An Analysis of x86-64 Inline Assembly in C Programs

Manuel Rigger
Johannes Kepler University Linz
Austria
manuel.rigger@jku.at

Stefan Marr
University of Kent
United Kingdom
s.marr@kent.ac.uk

Stephen Kell
University of Cambridge
United Kingdom
stephen.kell@cl.cam.ac.uk

David Leopoldseder
Johannes Kepler University Linz
Austria
david.leopoldseder@jku.at

Hanspeter Mössenböck
Johannes Kepler University Linz
Austria
hanspeter.moessenboeck@jku.at

Abstract

C codebases frequently embed nonportable and unstandardized elements such as *inline assembly code*. Such elements are not well understood, which poses a problem to tool developers who aspire to support C code. This paper investigates the use of x86-64 inline assembly in 1264 C projects from GitHub and combines qualitative and quantitative analyses to answer questions that tool authors may have. We found that 28.1% of the most popular projects contain inline assembly code, although the majority contain only a few fragments with just one or two instructions. The most popular instructions constitute a small subset concerned largely with multicore semantics, performance optimization, and hardware control. Our findings are intended to help developers of C-focused tools, those testing compilers, and language designers seeking to reduce the reliance on inline assembly. They may also aid the design of tools focused on inline assembly itself.

CCS Concepts • **General and reference** → **Empirical studies**; • **Software and its engineering** → **Assembly languages**; **Language features**; • **Computer systems organization** → *Complex instruction set computing*;

Keywords Inline Assembly, C, Empirical Survey, GitHub

ACM Reference Format:
Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. 2018. An Analysis of x86-64 Inline Assembly in C Programs. In *VEE '18: 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, March 25, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3186411.3186418

VEE '18, March 25, 2018, Williamsburg, VA, USA

1 Introduction

Inline assembly refers to assembly instructions embedded in C code in a way that allows direct interaction; for example, they can directly access C variables. Such code is inherently platform-dependent; it uses instructions from the target machine's *Instruction Set Architecture* (ISA). For example, the following C function uses the `rdtsc` instruction to read a timer on x86-64. Its two output operands `tickh` and `tickl` store the higher and lower parts of the result. The platform-specific constraints `=a` and `=d` request particular registers.

```
uint64_t rdtsc() {
  uint32_t tickl, tickh;
  asm volatile ("rdtsc":"=a"(tickl),"=d"(tickh));
  return ((uint64_t)tickh << 32)|tickl;
} /* see §2.5 for detailed syntax */
```

This kind of platform dependency adds to the complexity of C programs. A single complex ISA, such as x86, can contain about a thousand instructions [25]. Furthermore, inline assembly fragments may contain not only instructions, but also assembler directives (such as `.hidden`, controlling symbol visibility) that are specific to the host system's assembler.

It is not surprising that many tools that process C code or associated intermediate languages (such as LLVM IR [38] and CIL [45]) partially or entirely lack support for inline assembly. For example, many bug-finding tools (e.g., the Clang Static Analyzer [70], splint [18, 19, 63], Frama-C [69], uno [26], and the LLVM sanitizers [55, 58]), tools for source translation (e.g., c2go [44]), semantic models for C [36, 43], and alternative execution environments such as Sulong [51–53] and KLEE [12] still lack support for inline assembly, provide only partial support, or overapproximate it (e.g., by analyzing only the side effects specified as part of the fragment), which can lead to imprecise analyses or missed optimization opportunities. How to provide better support depends on the tool, for example, in Sulong, adding support for assembly instructions requires emulating their behavior in Java, while support in a formal model would require specifying the instructions in a language such as Coq.

Literature on processing C code seldom mentions inline assembly except for stating that it is rare [31]. Tool writers would benefit from a thorough characterization of the occurrence of inline assembly in practice, as it would enable them to make well-informed decisions on what support to add. Hence, we analyzed 1264 C projects that we collected from GitHub. We manually analyzed the inline assembly fragments in them (i.e., inline assembly instructions that are part of a single `asm` statement). From these fragments, we created a database with fragments specific to the x86-64 architecture to quantitatively analyze their usage.

We found that:

- Out of the most popular projects, 28.1% contain inline assembly fragments.
- Most inline assembly fragments consist of a single instruction, and most projects contain only a few inline assembly fragments.
- Since many projects use the same subset of inline assembly fragments, tool writers could support as much as 64.5% of these projects by implementing just 5% of x86-64 instructions.
- Inline assembly is used mostly for specific purposes: to ensure semantics on multiple cores, to optimize performance, to access functionality that is unavailable in C, and to implement arithmetic operations.

Our findings suggest that tool writers might want to consider adding support for inline assembly in their C tools, as it is used surprisingly often. We also found that inline assembly is not specific to a small set of domains, but appears in applications in which one might not expect it (e.g., text processing). Since most applications use the same subset of inline assembly instructions, a large proportion of projects could be supported with just a moderate implementation effort. Another finding, however, is that instructions are not all that matters. Rather, assembly instructions are only one of many non-C notations used in C codebases, all of which generally suffer from the same lack of tool support. For example, some uses of `asm` contain no instructions, consisting only of assembler directives and constraints. Others are interchangeable with non-portable compiler intrinsics or pragmas. Yet others gain meaning in conjunction with linker commandline options or scripts. This paper is therefore a first step towards characterizing this larger "soup" of notations that tools must support in order to fully comprehend C codebases.

2 Methodology

To guide tool developers in supporting inline assembly, we posed six research questions. We detail how we scoped the survey, selected and obtained C applications, and finally analyzed their inline assembly fragments.

2.1 Research Questions

To characterize the usage of inline assembly in C projects, we investigated the following research questions (RQs):

RQ1: How common is inline assembly in C programs? Knowing how commonly inline assembly is used indicates to C tool writers whether it needs to be supported.

RQ2: How long is the average inline assembly fragment? Characterizing the length of the average inline assembly fragment gives further implementation guidance. If inline assembly fragments typically contain only a single instruction, simple pattern-matching approaches might be sufficient to support them. If inline assembly fragments are large, numerous, or "hidden" behind macro meta-programming [57], it might be more difficult to add support for them.

RQ3: In which domains is inline assembly used? Answering this question helps if a tool targets only specific domains. It seemed likely that the usage of inline assembly differs across domains. We expected inline assembly in cryptographic libraries because instruction set extensions such as AES-NI explicitly serve cryptographic code [6]. This was supported by a preliminary literature search, as inline assembly is, for example, often mentioned in the context of cryptographic libraries [23, 37, 40, 61]. We also expected it to implement related security techniques, preventing timing-side channels [7] and compiler interference [66, 67]. It was less clear what other domains make frequent use of inline assembly.

RQ4: What is inline assembly used for? Knowing the typical use cases of inline assembly helps tool writers to assign meaningful semantics to inline assembly instructions. It also helps to determine whether alternative implementations in C could be considered. We hypothesized that inline assembly is used—aside from cryptographic use cases—mainly to improve performance and to access functionality that is not exposed by the C language.

RQ5: Do projects use the same subset of inline assembly? Answering this question determines how much inline assembly support needs to be implemented to cope with the majority of C projects. Currently, C tool writers have to assume that the whole ISA needs to be supported. However, one of our assumptions was that most projects—if they use inline assembly—rely on a common subset of instructions. By adding support for this subset, C tool writers could cope with most of the projects that use inline assembly.

2.2 Scope of the Study

Our focus was to quantitatively and qualitatively analyze inline assembly code. For our quantitative analysis, we built a database (using `SQlite3`) of inline assembly occurrences in code written for x86-64, as it is one of the most widely used architectures. The database contains information about each project, inline assembly fragment, and assembly instruction analyzed. We used this database to perform aggregate queries, for example, to determine the most common instructions. The database and aggregation scripts are available at *https://github.com/jku-ssw/inline-assembly* to facilitate further research. Additionally, we qualitatively analyzed all instructions to summarize them in a meaningful way.

2.3 Obtaining the Projects

In our survey, we selected C applications from GitHub, a project hosting website. To gather a diverse corpus of projects, we used two strategies:

We selected all projects with at least 850 GitHub stars—an arbitrary cut-off that gave us a manageable yet sufficiently large sample—which resulted in 327 projects being selected. Stars indicate the popularity of a project and are given by GitHub users [10]. We assumed that the most popular projects reflect those applications that are most likely processed by C tools.

We selected another 937 projects by searching for certain keywords[1] and by taking all matching projects that had at least 10 stars. The goal was to select projects of a certain domain with different degrees of popularity to account for the long tail of the distribution. In order to avoid personal forks, experiments, duplicate projects and the like [32, 41], we did not consider projects that had fewer than 10 stars.

2.4 Filtering the Projects

Our primary goal was to analyze C application-level code which we consider to be of general interest. Consequently, we ignored projects if they were operating systems, device drivers, firmware, and other code that is typically considered part of an operating system. Such code directly interacts with hardware and thus comes with its own special set of issues and usage patterns of inline assembly. Further, to keep the scope manageable, we focused on code for x86-64 Linux systems. Therefore, we excluded projects that worked only for other architectures or other operating systems. Further, we did not consider uncommon x86 extensions such as VIA's Padlock extensions [65].

We restricted our analysis to C code, excluding C++ code. Projects that mixed C/C++ code were also excluded if the C++ LOC were greater in number than the C LOC. We also excluded C/C++ header files (ending with `.h`) when they contained C++ code. A number of projects used C code to implement native extensions for PHP, Ruby, Lua, and other languages; we included such code in our analysis. In a few cases, inline assembly was part of the build process; for example, some `configure` scripts checked the availability of CPU features by using `cpuid`. We discarded these cases because inline assembly was not part of the application; however, we checked whether build scripts generated source files with inline assembly, which we then incorporated in our analysis.

[1] Our keywords were: crc, argon, checksum, md5, base64, dna, web server, compression, math, fft, string, aes, simulation, editor, single header library, parser, debugger, ascii, xml, markdown, smtp, sqlite, mp3, sort, json, bitcoin, udp, random, prng, metrics, misc, tree, parser generator, hash, font, gc, i18, and javascript.

34 projects used inline assembly in fairly large program fragments, notably featuring SIMD instructions and using preprocessor-based metaprogramming. Although written using inline assembly constructs, these fragments have more in common with separate (macro) assembly source files. In particular, supporting these would require a close-to-complete implementation of an ISA. We excluded these fragments from our quantitative analysis.

We performed our analysis on unpreprocessed source code to include all inline-assembly fragments independent of compile-time-configuration factors [62]. This is significant because inclusion of inline assembly is often only conditional, achieved by `#ifdefs` that not only check for various platforms and operating systems, but also for configuration flags, various compilers, compiler versions, and availability of GNU C intrinsics [17]; examining only preprocessed code would have left out many fragments.

2.5 Inline Assembly Constructs

Since inline assembly is not part of the C language standard, compilers differ in the syntax and features provided. In this study, we assume use of the GNU C inline assembly syntax, which is the de-facto standard on Unix platforms, recognizes the `asm` or `__asm__` keywords to specify an inline assembly fragment, and has both "basic" and "extended" flavors. Using basic `asm`, a programmer can specify only the assembler fragment or directive. Use cases for basic assembly are limited; however, in contrast to extended `asm`, basic inline assembly can be used outside of functions. For example,

```
asm(".symver memcpy,memcpy@GLIBC_2.2.5")
```

uses basic inline assembly for a symbol versioning directive (see Section 5).

The more commonly used form is extended `asm`, which also allows specifying output and input operands as well as side effects (e.g., memory modifications). It is specified using

```
asm ( AssemblerTemplate : OutputOperands
        [ : InputOperands [ : Clobbers ] ]).
```

Adding the `volatile` keyword restricts the compiler in its optimization; for example, it prevents reachable fragments from being optimized (e.g., by register reallocation).

2.6 Analyzing the Instructions

Our analysis focused on inline assembly fragments found with `grep` in the source code. We searched for strings containing "asm", which made it unlikely that we missed inline assembly instructions. For the quantitative analysis, we judged whether an inline assembly fragment was used for an x86-64 Linux machine. If so, we manually extracted the fragment and preprocessed it (see the criteria below) using a script created for this purpose.

We assumed that tools would support all addressing modes (e.g., register addressing, immediate addressing, and direct memory addressing) for a certain instruction. Consequently, we did not gather statistics for different addressing modes. Inline assembly can contain assembler directives that instruct the assembler to perform certain actions, for example, to allocate a global variable. We ignored such assembler directives in our quantitative analysis, but discuss them qualitatively. An exception is the `.byte` directive, which is sometimes used to specify instructions using their byte representation (and similar cases, see Section 5), for which we assumed their mnemonic (i.e., their textual) representation.

By default, GCC assumes use of the AT&T dialect [8, 9.15.3.1, 9.15.4.2]; however, some projects enabled the Intel syntax instead. Using the AT&T syntax, a size suffix is typically appended to denote the bitwidth of an instruction. An `add` instruction can, for example, operate on a byte (8 bit), long (32 bit), or quad (64 bit) using `addb`, `addl`, and `addq`, respectively. Using Intel syntax, the size suffix is typically omitted. For consistency, we stripped size suffixes and recorded only the instruction itself (e.g., `add`). We also applied other criteria to group instructions.[2]

3 Quantitative Results

Based on our quantitative analysis, we can answer the first three research questions on the use of inline assembly in C projects, the length of fragments used, and the domains in which they occur.

Projects using inline assembly. Our corpus contained 1264 projects, of which 197 projects (15.6%) contained inline assembly for x86-64. The distribution differed between the popular projects and those selected by keywords. Among the most popular 327 projects, 28.1% contained inline assembly, while of the 937 other projects only 11.2% used inline assembly. One possible explanation for this difference is that the popular projects were larger (69 KLOC on average) than the projects selected by keywords (13 KLOC on average).

Density of inline assembly fragments. The percentage of projects with inline assembly is high, which is surprising because many C tools are based on the assumption that inline assembly is rarely used. Nevertheless, in terms of density, inline assembly is rare, with one fragment per 40 KLOC of C code on average. The density of inline assembly is lower for the popular projects (one fragment per 50 KLOC) than for those selected by keywords (one fragment per 31 KLOC).

[2] The x86 architecture allows adjusting the semantics of an instruction with a prefix. This includes the `lock` prefix for exclusive access to shared memory, and `rep` to repeat an instruction a certain number of times. In inline assembly, these prefixes are denoted as individual instructions (e.g., `lock; cmpxchg`). In our survey, we merged the prefix and its instruction and handled them as a single instruction (e.g., `lock cmpxchg`). The `xchg` instruction has an implicit `lock` prefix when used with a memory operand. For jump-if-condition-is-met instructions and set-on-condition instructions, several mnemonics exist for the same instruction. We grouped such mnemonics and counted them as the same instruction. We also considered different software interrupts as distinct instructions, since their purposes differ markedly.

> **RQ1.1:** 28.1% of the most popular and 11.2% of the keyword-selected projects contained inline assembly.

Number of fragments per project. To measure the number of inline assembly fragments in a project, we considered only unique fragments because duplicates do not increase the implementation effort (see Figure 2). 36.2% of the projects with inline assembly contained only one unique inline assembly fragment. 93.3% of them contained up to ten unique inline assembly fragments. On average, projects analyzed in detail contained 3.7 unique inline assembly fragments (with a median of 2).

> **RQ1.2:** C projects with inline assembly which were analyzed in detail contained on average 3.7 unique inline assembly fragments (median of 2)

Overview of the fragments. In total, we analyzed 1026 fragments, of which 607 were unique per project. Projects that used inline assembly tended to bundle instructions for several operand sizes in the same source file; consequently, we found 715 fragments that were unique within a single file. Overall, we found 197 unique inline assembly fragments.

Analysis of the fragments. Of the 197 projects with inline assembly, we analyzed the inline assembly in 163 projects (82.7%) in detail. To this end, we extracted each fragment and added it together with metadata about the project to our database, which we then queried for aggregate statistics (e.g., the frequency of instructions). The 34 projects that we did not analyze used complicated macro metaprogramming and/or contained an excessive number of large inline assembly fragments, which made our manual analysis approach infeasible. We call these "big-fragment" codebases. They consisted mostly of mature software projects (such as video players) that used inline assembly for SIMD operations, for which they provided several alternative implementations (e.g., AVX, SSE, SSE2). We assumed that tools need to provide close-to-complete SIMD inline assembly support for these projects, and thus omitted them from the detailed analysis.

> **RQ2.1:** 17.3% of all C projects with inline assembly contained macro-metaprogramming and many large inline assembly fragments that were omitted from our detailed analysis.

Instructions in a fragment. When analyzing instructions in inline assembly fragments, we again considered those fragments that were unique to a project. Typically, they were very short (see Figure 1). 390 (64.3%) of them had only one instruction. 73.3% had up to two instructions. However, we also found inline assembly fragments with up to 438 instructions. The average number of instructions in an inline assembly

Table 1. The 10 most common file names that contained inline assembly and their average numbers of instructions

file name	projects	instr.	file name	projects	instr.
sqlite3.c	10	1.0	inffas86.c	4	1.0
atomic.h	8	3.4	mb.h	4	1.0
SDL_endian.h	4	2.0	timing.c	4	1.0
atomic-ops.h	4	2.0	util.h	4	1.0
configure.ac	4	1.0	utils.h	4	2.2

fragment was 9.9; the median was 1. In total, we found only 167 unique instructions, which contrasts with the approximately 1000 instructions that x84-64 provides [25].

> **RQ2.2:** Inline assembly fragments contained on average 9.9 instructions (median of 1) per fragment.

Duplicate fragments. It has been shown that file duplication among GitHub projects—mainly targeting popular libraries copied into many projects—is a common phenomenon [41], which we also observed for the projects we analyzed (see Table 1). For example, many projects contained `sqlite3.c`, which corresponds to the database with the same name (which uses the `rdtsc` instruction), `SDL_endian.h` for the SDL library (which uses inline assembly for endianness conversions), and `inffas86.c` (which implements a compression algorithm using inline assembly). We did not try to eliminate such duplicate files in the analysis, because the duplication is significant: tool authors have a stronger incentive to implement those inline assembly instructions that are used by many projects.

Project domains. Table 2 classifies the projects into domains and shows how many projects per domain contained inline assembly. We created this table by manually labelling the projects using an ad-hoc vocabulary of seventeen domain labels. Note that the domains differ in extent and intersect in some cases. As expected, the majority of projects were `crypto` libraries (with SSL/TLS libraries as a subdomain). However, in general, the domains were relatively diverse. In addition to the eleven domains in the table, we also used seven other domain labels[3] which had fewer than 7 projects each and were omitted for brevity.

> **RQ3:** Inline assembly is used in many domains, most commonly in projects for crypto, networking, media, databases, language implementations, concurrency, ssl, string and math libraries.

[3]These were: games, general-purpose libraries, reverse engineering, garbage collection, monitoring, and virtualization.

Table 2. Domains of projects that used inline assembly (each domain containing at least 7 projects)

domain	projects		description
	#	%	
crypto	23	11.7	encryption and decryption algorithms, cryptographic hashes, non-cryptographic hashes, base64 encodings
networking	20	10.2	protocols, email systems, chat clients, port scanners
media	17	8.6	video and music players and encoders, audio processing software, image libraries
database	16	8.1	databases, key/value storages, other in-memory data structures
language implementation	15	7.6	compilers, interpreters, virtual machines
misc	13	6.6	projects not assigned to any domain
concurrency	9	4.6	concurrency libraries, concurrent data structures
ssl	8	4.1	SSL/TLS libraries
string library	8	4.1	string algorithms, converters between different formats, parsers
math library	7	3.6	scientific applications, math libraries
web server	7	3.6	

4 Use Cases of Inline Assembly Instructions

We identified four typical use cases for inline assembly. One was to prevent instruction reorderings, either in the compiler (prevented by "compiler barriers") or in the processor, both in single-core execution and between multiple cores (prevented by memory barriers and atomic instructions—see Section 4.1). The second use case was performance optimization, for example, for efficient endianness conversions, hash functions, and bitscans (see Section 4.2). The third use case was to interact with the hardware, for example, to detect CPU features, to obtain precise timing information, random numbers, and manage caches (see Section 4.3). The fourth use case was for more general "management" instructions, for example, moving values, pushing and popping from the stack, and arithmetic instructions (see Section 4.4).

Note that there might be more than one reason for using assembly code: for example, programmers might read the elapsed clock cycles using the `rdtsc` instruction because

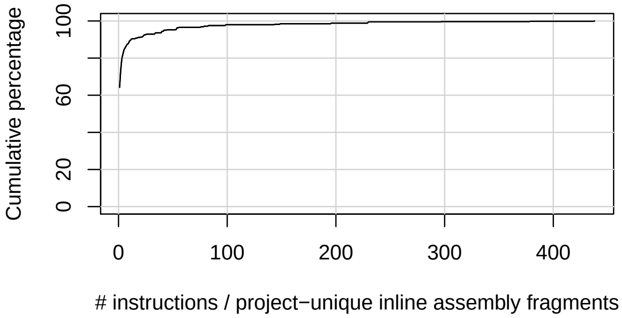

Figure 1. Inline assembly fragment lengths

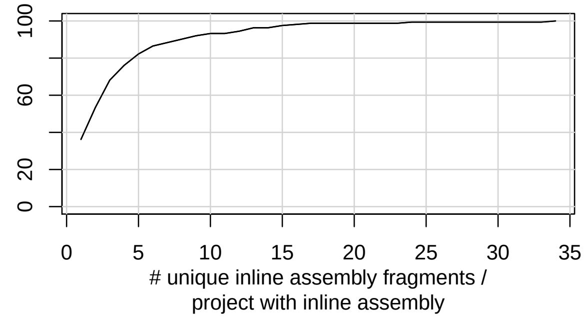

Figure 2. Number of fragments per project

similar C timing functions might not provide the same accuracy; however, they might also use it for efficiency because it has a lower overhead than those functions.

> **RQ4:** Inline assembly is used to ensure correct semantics on multiple cores, for performance optimization, to access functionality that is unavailable in C, and to implement arithmetic operations.

For each use case, we denoted in parentheses the percentage of projects that relied on at least one instruction. Some instructions were counted for several use cases; for example, `xchg` can be used to exchange bytes to convert the endianness of a 16-bit value and has an implicit `lock` prefix when applied to a memory operand, which is why it can also be used to implement an atomic operation.

We found that most inline assembly instructions can also be issued using compiler intrinsics instead of inline assembly (compiler barriers being the only exception). Although compiler intrinsics are specific to a compiler, they are easier to support in tools because they follow the same conventions as C functions, both syntactically and semantically. For example, unlike inline assembly, compiler intrinsics cannot modify local variables.

4.1 Instruction Reordering and Multicore Programming

For threading and concurrency control, most C programs rely on libraries (such as pthreads [9]), compiler intrinsics, and inline assembly instructions. Intrinsics and assembly instructions are used mainly for historical reasons, since atomic operations became standardized only in 2011 [29].

In this section, we describe how inline assembly was used to perform atomic operations and to control the ordering of instructions at the compiler and processor levels.

Atomic instructions (24.0%). In 24.0% of the projects, instructions were prefixed to execute atomically to prevent races when data is accessed by multiple threads (see Table 3). More recent code uses C11 atomic instructions as an alternative; for example, the add-and-fetch operation, which is equivalent to `lock xaddq`, can also be implemented using the C11 `atomic_fetch_add` function.

Compiler barriers (24.0%). C compilers are permitted to reorder instructions that access memory. Unless specially directed, these reorderings are allowed to assume single-threaded execution. A common use case of inline assembly is to implement such a special directive, called a "compiler barrier", telling the compiler to assume an arbitrary side effect to the memory, hence preventing reorderings around the barrier. This is expressed as follows (a `memory` clobber):

```
asm volatile("" : : : "memory");
```

Such barriers are often necessary in lock-free concurrent programming.

Additionally, compiler barriers were used to prevent the compiler from optimizing away instructions. For example, in Listing 1, the compiler is prevented from removing `memset` to implement a `secure_clear` function that can be used to clear sensitive data from memory. A compiler could remove the `memset` call, for example, if the call is inlined and the memory freed, because the compiler can assume that it is no longer accessible [16]. If so, attackers could exploit a buffer overflow at another location in the program to read the data. Note that the C11 standard specifies the function `memset_s`, which provides the same guarantees as the `secure_clear` implementation.

Listing 1. Implementing a secure memory-zeroing function

```
void secure_clear(void *ptr, size_t len) {
    memset(ptr, 0, len);
    asm volatile("" : : "r"(ptr) : "memory");
}
```

Memory barriers (11.2%). Not only compilers, but also processors reorder instructions. Memory barriers are used to prevent reorderings by the processor (see Table 4). On x86, they are mostly needed for special cases (e.g., write-combining memory or non-temporal stores), as all memory accesses except store-load are ordered, so a compiler barrier is often sufficient to ensure the desired ordering [1].

Spin loop hints (15.1%). We found that 27 projects used the `pause` instruction as a processor hint in busy-waiting loops. Busy waiting refers to a tight loop in which a check is performed repeatedly. For example, in spinlocks, a thread repeatedly tries to acquire a lock that has potentially already been acquired by another thread. To remedy the costs of busy waiting in terms of performance and energy, `pause` causes a short delay and controls speculative execution [48].

4.2 Performance Optimizations

Several inline assembly instruction categories were used to optimize performance, even when the code could have been written in pure C.

SIMD instructions (6.1% + 34 projects). In the quantitative analysis, only a few SIMD instructions ranked among the most common instructions, for example, `pxor` and `movdqa` (used in 9 and 8 projects, respectively). However, the actual number of projects using SIMD instructions was higher because the 34 "big-fragment" projects that we did not analyze mostly targeted various SIMD instruction sets (e.g., MMX, SSE, and AVX);

Endianness conversion (25.7%). A common use case of inline assembly is to change the byte order of a value (see Table 5), for example, when the file format of a read file and the processor differ in their endianness. On x86, the `xchg` instruction can be used to swap the bytes of 16-bit integers, because x86 allows both the higher and lower byte of a 16-bit register to be addressed. A less common alternative is to use rotation left or right (`rol` or `ror`) by eight places. For 32-bit and 64-bit values, the `bswap` instruction is used instead. Half of the projects with instructions for endianness conversions included the SDL library [54] as source files in the repository tree. Using inline assembly to implement endianness conversion is most likely a performance optimization that is no longer needed, because state-of-the-art compilers produce as efficient code [22].

Hash functions (15.6%). A number of projects used inline assembly to implement hash functions. This included the `crc32` instruction as well as the `rol`, `ror`, and `shl` instructions to compute the CRC32 and SHA hashsums (see Table 6). The shift and rotate instructions could also simply be implemented in C, and current C compilers produce efficient machine code for them [49].

Bit scan (7.8%). Several projects used bit-scan instructions to determine the most significant one-bit using `bsr` (in 12 projects) or the least significant one-bit using `bsf` (in 7 projects). Both instructions have many applications [68, Sections 5.3 and 5.4]. As `bsr` corresponds to a `log2` function that rounds the result down to the next lower integer, the instruction was often used for this purpose. For an input `value`, it is also possible to round the result up by providing the input `(value<<1)-1`. Bitscan instructions were mostly used by memory allocators such as jemalloc [20] (which was included in four projects) or dlmalloc as well as by compression and math libraries.

Advanced Encryption Standard (AES) instructions (2.2%). We found that 2.2% of the projects used inline assembly to speed up AES using AES-NI instructions.

4.3 Functionality Unavailable in C

Feature detection (28.5%). The `cpuid` instruction allows programs to request information about the processor. It was often used to check cache size, facilities for random-number generation, or support for SIMD instructions such as SSE and AVX. Also, perhaps surprisingly, `cpuid` is defined as a "serializing instruction" in the processor's out-of-order execution semantics, guaranteeing that all instructions preceding it have been executed and none is moved above it.

Clock cycle counter (60.9%). Inline assembly was most commonly used for accurate time measurement using the `rdtsc` instruction. The `rdtsc` instruction reads the timestamp counter provided by the CPU. This instruction is both efficient and accurate for measuring the elapsed cycles, which makes it suitable for benchmarking [27]. As the CPU's out-of-order execution could move the code-to-be-benchmarked before the `rdtsc` instruction, it is typically used together with a serializing instruction (such as `cpuid`) when measuring the elapsed clock cycles. To minimize the overhead when measuring the end time, the `rdtscp` instruction can be used, which also reads the timestamp counter but has serializing properties; to prevent subsequent code from being executed between the start- and end-measuring instructions, another `cpuid` instruction is needed.

Debug interrupts (3.9%). Some projects used an interrupt to programmatically set a breakpoint in the program. If a debugger, such as GDB, is attached to the program, executing a breakpoint causes the program to pause execution. A definition of a breakpoint, for example,

```
#define BREAKPOINT asm("int $0x03")
```

is often selectively enabled through ifdefs, depending on whether the debugging mode in the project is enabled.

Prefetching data (3.9%). The `prefetch` instruction was used in 7 projects. It is a hint to the processor that the memory specified by the operand will be accessed soon, which typically causes it to be moved to the cache. Using a `prefetch` instruction timely can improve performance, because the latency of fetching data can be bridged. However, as processors provide prefetching mechanisms in hardware, using them correctly requires a thorough understanding of cache mechanisms [39]. For example, software prefetches that are issued too early can reduce the effectiveness of hardware prefetching by evicting data that is still being used.

Random numbers (3.4%). The `rdrand` instruction was used in 6 projects. It computes a secure random number with an on-chip random-number generator that uses statistical tests to check the quality of the generated numbers [28].

Table 3. Instructions for atomics (with at least 4 projects using them)

instruction	% projects
lock xchg	14.2
lock cmpxchg	13.2
lock xadd	8.6
lock add	3.0
lock dec	2.5
lock inc	2.5
lock bts	2.0

Table 4. Instruction for fences

instruction	% projects
mfence	6.6
sfence	6.6
lfence	5.6

Table 5. Instructions for endianness conversion

instruction	% projects
lock xchg	14.2
bswap	9.1
ror	5.6
rol	4.6

Table 6. Instructions for hash functions

instruction	% projects
shl	6.1
ror	5.6
rol	4.6
crc32	2.5

Table 7. Instructions for timing

instruction	% projects
rdtsc	27.4
cpuid	25.4
rdtscp	2.5

Table 8. Instructions for feature detection

instruction	% projects
cpuid	25.4
xgetbv	4.1

Table 9. Instructions to move around data

instruction	% projects
mov	24.9
pop	7.1
push	7.1
pushf	1.5
popf	1.0

Programmers can verify successful random-number generation by checking the carry flag (`CF`), for example, by writing its value to a variable (using the `setc` instruction).

4.4 Supporting Instructions

Some instructions were most commonly used together with other instructions, and we therefore classify them as "supporting instructions".

Moving and copying data (30.2%). Some inline assembly fragments, mainly those larger in size, contained instructions to copy data to a register before some other instruction accessed this register (see Table 9). While the `mov` instruction was also used in smaller fragments for that purpose, the instruction could in many cases have been omitted entirely, simply by correctly specifying the input and output constraints and letting the compiler generate the data-movement code. In rarer cases, `mov` was also used to build a stack trace by retrieving the value of `%rbp`. Additionally, the `push` and `pop` instructions were used to save register values on the stack and restore them. The `pushf` and `popf` instructions were used to save and restore processor flags.

Arithmetic operations (21.2%). Arithmetic instructions (see Table 10) were used in larger inline assembly fragments, for example, in vector-reduction arithmetic (e.g., vector summation, inner product, and vector chain product) [47] in crypto and math libraries. Additionally, they were used to implement operations that are not available in standard C.

Table 10. Instructions for arithmetics

instruction	% projects
xor	12.7
add	10.7
mul	6.6
sub	6.6
adc	6.1
lea	5.6
or	5.6
and	4.6
inc	3.6
dec	3.0
neg	3.0

Table 11. Instructions for control flow (with at least 4 projects using them)

instruction	% projects
jmp	9.1
cmp	6.6
jz/je	5.6
jne/jnz	5.1
test	4.6
jb/jnae/jc	4.1
jnb/jae/jnc	3.6
ja/jnbe	2.0
jbe/jna	2.0

Table 12. Instructions that set a value based on a flag

instruction	% projects
sete/setz	5.1
setc/setb	3.6
setne/setnz	2.0

Table 13. Instructions with `rep` prefixes

instruction	% projects
rep movs	3.0
cld	2.0
rep stos	2.0

An example is the `mulq` instruction, which can be used to obtain a 128-bit result when multiplying two 64-bit integers.

Another example is use of the `add` instruction for implementing signed integer addition with wraparound semantics, because signed integer overflow has undefined behavior in C [15]. Inline assembly was also used to implement operations on large integer types; for example, adc was used for multi-word additions, because it adds the value of the carry flag to the addition result (e.g., see [13]).

Control-flow instructions (13.4%). Control-flow-related instructions were mostly confined to larger inline assembly fragments (see Table 11). Some of these instructions compute condition values (test and cmp), while others transfer control flow (e.g., jmp). However, they were also used for indirect calls, for example, when implementing setjmp and longjmp for coroutines. Another example was retrying the rdrand instruction using jnc because it sets CF≠1 if unsuccessful.

Set-byte-on-condition (10.6%). Several projects used instructions that extract a value from the flags register (see Table 12). They were typically used together with instructions that indicate their success via a flag. For example, rdrand sets CF=1 on success, and the flag's value can be used from C by loading it into a variable using setc. As another example, cmpxchg sets ZF=1 if the values in the operand and destination are equal, which can be checked using setz.

No-ops (3.9%). The nop operation was used in 7 projects and does not have any semantic effects. Normally, it is used for instruction alignment to improve performance.

Rep instructions (3.4%). Instructions with a rep prefix were used to implement string operations (see Table 13). The rep prefix specifies that an instruction should be repeated a specified number of times. To control the direction of repetition, cld was used to clear the direction flag.

4.5 Implementing Inline Assembly

One goal was to determine the "low-hanging fruits" when implementing inline assembly. Therefore, the question was how many projects could be supported by implementing only 5% of all x86-64 instructions (i.e., 50 instructions). The result is shown in Table 14. It groups similar instructions that can be easily implemented together in an order that maximizes the number of supported projects with each new group. Note that the order of the implementation makes a difference because a project is considered to be supported only if all the instructions it uses are supported.

First, the timing instructions should be implemented; although rdtscp is seldom used, it is similar to rdtsc and could be implemented together with it. Next would be the feature detection instructions. For tools that execute C code, the feature detection instructions could also be used to indicate that certain features are missing (e.g., SIMD support), which could then guide the program not to use inline assembly for these features. Some instructions could be implemented as "no-ops", as they either have no semantic effect (e.g., prefetch) or are important only when multithreaded execution needs to be modeled or analyzed (e.g., memory fences). Implementing bit operations and atomics, would

Table 14. Instruction groups and the percentage of projects covered by them

Instruction group	Instructions	Supported Projects
Timing	rdtsc, rdtscp	11.0%
Feature detection	cpuid, xgetbv	18.4%
"No-ops"	<compiler barrier>, mfence, sfence, lfence, prefetch, nop, int $0x03, pause, ud2	28.2%
Bit operations	bsr, bsf, or, xor, neg, bswap, shl, rol, ror	41.1%
Atomics	lock xchg, lock cmpxchg, lock xadd, lock add, lock dec, lock inc	51.5%
Moving data	mov, push, pop	58.9%
Checksum	crc32	62.0%
Flag operations	sete/setz, setc/setb, setne/setnz, stc	67.5%
Arithmetics	add, sub, mul, adc, lea, div, imul, sbb, inc, dec	70.6%
Random numbers	rdrand	72.4%
Control flow	jmp, jnb/jae/jnc	76.7%
String operations	rep movsb	77.9%

allow half of the projects to be supported. Finally, by implementing the other instructions in the table (50 in total), tool writers could support 77.9% of the projects that we analyzed in detail and 64.5% when counting also the projects that we did not analyze in detail. An alternative to implementing rep movsb would be int $0x80; however, we thought that this instruction is difficult to implement because it is used for system calls, and thus preferred rep movsb. In general, we believe that the semantics of most instructions in the table are relatively straightforward to support in comparison with some other portions of the instruction set, such as extensions for hardware transactional memory [72].

> **RQ5:** By implementing 50 instructions (5% of x86-64's total number of instructions) tool writers could support 64.5% of all projects that contain inline assembly.

Note that, depending on the tool, another order could be more suitable—tool writers can consult the database to determine the order that best suits their project.

5 Declarative Use Cases of Inline Assembly

Our syntactic analysis naturally turned up uses of the asm keyword, but, perhaps surprisingly, not all of these were for inserting instructions. A small number of projects used it instead for declarative means, for example, to control the behavior of the linker. While many tools can ignore or work around these usages of inline assembly, we discuss some of the examples found as a first step towards characterizing the remaining "soup" of non-C notations used in C codebases, as noted in the *Introduction*. Additionally, we discuss examples in which a mix of instruction representations was used to encode certain instructions.

Specifying assembler names. Some projects use the inline assembly `asm` keyword to specify the names of symbols, thus preventing name mangling. For example,

```
AES_ECB_encrypt(...) asm("AES_ECB_encrypt");
```

is a function declaration with an inline assembly label that specifies its symbol name in the machine code. In this example, the function was implemented in macro assembly, so, in order to guarantee binary compatibility, the name must not be mangled. Labels are also used when the symbol cannot be written in plain C (e.g., because it contains special characters that are forbidden in C), and when symbol names need to be accessible by a native function interface.

Linker warnings. A few projects used inline assembler directives to emit linker warnings when incompatible or deprecated functions of a library were included. C library implementations often make use of this; for example, using

```
asm(".section .gnu.warning.gets; .ascii
  \"Please do not use gets!\"; .text");
```

at global scope causes the linker to emit a warning when the unsecure `gets` function is linked.

Symbol versioning. Several libraries used symbol versioning to refer to older libc functions in order for code compiled on a recent platform to work also on older platforms. A common example is `memcpy`, where current Linux versions link to the relatively new glibc function `memcpy@@GLIBC_2.14`. Most other standard library functions link to older glibc versions; for example, `memset` links to `memset@@GLIBC_2.2.5`. If the most recent `memcpy` is not needed, and older platforms should be supported, one can directly bind `memcpy` to the older 2.2.5 version, for example, using

```
asm(".symver memcpy,memcpy@GLIBC_2.2.5").
```

Register variables. Programmers can use inline assembly to associate local or global variables with a specific register [24]. For example, one could store an interpreter's program counter in the `%rsi` register:

```
register unsigned char *pc asm("%rsi");
```

Such code was used for performance optimization.

Instruction representations. Inline assembly instructions are normally written using mnemonics, which are textual representations of the assembly instructions. However, 16.6% of the projects with inline assembly (27 of 163) deviated from this, either by avoiding instruction mnemonics entirely or by combining mnemonics to surprising effect.

A number of projects denoted the `pause` instruction as `rep; nop`. Even though the `rep (0xF3)` prefix is unspecified for `nop (0x90)`, the resulting opcode corresponds to that of the `pause (0xFE90)` instruction. This works because portions of the prefix-opcode space are, in effect, aliased, and the assembler will accept an aliased combination in place of the more direct encoding. In other cases, instructions were directly specified by their opcode, for example, `.byte 0x0f, 0x01, 0xd0` to represent the `xgetbv` instruction. Some projects even mixed both representations within an instruction; for example, `.byte 0x66; clflush %0` was used to specify `clflushopt`, because prepending `0x66` to the opcode of `clflush (0x0FAE)` yields the opcode for `clflushopt (0x660FAE)`.

Programmers resort to such notations to allow use of older assemblers which fail to recognize mnemonics, but can process opcodes or simpler instructions. Such notations were also used for less common architectures, for example, for VIA's Padlock extensions [65]. While tool writers could treat common patterns not specified by their mnemonics as special cases, canonicalizing them would be more comprehensive, as also rare or unknown combinations of instruction-representations could be supported.

6 Threats to Validity

We used a standard methodology [21] to identify validity threats, which we mitigated where possible. We considered internal validity (i.e., whether we controlled all confounding variables), construct validity (i.e., whether the experiment measured what we wanted to measure), and external validity (i.e., whether our results are generalizable).

6.1 Internal Validity

The greatest threat to internal validity is posed by errors in the analysis. We used a manual best-effort approach to analyze x86-64 inline assembly fragments detected by our string search. It cannot be ruled out that we incorrectly included inline assembly that works only for other architectures (e.g., x86-32), or, conversely, that we rejected some erroneously. To address this, we carefully analyzed inline assembly fragments and repeated analyses when we had doubts or when we found a single inline assembly fragment in several projects, so we believe that errors in the analysis have little impact on the result. A threat in the qualitative analysis is that biases in our judgements influenced the outcome of the study; however, since we also used a quantitative approach, gross distortions or misinterpretations are unlikely.

6.2 Construct Validity

The main threat to construct validity is that we used a source-code-based search to determine the usage of inline assembly. This approach enabled us to analyze the usage of inline assembly independent of conditions such as operating system, compiler and its version, platform, and availability of functions and intrinsics. However, while conducting this survey, we found that some system library headers (which are not part of the project repository) contained inline assembly in macros. For example, GCC provides a `cpuinfo.h` header file that wraps the `cpuid` instruction. We ignored such system header libraries, and inspected only the source code of the projects. While we recognize that this could have had a minor impact on the quantitative analysis, we would expect the qualitative analysis to remain unaffected, as the macros were used for the same purposes as inline assembly fragments. Note that, if the goal had been to analyze what inline assembly instructions are actually executed on a given system, a binary-level approach would have been more appropriate. Similarly, to analyze which instructions appear in built binaries in any particular configuration, analysis after C preprocessing would have been more useful.

6.3 External Validity

There are several threats to external validity, which are given by the scope of our work.

Sample Set. One problem could be that the set of projects is not representative of user-level C. To mitigate this and increase the variety of projects, we employed two different strategies to collect samples for analysis, one based on GitHub stars as a proxy for popularity and one based on keywords. Nevertheless, the number of stars of a project might not reflect its popularity, and our search keyword could also bias the results. While inline assembly could differ in domains not represented in the survey, we believe that the overall results would differ only marginally, given the large body of source code that we examined (1264 projects and 56 million LOC).

OS software. We excluded projects with software that typically forms part of an operating system, which we would expect to use more inline assembly than typical user applications, for example, in order to implement interrupt logic, context switches, clearing pages, and for virtualization extensions [5, 42]. The usage of inline assembly in such projects would best be analyzed separately (especially when considering the size of operating systems), which we will consider as part of future work. The findings of our survey are thus not generalizable to such software.

Macro assembly code. We analyzed only inline assembly in detail and not macro assembly, which is stored in separate files. Macro assembly is used to implement larger program parts. This is reflected in the high average number of 888.3 LOC of macro assembly in the 7.8% of projects that used macro assembler. Note that projects with inline assembly were likely to also contain macro assembly, namely with 33.5%. While inline assembly is syntactically and semantically embedded into C code (e.g., C code can access registers, and inline assembly can access local C variables), macro assembler communicates only via the calling convention of the platform. As macro assembly can be called via native function interfaces by C execution environments and allows modular reasoning by analysis tools, we generally ignored it. Our findings are not generalizable to macro assembly.

Architectures. In our study, we focused on x86 inline assembly. However, when inline assembly was used for a particular use case, it was typically implemented for several common architectures (e.g., x86, ARM, and PowerPC). Most projects provided both x86-32 and x86-64 implementations, which were either the same or only slightly different (also see [30]). In rare cases, x86 lacked an inline assembly implementation that other architectures provided; for example, reversing the individual bits in an integer is available on ARM using the instruction `rbit`, with no equivalent x86 instruction. However, in general, we believe that we would have come to similar conclusions regarding the usage of inline assembly for other mainstream architectures.

GitHub. We performed the survey on open-source GitHub projects, and our findings might not apply to proprietary projects. Additionally, our findings might not be generalizable to older code, where inline assembly may be more frequent, since 89.1% of the projects we analyzed had their first commit in 2008 or later (the year GitHub was launched).

7 Related Work

To the best of our knowledge, inline assembly has to date attracted little research attention, and consequently we consider a wider context of related work.

Linux API usage. Our methodology was inspired by a study of Linux API usage which analyzed the frequency of system calls at the binary level to recommend an implementation order [64]. While we adopted a similar perspective, we analyzed the usage of inline assembly in C projects. Additionally, we directly analyzed the source code because we were interested in inline assembly usage independent of, for example, compilers and compiler versions.

Inline assembly and teaching. Anguita et al. discussed student motivation when learning about assembly-level machine organization in computer architecture classes [2]. In these classes, students were taught instructions that high-level languages lack (e.g., `cpuid` and `rdtsc`) and those that can improve the performance of a program (e.g., by prefetching data or using SIMD instructions). We found strong similarities between those instructions and the most frequently used inline assembly instructions, which further supports the validity of both studies.

Linker. Kell et al. studied the semantic role of linkers in C [34]. As with inline assembly, linker features are used in C programs, but transcend the language. Furthermore, some linker-relevant functionality, such as symbol versioning, is expressed in inline assembly.

C preprocessor. Ernst et al. explored the role of the C preprocessor [17]. As with linker features, the C preprocessor is also relied upon by C programs, but is not part of the language. They found that the preprocessor served—among other purposes—to include inline assembly.

Formal verification. Some formal verification approaches support inline assembly and/or macro assembly [5]. For example, Vx86 translates macro assembly to C code by abstracting its functionality [42]. Manual approaches assume that such inline assembly portions need to be converted to C functions [31]. Note that it is more straightforward to translate macro assembler, because C code mixed with assembler typically exchanges values between registers and variables.

Binary analysis. Tools that analyze or process binaries are widely established [3, 4, 11, 35, 46, 56] and could analyze C projects after they have been compiled to machine code. However, they are not always applicable, for example, when analyzing the high-level semantics of a program or when converting between different source languages.

8 Conclusion

We analyzed 1264 GitHub projects to determine the usage of inline assembly in C projects using both quantitative and qualitative analyses.

Our results demonstrate that inline assembly is relatively common in C projects. 28.1% of the most popular C projects contain inline assembly fragments, even when operating-system-level software, which might be more likely to contain inline assembly, is excluded. Inline assembly fragments typically consist of a single instruction, and most projects with inline assembly contain fewer than ten fragments. We found that the majority of projects use the same subset of instructions: by implementing 50 instructions, tool writers could support as much as 64.5% of all projects that contain inline assembly. 17.3% of the remaining projects use macro-metaprogramming techniques and/or many inline assembly fragments, for example, to benefit from SIMD instruction set extensions. By implementing the remainder of the total of 167 instructions and the SIMD instruction set extensions, tool writers could support the majority of projects "in the wild". Another challenge to implementing inline assembly is that invalid combinations of mnemonics are used that form valid opcodes when converted to machine code.

We found that inline assembly is often used in cryptographic applications. However, networking applications, media applications, databases, language implementations, concurrency libraries, math libraries, text processing and web servers also contain inline assembly. It is therefore likely that tools have to deal with inline assembly, even if they are intended for a specific domain. Inline assembly is used for multicore programming, for example, to implement compiler barriers, memory barriers, and atomics. It is employed for performance optimization, namely for SIMD instructions, endianness conversions, hash functions, and bitscan operations. Further, it is used when a functionality is unavailable in C, for example, for determining the elapsed clock cycles, for feature detection, debug interrupts, data prefetching, and generating secure random numbers. Finally, larger inline assembly fragments use moves, arithmetic instructions, and control flow instructions as "filler" instructions. Interestingly, the inline assembly syntax of compilers is not only used to insert instructions but also to control symbol names, linker warnings, symbol versioning, and register variables.

We believe that the results of our study are important to tool writers who consider adding support for inline assembly. Our study gives guidance on the need for such support and helps to plan and prioritize the implementation of instructions. Additionally, this study could be useful to language designers, as it reveals where plain C is inadequate to a task and where developers fall back on assembler instructions. Finally, compiler writers could obtain feedback on which instructions are frequently used, for example, to handle them specifically in compiler warnings [59] (e.g., by analyzing whether constraints and side effects are specified correctly).

9 Future Work

Our study opened up several directions for future work. One question is how inline assembly influences program correctness, since its use is error-prone; for example, undeclared side effects are not detected by state-of-the-art compilers and might remain as undetected faults or hard-to-debug errors in the source code. This question might be addressed by novel bug-finding tools that specifically target inline assembly. Similarly, an open question is whether compilers handle inline assembly correctly in every case. In recent years, random program generators for testing compilers [50, 60, 71] and other tools [14, 33] have been successful in identifying bugs. Future work could investigate whether generating programs with inline assembly could expose additional compiler bugs. While investigating inline assembly, we found that many programs use compiler intrinsics as an alternative to inline assembly. However, we did not investigate the usage of compiler intrinsics, which could be done as part of a future study. Finally, we believe that our study could be extended, for example, by investigating inline assembly in software (I) that is close to the machine (e.g., in operating systems), (II) in other languages (e.g., in C++), and (III) for other architectures (e.g., for ARM), and by investigating macro assembly.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions, and Ingrid Abfalter for her proofreading and editorial assistance. We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at Johannes Kepler University Linz for their support and contributions. The authors from Johannes Kepler University Linz are funded in part by a research grant from Oracle. Kell was supported by EPSRC Programme Grant "REMS: Rigorous Engineering for Mainstream Systems", EP/K008528/1.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *Computer* 29, 12 (Dec. 1996), 66–76. https://doi.org/10.1109/2.546611

[2] Mancia Anguita and F. Javier Fernández-Baldomero. 2007. Software Optimization for Improving Student Motivation in a Computer Architecture Course. *IEEE Transactions on Education* 50, 4 (Nov 2007), 373–378. https://doi.org/10.1109/TE.2007.906603

[3] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86—A Platform for Analyzing x86 Executables. In *Proceedings of the 14th International Conference on Compiler Construction (CC'05)*. Springer-Verlag, Berlin, Heidelberg, 250–254. https://doi.org/10.1007/978-3-540-31985-6_19

[4] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 23 (Aug. 2010), 84 pages. https://doi.org/10.1145/1749608.1749612

[5] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. 2009. Better avionics software reliability by code verification. In *Proceedings, embedded world Conference, Nuremberg, Germany*.

[6] Ryad Benadjila, Olivier Billet, Shay Gueron, and Matt J. Robshaw. 2009. The Intel AES Instructions Set and the SHA-3 Candidates. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT '09)*. Springer-Verlag, Berlin, Heidelberg, 162–178. https://doi.org/10.1007/978-3-642-10366-7_10

[7] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. (2005).

[8] binutils. 2017. Using as. (2017). https://sourceware.org/binutils/docs/as/index.html (Accessed October 2017).

[9] Hans-J. Boehm. 2005. Threads Cannot Be Implemented As a Library. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 261–268. https://doi.org/10.1145/1065010.1065042

[10] Hudson Borges, André C. Hora, and Marco Tulio Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. 334–344. https://doi.org/10.1109/ICSME.2016.31

[11] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 213–223.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224.

[13] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. 2014. Verifying Curve25519 Software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 299–309. https://doi.org/10.1145/2660267.2660370

[14] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *Proceedings of the 4th International Conference on NASA Formal Methods (NFM'12)*. Springer-Verlag, Berlin, Heidelberg, 120–125. https://doi.org/10.1007/978-3-642-28891-3_12

[15] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2012. Understanding Integer Overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 760–770.

[16] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *Proceedings of the 2015 IEEE Security and Privacy Workshops (SPW '15)*. IEEE Computer Society, Washington, DC, USA, 73–87. https://doi.org/10.1109/SPW.2015.33

[17] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng.* 28, 12 (Dec. 2002), 1146–1170. https://doi.org/10.1109/TSE.2002.1158288

[18] David Evans, John Guttag, James Horning, and Yang Meng Tan. 1994. LCLint: A Tool for Using Specifications to Check Code. (1994), 87–96. https://doi.org/10.1145/193173.195297

[19] David Evans and David Larochelle. 2002. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Softw.* 19, 1 (Jan. 2002), 42–51. https://doi.org/10.1109/52.976940

[20] Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. (2006). https://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf (Accessed October 2017).

[21] Robert Feldt and Ana Magazinius. 2010. Validity Threats in Empirical Software Engineering Research - An Initial Survey. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010), Redwood City, San Francisco Bay, CA, USA, July 1 - July 3, 2010*. 374–379.

[22] Mike Frysinger. 2015. Amd64 [un]fixes in SDL_endian.h. (2015). https://discourse.libsdl.org/t/amd64-un-fixes-in-sdl-endian-h/11792 (Accessed October 2017).

[23] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Thomsen Søren S. 2009. Grøstl - a SHA-3 candidate. In *Symmetric Cryptography (Dagstuhl Seminar Proceedings)*, Helena Handschuh, Stefan Lucks, Bart Preneel, and Phillip Rogaway (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2009/1955

[24] GCC Manual. 2017. Variables in Specified Registers. (2017). https://gcc.gnu.org/onlinedocs/gcc/Explicit-Register-Variables.html (Accessed October 2017).

[25] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 237–250. https://doi.org/10.1145/2908080.2908121

[26] Gerard J Holzmann. 2002. UNO: Static source code checking for user-defined properties. In *Proc. IDPT*, Vol. 2.

[27] Intel. 2010. How To Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. (2010). https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf (Accessed October 2017).

[28] Intel. 2014. Intel® Digital Random Number Generator (DRNG) Software Implementation Guide. (2014). https://software.intel.com/sites/default/files/managed/4d/91/DRNG_Software_Implementation_Guide_2.0.pdf (Accessed October 2017).

[29] International Organization for Standardization. 2011. ISO/IEC 9899:2011. (2011).

[30] Andreas Jaeger. 2003. Porting to 64-bit GNU/Linux Systems. In *Proceedings of the GCC Developers Summit.* 107–121.

[31] Rob Johnson and David Wagner. 2004. Finding User/Kernel Pointer Bugs with Type Inference. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04).* USENIX Association, Berkeley, CA, USA, 9–9.

[32] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014).* ACM, New York, NY, USA, 92–101. https://doi.org/10.1145/2597073.2597074

[33] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017).* IEEE Press, Piscataway, NJ, USA, 590–600.

[34] Stephen Kell, Dominic P. Mulligan, and Peter Sewell. 2016. The Missing Link: Explaining ELF Static Linking, Semantically. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016).* ACM, New York, NY, USA, 607–623. https://doi.org/10.1145/2983990.2983996

[35] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. 2002. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium.* USENIX Association, Berkeley, CA, USA, 191–206.

[36] Robbert Krebbers and Freek Wiedijk. 2015. A Typed C11 Semantics for Interactive Theorem Proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs (CPP '15).* ACM, New York, NY, USA, 15–27. https://doi.org/10.1145/2676724.2693571

[37] John B. Lacy. 1993. CryptoLib: Cryptography in Software. In *Proceedings of the 4th USENIX Security Symposium, Santa Clara, CA, USA, October 4-6, 1993.*

[38] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04).* IEEE Computer Society, Washington, DC, USA, 75–88.

[39] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.* 9, 1, Article 2 (March 2012), 29 pages. https://doi.org/10.1145/2133382.2133384

[40] A. Liu and P. Ning. 2008. TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008).* 245–256. https://doi.org/10.1109/IPSN.2008.47

[41] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéJàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 84 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133908

[42] Stefan Maus, Michal Moskal, and Wolfram Schulte. 2008. Vx86: X86 Assembler Simulated in C Powered by Automated Theorem Proving. (2008), 284–298. https://doi.org/10.1007/978-3-540-79980-1_22

[43] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the De Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).* ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/2908080.2908081

[44] mrigger. 2017. Inline Assembler. (2017). https://github.com/elliotchance/c2go/issues/228 (Accessed October 2017).

[45] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings.* 213–228. https://doi.org/10.1007/3-540-45937-5_16

[46] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007.* 89–100. https://doi.org/10.1145/1250734.1250746

[47] Lionel M. Ni and Kai Hwang. 1985. Vector-Reduction Techniques for Arithmetic Pipelines. *IEEE Trans. Comput.* C-34, 5 (May 1985), 404–411. https://doi.org/10.1109/TC.1985.1676580

[48] Joe Olivas, Mike Chynoweth, and Tom Propst. 2015. Benefitting Power and Performance Sleep Loops. (2015). https://software.intel.com/en-us/articles/benefitting-power-and-performance-sleep-loops (Accessed October 2017).

[49] John Regehr. 2013. Safe, Efficient, and Portable Rotate in C/C++. (2013). https://blog.regehr.org/archives/1063 (Accessed October 2017).

[50] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12).* ACM, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[51] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016).* ACM, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[52] Manuel Rigger, Roland Schatz, Matthias Grimmer, and Hanspeter Mössenböck. 2017. Lenient Execution of C on a Java Virtual Machine: Or: How I Learned to Stop Worrying and Run the Code. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes (ManLang 2017).* ACM, New York, NY, USA, 35–47. https://doi.org/10.1145/3132190.3132204

[53] Manuel Rigger, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. Sulong, and Thanks For All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2018).* https://doi.org/10.1145/3173162.3173174

[54] SDL. 2017. Simple DirectMedia Layer. (2017). https://www.libsdl.org/ (Accessed October 2017).

[55] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012.* 309–318.

[56] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS '08).* Springer-Verlag, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1

[57] Henry Spencer and Geoff Collyer. 1992. #ifdef Considered Harmful, or Portability Experience with C News. In *USENIX Summer 1992 Technical Conference, San Antonio, TX, USA, June 8-12, 1992.* https://www.usenix.org/conference/usenix-summer-1992-technical-conference/ifdef-considered-harmful-or-portability

[58] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11,*

2015. 46–55. https://doi.org/10.1109/CGO.2015.7054186

[59] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and Analyzing Compiler Warning Defects. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 203–213. https://doi.org/10.1145/2884781.2884879

[60] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 849–863. https://doi.org/10.1145/2983990.2984038

[61] Piotr Szczechowiak, Leonardo B. Oliveira, Michael Scott, Martin Collier, and Ricardo Dahab. 2008. NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. (2008), 305–320. https://doi.org/10.1007/978-3-540-77690-1_19

[62] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2012. Configuration Coverage in the Analysis of Large-scale System Software. *SIGOPS Oper. Syst. Rev.* 45, 3 (Jan. 2012), 10–14. https://doi.org/10.1145/2094091.2094095

[63] Lucas Torri, Guilherme Fachini, Leonardo Steinfeld, Vesmar Camara, Luigi Carro, and Érika Cota. 2010. An evaluation of free/open source static analysis tools applied to embedded software. In *2010 11th Latin American Test Workshop*. 1–6. https://doi.org/10.1109/LATW.2010.5550368

[64] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. 2016. A Study of Modern Linux API Usage and Compatibility: What to Support when You're Supporting. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 16, 16 pages. https://doi.org/10.1145/2901318.2901341

[65] VIA. 2005. New VIA PadLock SDK Extends Security Support in VIA C7®/C7®-M Processors for Windows and Linux Software Developers. (2005). https://www.viatech.com/en/2005/11/new-via-padlock-sdk-extends-security-support-in-via-c7c7-m-processors-for-windows-and-linux-software-developers/ (Accessed October 2017).

[66] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined Behavior: What Happened to My Code?. In *Proceedings of the Asia-Pacific Workshop on Systems (APSYS '12)*. ACM, New York, NY, USA, Article 9, 7 pages. https://doi.org/10.1145/2349896.2349905

[67] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 260–275. https://doi.org/10.1145/2517349.2522728

[68] Henry S Warren. 2013. *Hacker's delight.* Pearson Education.

[69] Deng Xu. 2011. [Frama-c-discuss] inline assembly code. (2011). https://lists.gforge.inria.fr/pipermail/frama-c-discuss/2011-March/002589.html (Accessed October 2017).

[70] Zhongxing Xu, Ted Kremenek, and Jian Zhang. 2010. A Memory Model for Static Analysis of C Programs. In *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*. 535–548. https://doi.org/10.1007/978-3-642-16558-0_44

[71] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[72] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 19, 11 pages. https://doi.org/10.1145/2503210.2503232

Appendix

Table 15 shows the instructions sorted by their frequency.

Table 15. Instruction table with instructions that were contained in at least 2 projects

instruction	# projects	% projects	instruction	# projects	% projects	instruction	# projects	% projects
rdtsc	54	27.4	test	9	4.6	aeskeygena	4	2.0
cpuid	50	25.4	jc	8	4.1	cld	4	2.0
mov	49	24.9	movdqa	8	4.1	ja	4	2.0
	43	21.8	shr	8	4.1	jbe	4	2.0
lock xchg	28	14.2	xgetbv	8	4.1	lock bts	4	2.0
pause	27	13.7	bsf	7	3.6	lods	4	2.0
lock cmpxchg	26	13.2	call	7	3.6	pclmulqdq	4	2.0
xor	25	12.7	inc	7	3.6	pslldq	4	2.0
add	21	10.7	int $0x03	7	3.6	psllq	4	2.0
bswap	18	9.1	jnc	7	3.6	psrldq	4	2.0
jmp	18	9.1	nop	7	3.6	rep stos	4	2.0
lock xadd	17	8.6	por	7	3.6	sar	4	2.0
pop	14	7.1	prefetch	7	3.6	setnz	4	2.0
push	14	7.1	setc	7	3.6	stos	4	2.0
cmp	13	6.6	dec	6	3.0	imul	3	1.5
mfence	13	6.6	lock add	6	3.0	lock or	3	1.5
mul	13	6.6	neg	6	3.0	lock sub	3	1.5
sfence	13	6.6	rdrand	6	3.0	movzb	3	1.5
sub	13	6.6	rep movs	6	3.0	pand	3	1.5
adc	12	6.1	crc32	5	2.5	pushf	3	1.5
bsr	12	6.1	lock dec	5	2.5	shrd	3	1.5
shl	12	6.1	lock inc	5	2.5	div	2	1.0
jz	11	5.6	movdqu	5	2.5	emms	2	1.0
lea	11	5.6	pshufd	5	2.5	fldcw	2	1.0
lfence	11	5.6	psrlq	5	2.5	int $0x80	2	1.0
or	11	5.6	rdtscp	5	2.5	jl	2	1.0
ror	11	5.6	ret	5	2.5	ldmxcsr	2	1.0
jnz	10	5.1	aesdec	4	2.0	lock and	2	1.0
setz	10	5.1	aesdeclast	4	2.0	popf	2	1.0
and	9	4.6	aesenc	4	2.0	punpcklb	2	1.0
pxor	9	4.6	aesenclast	4	2.0	punpckldq	2	1.0
rol	9	4.6	aesimc	4	2.0	stmxcsr	2	1.0