests per second, across 10 runs; we did not see significant amounts of variance.

lighttpd: We attempted to configure lighttpd to optimize throughput, allowing 100 requests per pipelined connection, and evaluated both the `sendfile` (default) and `writev` network backends. The results are shown in Figure 16.

Average overhead is minimal when sending the large (1MB) file. In the extreme case of the small 4Kb file, where we process almost 70,000 requests per second, overhead is still less than 3%; the majority of execution time here is spent parsing incoming requests and maintaining internal structures.

Much of lighttpd's overhead for these tiny requests is caused by small heap allocations for strings in the chunk queue; only the first byte of these is initialized by the caller, but our hardened allocator clears the entire allocation for safety. The remaining overhead for both situations is due to lighttpd's `writev` code, used by both backends for writing these allocations to the network, uses a fixed-size stack buffer. Our current optimizer fails to optimize away the unused portion of the buffer, but improved optimizations or minor changes to the code could reduce the overhead further. In fact, older versions of lighttpd used a larger buffer in this code, but recently a "sane limit" was imposed on the buffer size; such modifications demonstrate how general code improvements can also reduce the overhead imposed by SafeInit.

nginx: We tested nginx both with a default configuration (which is similar to the one we used for lighttpd) and with sendfile enabled (which significantly increases performance for serving the 64Kb and 1MB files). All logging was disabled; our overhead is slightly reduced when logs are enabled. The results are shown in Figure 17.

Overhead of full SafeInit, including our optimizer, is noticably higher with the 64Kb files; however, the overhead of SafeInit remains below 5% in all circumstances.

nginx makes use of a custom pool-based memory allocator, which makes it difficult for our optimizer to analyse code. However, we manually verified that memory is not (by

TABLE II. PHP 7.0.9 MICRO-BENCHMARK RESULTS (IN SECONDS)

	`bench.php`	`micro_bench.php`
baseline	1.029	3.983
new optimizer	1.007 (-2.1%)	3.879 (-2.6%)
naive SafeInit	1.004 (-2.5%)	3.994 (0.3%)
SafeInit	0.999 (-3%)	3.897 (-2.8%)

TABLE III. LMBENCH RESULTS. TIME IN MICROSECONDS, PLUS % OVERHEAD ABOVE BASELINE.

Sub-benchmark	Baseline	w/Optimizer	Stack SafeInit
syscall null	0.0402	0.0402 (0%)	0.0402 (0%)
syscall stat	0.2519	0.2369 (-5.9%)	0.2571 (2.1%)
syscall fstat	0.0739	0.0742 (0.4%)	0.0775 (4.9%)
syscall open	0.7049	0.6778 (-3.8%)	0.7119 (1%)
syscall read	0.0817	0.0819 (0.2%)	0.0819 (0.2%)
syscall write	0.0981	0.0979 (-0.2%)	0.0971 (-1%)
select tcp	4.5882	4.6714 (1.8%)	4.6497 (1.3%)
sig install	0.0964	0.0977 (1.4%)	0.1000 (3.7%)
sig catch	0.6534	0.6495 (-0.6%)	0.6648 (1.7%)
sig prot	0.2220	0.2210 (-0.4%)	0.2350 (5.9%)
proc fork	65.5904	66.6386 (1.6%)	67.7927 (3.4%)
proc exec	208.8846	209.8519 (0.5%)	212.3462 (1.7%)
pipe	3.3500	3.3834 (1%)	3.4145 (1.9%)
tcp	6.7489	6.7163 (-0.5%)	6.6835 (-1%)
bw pipe (MB/s)	4988.09	4974.89 (0.3%)	5182.4 (-3.9%)
bw tcp (MB/s)	8269.34	8245.39 (0.3%)	8350.71 (-1%)

default) re-used within the pool, to ensure that any potential uninitialized memory vulnerabilities would still be mitigated.

We also ran nginx using our detection tool (using two variants); overhead (above our hardened version) was generally similar to that reported by Koning et al. [30], with worst-case overhead of <75%.

PHP: We also evaluated a modern high-performance scripting language, PHP 7.0.9. We used the default compiler flags (-O2), since we encountered build system problems when attempting to use LTO. However, PHP makes extensive use of an internal memory allocator, which re-uses memory obtained from our hardened allocator; this reduces our safety guarantees for smaller allocations.

We ran both supplied PHP micro-benchmarks (from the `Zend` directory). The median of 21 runs (we saw little variation between runs) is shown in Table II; the combination of SafeInit and our new optimizer result in performance improvements of around 3% for both micro-benchmarks. We saw approximately 1% overhead (above the hardened version) when running these benchmarks under our detection system (using two variants).

C. Linux

We built the latest LLVMLinux [37] kernel tree[1] using our toolchain. We customized the build system to allow use of LTO, re-enabled built-in clang functions, and modified the gold linker to work around some LTO code generation issues we encountered with symbol ordering.

Since the Linux kernel (inherently) performs its own memory management, it does not get linked with our hardened userspace allocator; our automatic hardening only protects local variables. Protecting other sources of uninitialized data, such as the SLAB and buddy allocators, would require manual changes, and presumably add further overhead; such sanitization is already offered by kernel patches such as grsecurity.

Table III provides a selection of latency and bandwidth figures for typical system calls, using LMbench, a kernel microbenchmarking tool [43]. We ran each benchmark 10 times, with a short warming-up period and a high number of iterations (100) per run, and provide the median result. TCP connections were to localhost, and other parameters were those used by the default LMbench script. The overhead numbers for the hardened kernels include the (negligible) overhead of hardening LMbench itself.

We incur substantial overhead for the `stat` and `open` system calls; while this is largely mitigated by the improved performance provided by our optimizer, it is a cause for concern, and we intend to investigate it further, along with `fstat` and the (signal) protection fault, which is the only system call we saw with overhead >5%.

To evaluate the real-world performance of SafeInit applied to the kernel stack, we hardened *both* nginx and the kernel with SafeInit, and compared performance to a non-hardened nginx running under a non-hardened kernel. Using the sendfile configuration we discussed above, and again using the loopback interface to provide an extreme situation, we observed overhead of 2.9%, 3% and 4.5% for the 1M, 64kB and 4kB cases respectively.

We present the numbers above as a view of what is possible with only *automatic* mitigation, without application-specific knowledge. Our optimizer could be extended with knowledge of heap functions, inline assembly, and core kernel functions such as `copy_from_user`, which would provide both improved guarantees and more opportunities for optimization.

D. Residual Overhead

The average overhead of CINT2006 is distorted by the performance overhead of two outliers. The most significant is sjeng, a chess program. It stores game moves in large on-stack arrays in several recursive functions, and these arrays are then passed to many other functions, with the size stored in a global variable. This code is so convoluted that, even with manual inspection, we are unable to determine whether or not array elements may be used without being initialized. An appropriate approach might be to refactor or rewrite the code in question, removing such 'code smells', which would benefit both compiler analysis as well as our manual inspection.

This may be unrealistic in some cases, so we added compiler support for annotating variables and types with a 'no_zeroinit' attribute, and annotated sjeng's `move_s` type; this single annotation successfully reduced sjeng's runtime overhead to 6.5% (which would, in turn, reduce the mean overhead for CINT2006 to less than 2%), in combination with our full set of optimizations.

lighttpd's buffer preparation function, discussed earlier, could also benefit from such an annotation. However, since lighttpd does *not* clear the entire buffer, this would also require detailed manual inspection to ensure it was safe; we

[1] based on mainline revision f800c25b

TABLE IV. WARNING PASS OUTPUT, FOR CINT2006

Benchmark	#Warnings	Notes
bzip2	4	one is a 4MB buffer added by SPEC
gcc	1	
gobmk	8	mostly too complex to analyze
h264ref	7	
perlbench	1	unused at runtime
sjeng	19	17 of these are `move_s`
xalancbmk	16	temporary (wide) string buffers

TABLE V. VERIFIED UNINITIALIZED VALUE MITIGATIONS

CVE number	Software	Mitigated?	Description
2016-4243	PHP	✓	Use of uninitialized stack variables, including a pointer.
2016-5337	qemu	✓	Info disclosure to guest; missing null terminator for stack string buffer.
2016-4486	Linux	✓	Info disclosure to userspace; uninitialized padding in struct on stack.

do not believe the reduced safety in adding such annotations is justified, given the low overhead of our approach.

We also added a warning pass to our compiler, which can omit warnings (at link time) about large on-stack allocations (by default, >4kB) for which our optimizer failed to remove initialization. Figure IV summarizes the results for CINT2006 (excluding the benchmarks which output no warnings). Many of these are not on critical paths for performance, and some are completely unused in practice, such as a 8kB buffer in perlbench described in the source code as "The big, slow and stupid way". These warnings could be combined with profiling to determine which code needs to be refactored or annotated.

E. Security

To verify that SafeInit works as expected, we not only considered a variety of real-world vulnerabilities, such as those in Table V, but also created a suite of individual test cases. We inspected the bitcode and machine code generated for the relevant code manually, and also ran our test suite using the detection system we described above. We also used valgrind to verify our hardening; for example, we confirmed that all uninitialized value warnings from valgrind disappear when OpenSSL 0.8.9a is hardened with SafeInit.

As with all compiler optimizations, our improvements may expose latent bugs in other compiler code or in the source being compiled, or even contain bugs themselves. We verified that the benchmarks we ran produced correct results. We also extensively tested our hardened kernel, and where available ran test suites for the compilers and software we hardened (such as PHP). However, the potential for such issues remains an inherent risk when using any modern compiler, as shown by Yang et al. [64]. Formal verification of compilers (e.g., CompCert [36]) or individual optimization passes (such as that by Zhao et al. [67] and Deng et al. [16]).

However, in total, our SafeInit prototype adds or modifies less than 2000 lines of code in LLVM, including some debugging code and around 400 lines of code based on third-party patches. Although our modifications are complex, this is a relatively small amount of code and each component should be individually reviewable; for comparison, our (separate) frame clearing pass alone is more than 350 lines of code.

```
$ multivar php poc.php
Starting php-zero (20439)
Starting php-poison (20440)
20440 term sig: Segmentation fault (11)
```

Fig. 18. Detection output when checking PHP CVE-2016-4243

Our hardening does not prevent programs from reusing memory internally. For example, a stack buffer may be reused for different purposes within the same function, or a custom internal heap allocator may reuse memory without clearing it, such as we saw with PHP. Although it would potentially be possible to catch some of these cases using heuristics, or by attaching annotations of some kind, we do not believe it is realistic nor reasonable for a compiler to support this.

Clearing variables to zero ensures that any uninitialized pointers are null. An attempt to dereference such a pointer will result in a fault; in such situations, our mitigation has reduced a more serious problem to a denial-of-service vulnerability.

In many cases, code will specifically check for null pointers or other variables, and so clearing variables mitigates issues entirely; when running our detection system, we noticed that many uninitialized pointer dereferences were only triggered in the variant initialized with a non-zero value. For example, Figure 18 shows the output of our detection system executing a proof-of-concept exploit for PHP CVE-2016-4243. Only the variant initialized with a non-zero value attempts to dereference the value (which results in a fault, caught by our system).

Initializing all variables with zero also has the potential to activate vulnerabilities which would otherwise have remained dormant. A contrived example could be a 'insecure' variable, which is used to force a check of some kind, but is used uninitialized. This may not be a problem in practice under some environments, where the underlying memory happens to always contain a non-zero value. However, this may change at any time, and since compilers are allowed to transform such undefined behavior, it is always possible that such code may be optimized away.

As stated in our threat model, we only consider C/C++ code; assembly routines fall outside the scope of this work, although typical inline assembly will declare local variables in C/C++ code, which would then be initialized by our prototype. Since we have implemented our SafeInit prototype as an LLVM pass, other compiler frontends making use of LLVM could also easily benefit from our work; we look forward to experimenting with NVIDIA's upcoming Fortran front-end.

IX. LIMITATIONS

Libraries: For complete protection against all uninitialized value vulnerabilities, all used libraries must also be instrumented. The standard C library used on Linux, glibc, does not build under clang, so our prototype implementation is unable to instrument it; this is a limitation of our specific implementation, not our design. Stepanov et al. [58] state that they implemented interception of 'close to 300' libc functions in MemorySanitizer; while such knowledge of library functions is not required by SafeInit, having access to the bitcode for libraries would also allow further performance improvements.

Since both the toolchain and C library are usually provided together, we feel it would be reasonable to make small improvements to the C library to mitigate any performance issues for specific functions. However, in any case, we observed no meaningful overhead (<0.1%) when building benchmarks and applications against an unmodified alternative C library (musl).

Performance: Our modified optimizer can cause (small) performance regressions in some code, caused by unintended consequences on other optimizations and code generation. For example, removing stores makes functions smaller, and so more likely to fall under the inlining threshold; we can improve performance across all of our benchmarks by modifying the threshold. To be as fair as possible, we presented our results without any such changes. The optimizations proposed in our design and implemented in our prototype are deliberately minimal, without additional analysis, to show they are practical to implement in current compilers; this limits many of the possible transformations. Despite this, we expect the overhead of SafeInit to decrease significantly over time, as the related compiler optimizations continue to improve.

There will inevitably be cases where performance is unacceptable in real-world code, as we saw with sjeng. Where annotations are an unacceptable solution, making changes to the code may be necessary. However, such refactoring can also improve the code in other ways, whether just making it more readable and easier to understand, or as we saw with lighttpd, also by resolving potential memory or performance issues.

Relevant recent developments in LLVM include improvements to loop analysis and optimization [47] as well as transforming entire structure definitions to improve performance [26]. During the development of our project, improvements to LLVM's store optimizations have also continued; for example, one recent patch improved removal of stores which are overwritten by multiple later store instructions, allowing removal of unnecessary initializations when individual members of a structure are initialized. We look forward to seeing how future optimizations further decrease the overhead of our work.

X. Related Work

Detection: Dynamic analysis tools for detecting uses of uninitialized data, such as valgrind's memcheck [54], track the initialized state of each bit of memory and (optionally) the origin of any uninitialized data. The high overhead of this tracking makes it often prohibitive for use during development, and completely impractical to deploy. It is almost essential to use optimized binaries, where undefined behavior may have already introduced undetectable vulnerabilities, along with other issues which reduce the reliability of this approach, such as re-use of stack memory within functions.

More recent detection tools using a similar approach include Dr. Memory [7], which significantly reduces overhead by applying optimizations, and MemorySanitizer [58] (MSan), which reduces overhead even further by instrumenting binaries during compilation (using LLVM). The execution time overhead for MSan is reported as 2.5x (with optimized binaries), which is sufficient to make it usable as part of continuous integration for projects such as Chrome, and advancements such as chained origin tracking mean that reported errors require less manual effort to fix. Recent research [65] claims to have reduced MSan overhead even further.

Berger et al. proposed using multi-variant execution to detect uses of uninitialized heap allocations in DieHard [5]. By running multiple variants of the same program and filling newly-allocated heap memory with random values, and providing all variants with identical input, any deviation in output was likely to be due to use of uninitialized memory. To obtain reliability against memory errors, they proposed running several variants, and dropping any reporting inconsistent results.

Stack clearing: gcc's Fortran compiler provides an `-finit-local-zero` option, intended only for compatibility with legacy Fortran code. Several C/C++ compilers provide options for automatic initialization of function stack frames, intended only for debugging purposes. As discussed, such stack frame clearing adds a significant performance penalty, and provides fewer guarantees.

Chen et al. presented StackArmor [9], a binary hardening system which isolated function frames containing potentially unsafe buffers using guard pages and random reordering. This makes it more difficult for attackers to predict which data may be present in uninitialized portions of frames, providing probabilistic mitigation of uninitialized data vulnerabilities; they combined it with analysis to add zero-initialization to potentially uninitialized portions of non-isolated frames, but reported high average overhead of 28% on SPEC CPU2006.

Heap clearing: Heap allocation clearing is an option in some existing allocators, such as jemalloc [20], although generally intended only for debugging; for example, the jemalloc documentation warns that it "will impact performance negatively". Wang et al. [62] proposed zero-initializing and padding heap allocations at allocation time, by wrapping `malloc`, to protect against buffer overread vulnerabilities. Araujo and Hamlen [3] suggested just zeroing the first byte of all allocations, giving limited benefits (e.g., for C strings) but adding almost no overhead.

Chow et al. proposed Secure Deallocation [11], which modifies the system C library to zero heap allocations when freed, and modifies compiler code generation to clear stack frames in function epilogues; this provides less comprehensive protection and misses optimization opportunities. They claimed runtime overhead of <7% for heap clearing, but 10%-40% overhead for stack clearing, although their approach does protect against some vulnerabilities outside our threat model.

Heap isolation: Isolating all heap allocations mitigates some classes of memory vulnerabilities, such as overflows; however, this is at best a probabilistic defense, since limited available address space means memory is inevitably reused after a certain point. DieHard [5] allocates memory randomly across an oversized heap, and Archipelago [41] allocates memory across the entire address space. OpenBSD [46] implemented such a random allocator by default, including moving metadata out-of-bound, and DieHarder [48] built upon this to increase entropy at an additional performance cost of 20%, due to the cost of memory fragmentation.

Information disclosure defenses: Many defenses have been proposed for protecting sensitive data. TaintEraser [68] uses tainting to track sensitive user input and prevent it from

escaping to the filesystem or network. Harrison and Xu [24] proposed techniques for probabilistically protecting private cryptographic keys against memory disclosure attacks, and SWIPE [23] tracks sensitive data using static analysis and erases it at the end of its lifetime.

Defenses which depend on information hiding to protect pointers or other metadata are particularly vulnerable to information disclosure. Advances such as fine-grained ASLR [25] are rendered useless if uninitialized memory errors can be used to disclose pointers. Defenses such as Code-Pointer Integrity [32], Readactor [15] and ASLR-Guard [39] aim to protect code pointers against more sophisticated disclosure attacks such as those proposed by Evans et al. [19] and Schuster et al. [51].

Linux kernel

Uninitialized data vulnerabilities in the Linux kernel have had increased attention in recent years; as well as obvious issues of exposing confidential information, knowledge of kernel addresses has become important for attackers wishing to bypass defenses such as stack canaries (using gcc's StackGuard [14]) and ASLR (kASLR [13]). In 2011, Chen et al. [8] performed an extensive analysis of kernel vulnerabilities and reported that the most common category were uninitialized data errors, almost all of which led to information disclosure. More recently, Peiró et al. [50] provided more in-depth discussion of such kernel info disclosure vulnerabilities, and presented a technique for identifying stack information disclosures using static analysis. Linux also includes kmemcheck, a dynamic analysis tool for detecting uses of uninitialized heap memory.

grsecurity/PaX: The PaX project [49], as part of the hardened grsecurity Linux patches, provides two different mitigations for potentially uninitialized kernel stack data, using gcc plugins. One annotates structures which may be disclosed to userspace, and initializes any such structures on the stack to prevent accidental information disclosure. The other takes a more aggressive approach, clearing the kernel stack before/after system calls. A gcc plugin tracks the maximum stack depth used for each call, providing efficient protection against stack re-use between different system calls, although still theoretically allowing an attacker to exploit such issues within a single call. Both grsecurity and recent mainline kernels can also be configured to initialize and/or clear heap allocations.

UniSan: Concurrently to our work, Lu et al. developed UniSan[38], a compiler-based approach for mitigating information disclosure vulnerabilities caused by uninitialized values in the Linux kernel. They propose using static data-flow analysis to trace potential execution paths (after optimizations have been applied), and initializing any variables which cannot be proven to be initialized before potentially being disclosed; they implemented a prototype using LLVM, and manually inspected their analysis results to find and disclose various new uninitialized value disclosure vulnerabilities (some of which we used to verify the correctness of our own work).

Our approach mitigates a wider range of potential uninitialized value vulnerabilities on the stack (such as dereferencing uninitialized pointers [40] or even control-flow-based side-channel attacks [52]), and SafeInit obtains good performance without additional data-flow analysis. However, UniSan's interprocedural analysis and specific knowledge of kernel functions may result in significantly better overhead in some cases, particularly for the heap. We believe similar results could be obtained by adding knowledge of Linux heap functions and a Linux-specific optimization pass to SafeInit; combining both techniques may also be a promising approach.

XI. Conclusion

Uninitialized data vulnerabilities continue to pose a security problem in modern C/C++ software, and ensuring safety against the use of uninitialized values is not as easy as it might seem. Threats ranging from simple information disclosures to serious issues such as arbitrary memory writes, static analysis limitations, and compiler optimizations taking advantage of undefined behavior, combine to make this a difficult problem.

We presented a toolchain-based hardening technique, SafeInit, which mitigates uses of uninitialized values in C/C++ programs by ensuring that *all* local variables and stack allocations are initialized before use. By making use of appropriate optimizations, we showed that runtime overhead for many applications can be reduced to a level which makes it practical to apply as a standard hardening protection, and that this can be done practically in a modern compiler.

To foster further research in this area, we are open sourcing our SafeInit prototype, which is available at `https://github.com/vusec/safeinit`. We hope to work towards making SafeInit available as a standard compiler feature, and improving the optimizations it depends upon.

Acknowledgements

We would like to thank Kees Cook, Kangjie Lu and the anonymous reviewers for their comments. This work was supported by the European Commission through project H2020 ICT-32-2014 SHARCS under Grant Agreement No. 644571 and by the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI Dowsing.

References

[1] "CVE-2012-1889: Vulnerability in Microsoft XML core services could allow remote code execution," 2012.

[2] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security*, 2009.

[3] F. Araujo and K. Hamlen, "Compiler-instrumented, dynamic secret-redaction of legacy processes for attacker deception," in *USENIX Security*, 2015.

[4] M. Auslander and M. Hopkins, "An overview of the PL. 8 compiler," in *SIGPLAN Symposium on Compiler Construction*, 1982.

[5] E. D. Berger and B. G. Zorn, "DieHard: probabilistic memory safety for unsafe languages," in *PLDI*, 2006.

[6] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[7] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *CGO*, 2011.

[8] 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 *APSys*, 2011.

[9] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries." in *NDSS*, 2015.

[10] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *USENIX Security*, 2004.

[11] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation." in *USENIX Security*, 2005.

[12] K. Cook, "Kernel exploitation via uninitialized stack," *DEFCON 19*, 2011.

[13] ——, "Kernel address space layout randomization," 2013, Linux Security Summit.

[14] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security*, 1998.

[15] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *S&P*, 2015.

[16] C. Deng and K. S. Namjoshi, "Securing a compiler transformation," in *Static Analysis*, 2016.

[17] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *PLDI*, 2003.

[18] D. Edelsohn, W. Gellerich, M. Hagog, D. Naishlos, M. Namolaru, E. Pasch, H. Penner, U. Weigand, and A. Zaks, "Contributions to the GNU compiler collection," *IBM Systems Journal*, 2005.

[19] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point (er): On the effectiveness of code pointer integrity," in *S&P*, 2015.

[20] J. Evans, "A scalable concurrent malloc(3) implementation for FreeBSD," in *BSDCan*, 2006.

[21] H. Flake, "Attacks on uninitialized local variables," *Black Hat Europe*, 2006.

[22] S. Ghemawat and P. Menage, "TCMalloc : Thread-caching malloc," 2007.

[23] K. Gondi, P. Bisht, P. Venkatachari, A. P. Sistla, and V. N. Venkatakrishnan, "SWIPE: Eager erasure of sensitive data in large scale systems software," in *CODASPY*, 2012.

[24] K. Harrison and S. Xu, "Protecting cryptographic keys from memory disclosure attacks," in *DSN*, 2007.

[25] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *S&P*, 2012.

[26] G. Hoflehner, "LLVM performance improvements and headroom," in *LLVM Developers' Meeting*, 2015.

[27] J. Hubicka, "Interprocedural optimization framework in GCC," in *GCC Developers Summit*, 2007.

[28] T. Johnson and D. L. Xinliang, "ThinLTO: A fine-grained demand-driven infrastructure," in *EuroLLVM*, 2015.

[29] M. Jurczyk, "Enabling QR codes in Internet Explorer, or a story of a cross-platform memory disclosure," 2015.

[30] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *DSN*, 2016.

[31] B. C. Kuszmaul, "SuperMalloc: a super fast multithreaded malloc for 64-bit machines," in *ISMM*, 2015.

[32] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," in *OSDI*, 2014.

[33] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, 1992.

[34] C. Lattner, "What every C programmer should know about undefined behavior," 2011, LLVM project blog.

[35] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO*, 2004.

[36] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, no. 7, pp. 107–115, 2009.

[37] Linux Foundation, "LLVMLinux project."

[38] K. Lu, C. Song, T. Kim, and W. Lee, "UniSan: Proactive kernel memory initialization to eliminate data leakages," in *CCS*, 2016.

[39] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping address space leakage for code reuse attacks," in *CCS*, 2015.

[40] K. Lu, M.-T. Walter, D. Pfaff, N. Stefan, W. Lee, and M. Backes, "Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying," in *NDSS*, 2017.

[41] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn, "Archipelago: trading address space for reliability and security," in *ASPLOS*, 2008.

[42] V. Makarov, "The integrated register allocator for GCC," in *GCC Developers Summit*, 2007.

[43] L. W. McVoy and C. Staelin, "LMbench: Portable tools for performance analysis," in *USENIX*, 1996.

[44] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, "Into the depths of c: Elaborating the de facto standards," in *PLDI*, 2016.

[45] Microsoft, "MS08-014 : The case of the uninitialized stack variable vulnerability," 1998.

[46] O. Moerbeek, "A new malloc (3) for OpenBSD," in *EuroBSDCon*, 2009.

[47] A. Nemet and M. Zolotukhin, "Advances in loop analysis frameworks and optimizations," in *LLVM Developers' Meeting*, 2015.

[48] G. Novark and E. D. Berger, "DieHarder: securing the heap," in *CCS*, 2010.

[49] PaX Team, "PaX - gcc plugins galore," 2013, H2HC.

[50] S. Peiró, M. Munoz, and A. Crespo, "An analysis on the impact and detection of kernel stack infoleaks," *Logic Journal of IGPL*, 2016.

[51] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *S&P*, 2015.

[52] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *CCS*, 2014.

[53] F. J. Serna, "The info leak era on software exploitation," *Black Hat USA*, 2012.

[54] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *USENIX*, 2005.

[55] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *CCS*, 2004.

[56] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *S&P*, 2013.

[57] B. Spengler, "Detection, prevention, and containment: A study of grsecurity," 2002, libres Software Meeting.

[58] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in c++," in *CGO*, 2015.

[59] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *ICSE*, 2016.

[60] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *S&P*, 2013.

[61] R. van Eeden, "Unexpected code execution in smbd," 2015.

[62] J. Wang, M. Zhao, Q. Zeng, D. Wu, and P. Liu, "Risk assessment of buffer "heartbleed" over-read vulnerabilities," in *DSN*, 2015.

[63] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Undefined behavior: what happened to my code?" in *APSys*, 2012.

[64] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *PLDI*, 2011.

[65] D. Ye, Y. Sui, and J. Xue, "Accelerating dynamic detection of uses of undefined values with static value-flow analysis," in *CGO*, 2014.

[66] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage, "When private keys are public: results from the 2008 Debian OpenSSL vulnerability," in *IMC*, 2009.

[67] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formal verification of SSA-based optimizations for LLVM," in *PLDI*, 2013.

[68] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser: Protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Operating Systems Review*, 2011.