x86-64 Instruction Usage among C/C++ Applications

Amogh Akshintala
University of North Carolina at Chapel Hill
aakshintala@cs.unc.edu

Bhushan Jain
University of North Carolina at Chapel Hill
bhushan@cs.unc.edu

Chia-Che Tsai
Texas A&M University
chiache@tamu.edu

Michael Ferdman
Stony Brook University
mferdman@cs.stonybrook.edu

Donald E. Porter
University of North Carolina at Chapel Hill
porter@cs.unc.edu

Abstract

This paper presents a study of x86-64 instruction usage across 9,337 C/C++ applications and libraries in the Ubuntu 16.04 GNU/Linux distribution. We present metrics for reasoning about the relative importance of instructions weighted by the popularity of applications that contain them. From this data, we systematize and empirically ground conventional wisdom regarding the relative importance of various components of an ISA, with particular focus on building binary translation tools. We also verify the representativity of two commonly used benchmark suites, and highlight areas for improvement.

CCS Concepts

• **General and reference** -> **Empirical studies**; • **Software and its engineering** -> **Assembly languages**;

ACM Reference Format:

Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E. Porter. 2019. x86-64 Instruction Usage among C/C++ Applications. In *The 12th ACM International Systems and Storage Conference (SYSTOR '19), June 3–5, 2019, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3319647.3325833

SYSTOR '19, June 3–5, 2019, Haifa, Israel

ACM ISBN 978-1-4503-6749-3/19/06…$15.00
https://doi.org/10.1145/3319647.3325833

1 Introduction

Instruction Set Architectures (ISAs) specify the interface of the CPU. In particular, the ISA dictates what instructions the CPU supports and the exact semantics of those instructions. Although performance and energy characteristics of different implementations of an ISA can vary, binary compatibility is guaranteed across all implementations: software written for a specific ISA is guaranteed to execute on all implementations of that ISA.

Various software tools operate on the ISA: CPU emulators [1, 4] and simulators [6], symbolic execution engines [12], binary translation software [9, 13, 14, 26] (including those used in hypervisors on non-virtualizable ISAs), binary analysis software [28, 32, 36], compilers [15, 23], and bug finding tools [17, 30], among others.

In order for these ISA-level tools to understand their binary inputs, they must implement a model of the ISA, which is a staggering challenge due to the sheer size of today's dominant ISAs. To wit, the Intel x86-64 ISA has 981 unique mnemonics—loosely, a group of instructions that all perform the same operation—and a whopping 3,684 binary instructions [17]. The second most popular ISA, ARM, is similarly gargantuan: the 64-bit variant A64 has 821 mnemonics [3].

In practice, tool developers prioritize development effort on the most "important" instructions — trading completeness for simplicity and a quicker development cycle. For instance, the authors of VMware workstation describe an "on-demand implementation" process, where the x86 binary translator focused on just the instructions needed for a target OS; the entire ISA was never supported, and guest OSes such as OS/2 did not work [11]. Similarly, Amit et al. [2] showed that KVM cannot correctly implement certain obscure x86 behaviors in a guest OS.

Prioritizing instruction support is a natural and ubiquitous engineering trade-off. Some instructions appear in program binaries more frequently than others, e.g., the `MOV` instruction (used to move data) is the most common x86-64 instruction. On the contrary, the `VFMADDSD` instruction, used to express a fused multiplication and addition operation, is relatively rare. Further, many instructions perform similar operations, albeit with subtle distinctions.

The question, then, is what is the basis for assigning priority to instructions? Common approaches include analyzing benchmark suites [5, 10, 16], or execution traces collected in target environments [20]. The ad-hoc nature of this approach leaves many useful questions unanswered: Is the chosen test suite actually representative? What is the path of least effort to support a new ISA in a software tool? What minimum set of instructions must be implemented to run at least one application? What instruction sub-set is sufficient to run the majority of deployed applications?

To paraphrase Hennessy and Patterson [27], the best thing to measure is what actually runs on the user's system. This paper presents and analyzes a dataset collected from static analysis of all x86-64 ELF binaries in the Ubuntu Linux 16.04 GNU/Linux distribution. We leverage package installation frequency, an approximation of a package's importance to users, from Ubuntu and Debian popularity contest data [29, 33], to infer the relative importance of an instruction from the percentage of binaries on a given system that contain that instruction.

We adapt metrics from a prior study of OS API compatibility [34], specifically, *instruction importance* — the relative importance of a given instruction, and *weighted completeness* — the completeness of a system that implements a subset of the ISA. Although this paper focuses on x86-64 and Ubuntu, the methodology and tools used in this study can be generalized to other architectures or operating systems with minimal effort. Overall, we believe our measurements are reasonably representative for deriving insights about modern desktop and server applications.

Our contributions include:

- An instruction occurrence dataset gathered using static analysis of 9,337 open-source applications in the Ubuntu Linux 16.04 repositories.
- Evaluation of conventional wisdom about ISA usage.
- An iterative plan for developing new tools that use the x86-64 ISA.
- Empirical validation of standard benchmarks.
- An instruction occurrence data visualization tool, and the analysis framework used in this study are available at http://x86instructionpop.com/.

2 Data Collection

To answer questions about the relative importance of instructions and other components of an ISA, we collect instruction occurrence frequency from all the ELF binaries in the Ubuntu Linux 16.04 repositories. We begin with a refresher on x86-64 instruction encoding, then explain our methodology and its limitations.

2.1 x86-64 Instruction Encoding

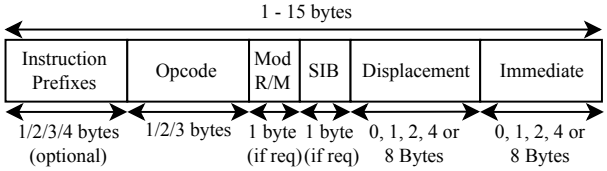

Figure 1: x86-64 instruction encoding format

Figure 1 depicts an overview of the x86-64 instruction encoding scheme, as explained in the Intel manual [19]. Instructions can be 1 to 15 bytes long and must contain an opcode. Opcodes indicate the operation that the instruction performs and may be 1 to 3 bytes long. Programmers usually think of instructions in terms of assembly mnemonics, such as `ADD`, `MOV` or `RET`, which can be encoded by multiple opcodes in x86-64 (e.g., at least 26 different opcodes correspond to the ubiquitous `MOV` instruction.)

Each opcode may also be paired with multiple prefixes that change its semantics. For example, in our corpus, the data movement instruction, `MOV` appears with 61 different prefixes. Some prefixes override the default operand and address sizes: e.g., `0x66` and `0x67` toggle between 16-bit and 32-bit operands, `REX` denotes 64-bit operands, and `VEX` denotes vector operands. A prefix can also indicate special behavior for an instruction, such as atomicity (`LOCK`), repetition (`REPNE`/`REPE`), overriding segment registers (e.g., `0x64` to override the FS segment), etc. Prefixes are generally optional, but may be mandatory for certain instructions. For instance, `REP` is a mandatory prefix for `POPCNT`, an instruction that counts the number of bits that are set in its operand.

The ModR/M byte indicates the registers the instruction operates on, the addressing mode used, and if the SIB byte is present. The SIB byte contains the scale, index and base to use in case the instruction the memory operand is addressed using the indexed or scaled addressing modes (see § ref-sub:addressing). The last two portions of the instruction are used to encode displacement or immediate values, as necessary. Instructions with the same prefixes and opcodes may be of different length, as a result of different operands, and addressing modes: e.g., the most common encoding of the MOV mnemonic (`0x48 8b`) occurs in 5 different sizes — 3, 4, 5, 7, and 8 bytes.

Probability	#Packages	Examples
90 — 100	127	coreutils, bash
80 — 90	52	firefox, apparmor
70 — 80	36	compiz-core, gimp
60 — 70	25	cups, mtools
50 — 60	27	ethtool, diffutils
40 — 50	44	unrar, pidgin
30 — 40	32	gawk, mplayer
20 — 30	72	libreoffice-core
10 — 20	127	inkscape, libkdecore5
1 — 10	917	git, texlive-binaries
0 — 1	6722	openjdk-8-jre-headless

Table 1: Ubuntu packages grouped by probability of being installed in an arbitrary installation of Ubuntu Linux 16.04, along with examples from each group. A package in the 90—100 row is almost always installed, whereas a package in the 0—1 row is installed on, at most, 1% of systems.

2.2 Methodology

Our analysis operates at the granularity of a **package**, the unit of distribution in `APT`. A package typically includes a common library or application. Of the 30,976 packages in the Ubuntu Linux 16.04 repositories, our analysis covered the 9,337 packages that contained ELF binaries. The remaining packages either only contained interpreted code, documentation, were drivers or otherwise LINUX kernel related, or were cross-compiled for a different ISA.

For each package, we calculated its **instruction footprint**, i.e., the set of instructions that occur in the package and their frequencies, by unioning the instruction footprints of the package's constituent executables, libraries, and other dependencies. For each constituent ELF binary, we extracted the start and end of all symbols in the binary, using the `readelf` tool. Each sequence was then linearly disassembled and analyzed: we recorded the opcode, any prefixes, the size of the disassembled instruction, and the mnemonic assigned by the disassembler. For each function or section of code, we also recorded all out-going function calls, operand sizes (aggregated), and the modes in which instructions addressed data (aggregated). In order to appropriately weight library calls in executables, we recursively applied the instruction footprint of any library functions the binary (or library) calls to the instruction footprint of the executable. We obtained this information from the function call data collected during disassembly.

We approximate the **importance** of an instruction by the popularity of the packages that instruction appears in. Intuitively, this is akin to asking: "if this instruction were missing on an ISA implementation, what percent of end-user systems would stop working?". Specifically, we weighted each package's instruction footprint is by its **popularity**, i.e., the probability of the package being installed on an arbitrary installation of Ubuntu Linux 16.04 (shown in Table 1). Each package's popularity was derived from package installation data collected by the Ubuntu [33] and Debian [29] distributions.

To facilitate easy exploration of this data, we built a web-based visualization tool that features filtering among dimensions such as size, mnemonics, opcodes, type of instruction, prefixes, and package installation probability to understand the occurrence of instructions in the packages that we analyzed. This tool and the analysis framework used to gather the data are available at http://x86instructionpop.com/.

2.3 Limitations

Static Analysis applies to a binary at rest, and, in this case, operates on a disassembled instruction stream. In contrast, *dynamic analysis* observes an executing application using hardware traps and debug instructions, or analyzes an instruction trace gathered during prior execution. The quality of a dynamic analysis depends on representativity of inputs provided to the application. Obtaining such a set of representative inputs for all of the packages in the Ubuntu Linux 16.04 repositories is impossible. This effectively rules out dynamic analysis. All data presented in this study was collected using static analysis.

Static analysis has its limitations: it can not account for loops and other code repetition constructs, and also overestimates the importance of rarely executed code blocks, such as error-handling routines. Therefore, it is possible to determine the number of times an instruction occurs in a binary, but not the frequency with which that instruction would be executed at run-time. In recognition of these limitations, our dataset cannot be used to draw conclusions about performance or energy, both of which can only be assessed based on dynamic execution traces.

Linear disassembly. Disassembly may be linear—from the point of entry till the end of a section of code, or recursive—by following the flow of control within the section. Linear static analysis can't identify and avoid *gaps* in the code, such as data in the text section, or alignment-related padding. Recursive disassembly skirts the issue of gaps, by following the flow of control in the binary, typically resulting in higher fidelity data. However, as Zhang and Sekar [37] point out, tracking control flow in binaries is non-trivial. Jump targets are often calculated at execution time, and are incredibly hard to determine during disassembly. Worse, control flow in a program frequently depends on input (e.g., checking the number of arguments provided). Implementing control flow tracking poorly results in incomplete analysis of binaries: parts of the binary are never encountered during analysis, leading to an

incomplete instruction footprint. We used linear disassembly in order to ensure that all code in a binary is visited at least once.

Availability of symbols. When symbol offsets are available, our analysis is able to generate instruction footprints at function granularity. However, when the symbol offset table or the dynamic symbol table are unavailable, the tool computes the instruction footprint and call targets at section granularity. This approach potentially over-estimates the importance of instructions in the binary when the disassembler can not divide the section into its constituent functions—i.e., it assumes any call into the section executes every instruction and outgoing call from the section. We believe this is preferable to the alternative, where one may incorrectly conclude a portion of the binary code is not reachable.

Obfuscated binaries. We assume the binaries in the `APT` repositories are not intentionally obfuscated, and do not include data in text sections.

Binary distribution. `APT` distributes applications as binary, unlike in source-based distributions. This necessitates avoiding optimizations that rely on specific (often newer) hardware features. However, our methodology can be applied to any GNU/Linux distribution.

Popularity Contest data. The Ubuntu and Debian Linux Popularity Contest [29, 33] datasets only include information about software distributed via the `APT` package distribution software. Although `APT` is the primary software distribution mechanism on both of these Linux distributions, some software, often commercial closed-source software, isn't distributed via `APT` and, hence, are omitted from the study. Both Ubuntu and Debian only collect data about package installation, which prevents insight into the usage patterns among the packages, i.e., actual utilization of installed packages. Moreover, the collection of installation data is purely opt-in. However, given that the data draws on a reasonably large number of installations (685,534), we believe it is representative.

ELF binaries only. Our framework considers only ELF binaries. Applications written in interpreted languages are not considered in our study. Rather, the static analysis results of the interpreters (and the dynamic libraries the interpreter uses) are used in place of the application, and weighted according to the application's popularity, on the assumption that the application will not issue instructions that are not in the interpreter binary. We do not consider dynamically-generated code, such as extensions or loadable modules, unless they are distributed in binary form and appear in the application's ELF headers.

Desktop and server applications only. The dominant mobile platforms of the day, Android and iOS, both distribute applications in byte-code formats and do not provide an easy method to access installation data. Our analysis focuses solely on desktop and server platforms.

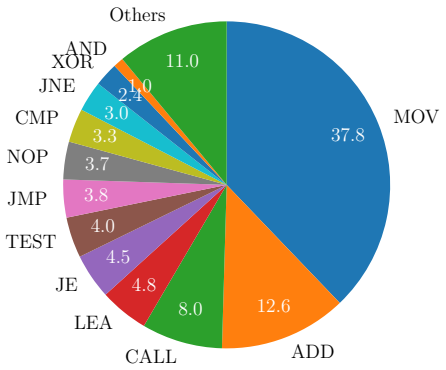

Figure 2: Popularity of instructions, grouped by mnemonic. 89% of the instructions in all of our application binary samples consist of 12 most frequently occurring mnemonics, out of 843 mnemonics in total.

System software. Our dataset excludes the GNU/Linux kernel and all associated drivers and kernel modules, as we do not have a good mechanism to determine the popularity or importance of a given module or kernel configuration.

3 Instruction Occurrence Trends

This section presents the data on instruction frequency, grouped by mnemonics, or the human-readable name (e.g., `MOV`) for one or more binary encodings. Figure 2 shows the top 12 mnemonics, with the remaining 831 mnemonics aggregated under "Others". The `MOV` and `ADD` mnemonics make up about ~50% of all instructions that occur in our binary samples. Unsurprisingly, the top 12 mnemonics are used to express Data Movement (`MOV`), Control Flow (`CALL`, `JE`, `JMP`, `CMP`, `JNE`, `TEST`) or Binary Arithmetic (`AND`, `ADD`, `XOR`). The two remaining mnemonics are `NOP` (no-op) and `LEA`; the former occurs widely because it is typically used by compilers as padding to ensure that branch (call/jump) targets are aligned to cache-line boundaries, while the latter is pervasively used by compilers to calculate memory addresses or for simple arithmetic operations.

Observation: The most common mnemonics are data movement and control flow instructions.

3.1 Distribution by CPU Feature Sets

We also categorize the data using the feature set categories defined in Chapter 5 of the Intel ® 64 and IA-32 Architectures Software Developer's Manual [19]. These results are depicted in Figure 3. The fourth-most frequently occurring category, `MISC`, contains instructions for tasks including flushing the cache (`CLFLUSH`), issuing performance hints

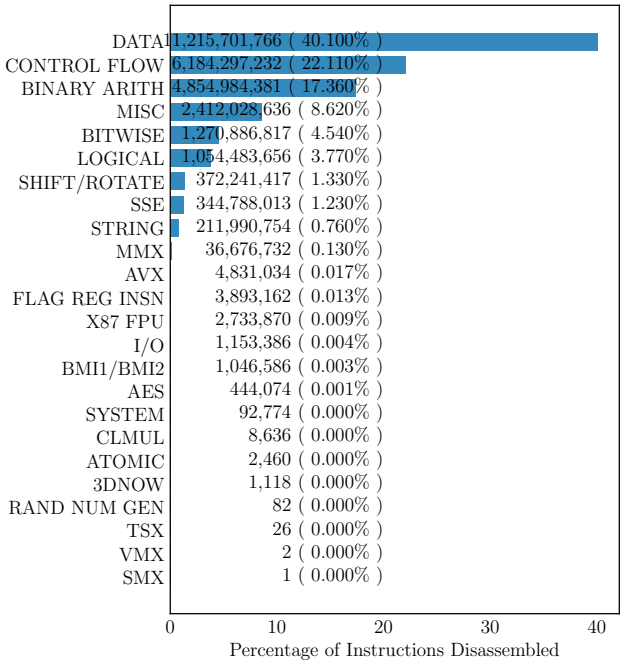

Figure 3: Instruction occurrence categorized by feature sets (raw counts).

(PREFETCH) to the processor, detecting available CPU features (CPUID), loading an address into a register (LEA), and to aligning/padding binary code to cache-line size (NOP). The next two most frequently occurring categories are bitwise assignments and comparisons (BITWISE), and logical operations, such as AND and XOR (LOGICAL). In total, these categories account for over 96% of instruction instances.

The long tail of rarely occurring instruction types include hyper-specialized instructions (e.g., CLMUL — Carry-Less MULtiplication instructions, which occur in 35 packages), older instructions that are not widely used or supported (e.g., 3DNOW — vector instructions introduced by AMD that have since been deprecated), or newly introduced instructions that are not yet widely adopted (e.g., TSX — Transactional Synchronization Instructions.)

3.2 Fraction of Unused x86-64 Instructions

No instructions from the following categories were observed in our data (Figure 3): AVX-512, DECIMAL ARITH, FMA, MPX, SEGMENT, SGX, SHA, XOP. Some of these categories correspond to instructions that were recently introduced or are privileged instructions, such as MPX, SGX, and AVX-512). Other instruction are not used because they are invalid in 64-bit mode, including DECIMAL ARITH and SEGMENT.

Table 2 shows the number of used and unused mnemonics in each CPU feature set category. A mnemonic is labeled

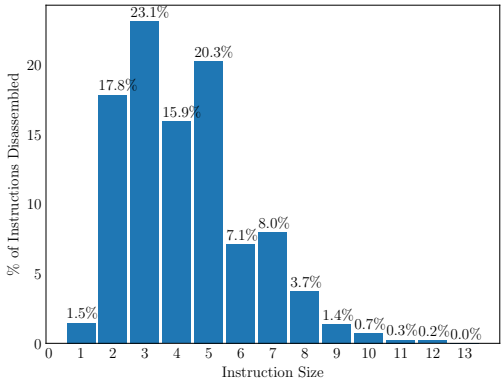

Figure 4: Distribution of instruction sizes.

"unused" if it does not occur once in any package in our corpus. Conversely, a mnemonic is "used" if it occurs in at least one of the packages. We find that the vast majority of unused instructions are either deprecated mnemonics (e.g., 3DNOW, XOP), newly added mnemonics (e.g., SGX, SHA, AVX-512), or aliases to other mnemonics (e.g., the conditional control flow instruction mnemonics JE, JZ are encoded by the same prefix-opcode pairs, and the disassembler emits the JE mnemonic, but not the JZ mnemonic). Specifically, 3DNOW, FMA3, and XOP are vector instruction extensions that were introduced by AMD and subsequently deprecated; although they occasionally occur among the binaries we analyzed, they are usually guarded by code that checks for the availability of these feature sets on the CPU. We present Decimal Arithmetic as a category in Table 2 for completeness, although these instructions are illegal on x86-64.

3.3 Instruction Length

This section evaluates the distribution of instruction lengths in our corpus. The x86-64 ISA has instructions that vary from 1 to 15 bytes. One may wish to understand common and uncommon cases, variable instruction length complicates the CPU's instruction decoder. as the decoder must determine the size of the instruction before it can proceed to other steps or decode the next instruction.

Figure 4 shows the distribution of instructions by length. 3-byte, 4-byte, and 5-byte instructions make up ~60% of the instructions disassembled. Only ~4.63% of the instructions in our corpus are longer than 8-bytes. On the other end, only ~1% of instructions are of length 1-byte. In part, this is because 1-byte opcodes do not have room for operands, and may only contain an opcode, such as PUSHF (push the eflags register onto the stack) and CBW (extend the sign bit of AL into AH). Although 14-byte and 15-byte instructions are theoretically possible, they did not occur in our dataset. The data seems

Instruction Type	Description	Total No. of Insts	Used Insts	Unused Insts
DATA	Data movement/manipulation	63	58	5
CONTROL FLOW	Control Flow Instructions	55	53	2
BINARY ARITH	Binary Arithmetic Instructions	25	22	3
MISC	e.g. LEA CLFLUSH NOP	21	16	5
BITWISE	Bitwise Operations TEST SET POPCNT	40	40	0
LOGICAL	Boolean Logic Instructions	10	10	0
SAR	Shift and Rotate	12	12	0
SSE	Vector Instructions	235	185	50
STRING	String processing	30	12	18
MMX	Multimedia Extensions (64-80 bit vector)	49	49	0
AVX	Advanced Vector Extensions	266	236	30
FLAG INSN	Instructions that deal with Flags Register	22	18	4
X87 FPU	Floating Point Unit Instructions	105	94	11
I/O	Input and Ouput instruction	16	8	8
BMI1/BMI2	Bit Manipulation Instructions	15	7	8
SYSTEM	System related instructions	48	17	31
AES	Advanced Encryption Standard	6	6	0
CLMUL	Carry-less Multiplication Instructions	8	4	4
ATOMIC	Atomic Instructions	1	1	0
3DNOW	Vector instructions introduced by AMD (deprecated)	3	2	1
AVX-512	512-bit Advanced Vector Extensions	260	3	257
DECIMAL ARITH	Data movement and manipulation Instructions	6	0	6
FMA	128-bit and 256-bit Fused Multiply Add instructions	60	25	35
MPX	Memory Protection Extensions	7	0	7
RNG	Random Number Generation	2	2	0
SEGMENT	Segment Register Operations	5	0	5
SGX	Software Guard Extensions	18	0	18
SHA	Hashing related Instructions	7	0	7
SMX	Safer Mode Extensions	6	6	0
TSX	Transactional Synchronization Instructions	6	5	1
VMX	Virtul Machine Extensions	13	1	12
XOP	AMD-specific vector instructions (deprecated)	59	3	56

Table 2: Used and unused mnemonics, grouped by CPU feature set category. An instruction is labeled used if it occurs at least once in any application in the corpus. Conversely, an instruction is said to be unused if never appears in any of the applications.

to be skewed towards smaller instructions, but also seems to follow a power law distribution.

We calculated the variance of instruction sizes across each CPU feature set, as well as, in the entire data set. We observe that instructions are distributed across all lengths, and that new instructions are also spread across multiple parts of the encoding space. A table showing this data is available at http://x86instructionpop.com/length.

Observation: 77% of observed instructions are 2–6 bytes in length. The average instruction is 4.25 bytes long.

3.4 Occurrence of Registers and Memory Operands

The binary encodings for a mnemonic can be of a wide variety of lengths, because x86-64 instructions may operate on values encoded in the instruction itself (immediate), values stored in a register (register), on values stored in memory, or may have implicit operands, e.g., `ret`. In order to access values stored in memory, the location of the value in memory, i.e., its address, must be made available to the processor through the instruction encoding. x86-64 supports several different ways of representing addresses (i.e., addressing schemes), some of which are described below:

- *Absolute*, where the address is specified in the instruction encoding (e.g., `je 0xdeadbeef`)
- *Register-indirect*, where the address is stored in a register (e.g., `sub [rax], rcx`)
- *Displacement*, where the address is computed by adding a displacement to a value in a register (e.g., `mov ecx, [rip + 0xdeadbeef]`)
- *Indexed*, where the address is obtained by adding a computed displacement (or index) to a base address stored in a different register (e.g, `mov r8, [r9 + rax*8]`).
- *Scaled*, which is similar to *indexed* with an additional displacement specified as an immediate (e.g., `lea rcx, [r15 + rcx*1 + 8]`).

Appendix A3, Hennessy and Patterson [27] states "register modes ... account for one-half of the operand references, while memory addressing modes (including immediate) account for the other half." This data was drawn from analysis of 3 applications encoded for the VAX ISA. Our data (shown in Figure 5), however, shows that only ~30% of the instructions operate solely on registers; ~70% of instructions access

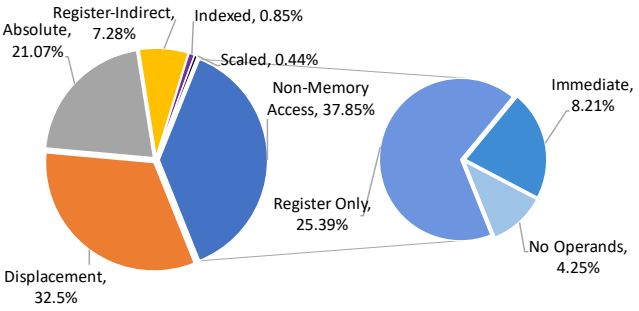

Figure 5: Relative popularity of addressing modes.

Operand Pair	Percentage
Register-Register	19.19
Register-Displacement	28.98
Absolute	18.86
Register-Immediate	8.05
Register	6.19
None	4.25
Register-RegisterIndirect	6.69
Displacement-Immediate	2.46
Register-Absolute	1.79
Displacement-Absolute	0.75
Register-Indexed	0.72
Displacement	0.68
RegisterIndirect-Immediate	0.45
Register-Scaled	0.27
Scaled-Register	0.15
Immediate	0.11

Table 3: Combinations of operands among raw data (%)

memory (we follow the same convention used in the aforementioned reference and include *immediate* with memory access). Most instructions that access memory either contain the memory address in the binary encoding (absolute addressing), or use the simpler register-indirect or displacement addressing modes.

Most instructions operate on more than one operand: some operands are sources of data, some are destinations to store the computed results, and some are both source and destination. These operands are encoded in the instruction using one of the methods described above. Table 3 presents the same data as Figure 5 in terms of the possible combinations of the ways of specifying operands an instruction operates on.

Observation: Instructions operate on memory and registers together more often than on just registers or just memory. The simpler addressing modes (Absolute, Displacement, and Register-indirect) are the most common.

3.5 Vector Instructions

Vector instructions have seen the most churn in the x86-64 ISA. `MMX` instructions, which operated on 80-bit `MMX` registers, were the first vector instructions to be added to the x86-64 ISA. Then came 6 generations of `SSE` — `SSE`, `SSE2`, `SSE3`, `SSSE3`, `SSE4.1`, and `SSE4.2`, all of which operate on 128-bit `XMM` registers. This was followed by the `AVX` instructions, which operate on 256-bit `YMM` registers. The newest member of the x86-64 vector instruction family, `AVX-512`, operates on 512-bit `ZMM` registers and was primarily introduced for the Intel Xeon-Phi platform and is otherwise only available on select Xeon processors.

To understand the adoption of the x86-64 vector extensions, we calculated the number of packages that would be able to run with just a baseline consisting of all non-vector instructions. We then added each of the vector extensions in the order they were introduced to the x86-64 ISA and computed the number of supported packages and the weighted completeness in each case. Our findings are presented in Table 4.

A very small percentage of all packages studied (~0.9%) do not use any vector instructions at all. The older vector extensions, `MMX` and `SSE` (a catch-all category for all 6 SSE versions), are most-widely used. The newest vector instructions have minimal adoption: `AVX` and `AVX-512` instructions occur in only ~50 and 14 packages respectively. This might be an artifact of conservative compilation by `APT` package maintainers.

The most popular MMX instructions are PXOR (Logical Exclusive OR) used by 94% of packages, PCMPEQB (Compare Packaged Data for Equal) used by ~89% of all packages, and MOVQ (Move Quadword) used by ~35% of all packages. The most popular SSE instructions are `MOVAPS` (Move Aligned Packed Single-Precision Floating-Point), which occurs in ~98% of packages, `MOVDQU` (Move Unaligned Double Quadword), which occurs in ~92.3% of all packages, and `MOVDQA` (Move Aligned Double Quadword), which occurs in ~91.4%. These instruction are used to implement the memory copying function `memcpy` where available, because they can read in or write out 128 bytes of memory in one operation. Of the 74 SSE instruction used in at least 1% of all packages, 62 operate on floating point data, 10 operate on integer data, and the remaining 2 store and load MXCSR Control/Status Register — `STMXCSR` (2.65%), and `LDMXCSR` (2.4%). Interestingly, of the top 10 AVX instructions, only 1 operates on floating point data. The other 9 are vector integer operations.

While vector instructions are ubiquitous—they occur in more than 99% of packages—the same small number of vector instructions are used everywhere. This may be an artifact of Ubuntu Linux 16.04 being a binary distribution, i.e., `APT` maintainers tend to compile packages for the most general subset of the x86-64 ISA, instead of using special/newer instructions.

Observation: Vector instructions are indispensable, especially for *memcpy*, but adoption of newer vector instructions is slow.

Description	No. of Mnemonics	No. of Packages Supported	Weighted Completeness
Baseline	340	85	1.553
Baseline + MMX	387	85	1.553
Baseline + SSE	523	483	5.818
Baseline + SSE + MMX	572	9260	99.351
Baseline + SSE + MMX + XOP	575	9260	99.351
Baseline + SSE + MMX + XOP + 3DNOW	577	9265	99.524
Baseline + SSE + MMX + XOP + 3DNOW + AVX	812	9280	99.586
Baseline + SSE + MMX + XOP + 3DNOW + AVX + AVX-512	813	9282	99.587

Table 4: Adoption of vector instructions represented as number of packages supported, and weighted completeness of the system. Baseline includes all non-vector instructions in the following categories: `DATA`, `CONTROL FLOW`, `BINARY ARITHMETIC`, `MISC`, `BITWISE`, `LOGICAL`, `SHIFT AND ROTATE`, `STRING`, `FLAG REG INSN`, `X87 FPU`, `I/O`, `BMI1/BMI2`, `SYTEM`, `AES`, `SYSTEM`, `CLMUL`, `ATOMIC`, `RAND NUM GEN`, `TSX`, `VMX`, `SMX`. Even though implementing all the `MMX` instructions does not support more packages than Baseline, additionally implementing just `SSE` instructions brings the weighted completeness above 0.99.

4 Development and Testing of New Tools

We use the data presented earlier to help developers design and test new binary tools. Given that x86-64 instructions are not used equally in practice, we look at the question of iterative development and testing. In other words, what order of implementing instruction support would have the largest return on effort, in terms of maximizing the number of packages that work for the minimal effort. Or, if one were to implement support for one more instruction, which new instruction would do the most to improve compatibility? In order to answer these questions, we adapt two metrics from Tsai et al. [34]:

Instruction importance estimates the probability of an arbitrary installation of Ubuntu Linux 16.04 including at least one package whose constituent binaries use a given instruction. This metric can help developers understand the relative importance of instructions to end-users. Intuitively, a ubiquitous instruction — one that is found in all applications or in packages that are installed on every Ubuntu Linux 16.04 system — has a very high instruction importance score (~100%). We assume all packages in an installation are indispensable, as the Ubuntu Popcon dataset only captures installation rates, not utilization.

Weighted Completeness measures the completeness of a binary tool, i.e., the percentage of packages from the baseline Ubuntu/Debian Linux distribution the tool can support, weighted by package popularity.

4.1 Developing x86-64 ISA Tools

Binary tools, such as emulators, binary translators, binary instrumentation tools, and decompilers, all operate on ISAs. While x86-64 is extremely complicated, as shown in the prior section, implementing every esoteric feature in the ISA isn't necessary to run common applications. We leverage instruction importance to determine which instructions are essential (Figure 6-top). We find that a small number of instructions,

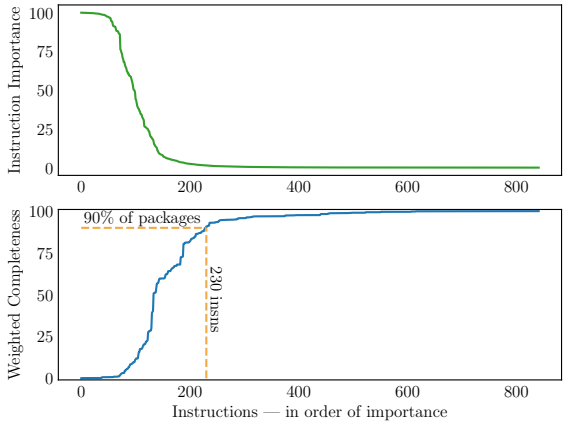

Figure 6: Instruction Importance (top): Distribution of instructions by percent of packages that need the instruction. Weighted completeness(bottom): What percent of packages can execute on a system that follows a greedy implementation strategy?

about 30, are indispensable to all packages. The top 100 most important instructions are used by ~88% of all packages. Importance drops to 10% by the 200 th instruction, and 1% by the 240 th instruction.

Table 5 shows a sample path for the developers to reach 100% completeness, by supporting most of the important packages greedily. Sixty-four instructions (available at http://x86instructionpop.com/mostused.csv), including `RET`, `LEA`, `CALL`, `ADD` are required by more than 90% of packages; however, only 1% of packages use *only* these instructions. Adding another 55 instructions brings the number of supported packages to 27% and another 65 gets us to 80%. An educational or proof-of-concept x86-64 emulator (or some other binary tool) only needs to implement ~230 out of 841 instructions to support 90% of the packages. The long tail of

Stage	Sample Instructions	# Instructions Added (total)	Weighted Completeness(%)
1	RET LEA CALL ADD TEST SUB MOV JNE CMP XOR SHL JE NOP JMP	69	1
2	MUL PMINUB FSTP FLD REPNZ SCAS CMOVLE REP MOVS CMOVBE BT	+55 (124)	27
3	FCHS SETNP BSR SQRTSD SETP MOVD CWDE PMAXUB JNP MFENCE INT3	+65 (189)	80
4	MOVUPD FSUBP PCMPEQD PADDQ FDIV SHLD SHUFPS PANDN MOVS IN	+41 (230)	90
5	everything else	+611 (841)	100

Table 5: Grouping of instructions, optimizing the path to implementation, based on instruction importance.

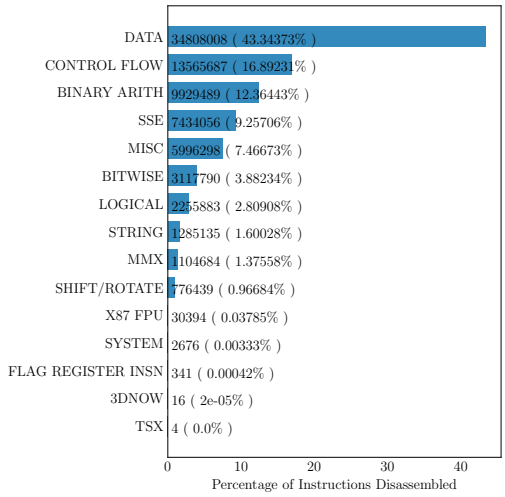

Figure 7: Distribution of Instructions by Category (Raw Data), from the PARSEC and SPEC Benchmarks

instructions after that is needed for completeness. Figure 6-bottom shows the weighted completeness of a system built using this order of implementation.

4.2 Testing x86-64 ISA Tools

Another question of interest to developers is how to comprehensively test their tool with the minimum number of packages, or which package to start with. Our analysis shows that developers only need to evaluate against 55 Ubuntu Linux 16.04 packages, which use all the 894 instructions observed in the Ubuntu dataset. We also observed the *ngetty* package has the smallest instruction footprint, making it a viable first test case while developing a new binary tool, as opposed to the 69 instructions in Stage 1 in Table 5.

5 Instruction distribution Fidelity Of CPU Benchmarks

Developers typically use applications from benchmark suites, such as SPEC or PARSEC, as test cases when developing binary tools. Benchmark suites generally contain applications that are representative of emerging and established workloads. We analyze the SPEC and PARSEC benchmark suites using the same methodology described in Section 2 in order to verify their representativity. These benchmarks were compiled using GCC 5.4.0.

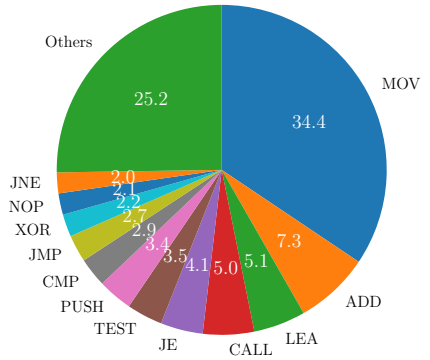

Figure 8: Assembly mnemonics among SPEC and PARSEC benchmarks (%). Total number of instructions: 281.

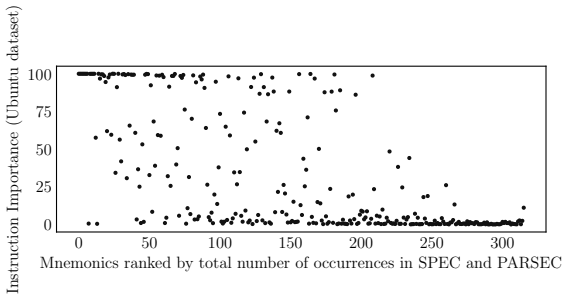

Figure 9: Instruction Importance (in the Ubuntu dataset) of instructions in SPEC and PARSEC (sorted by total number of occurrences in the benhcmarks)

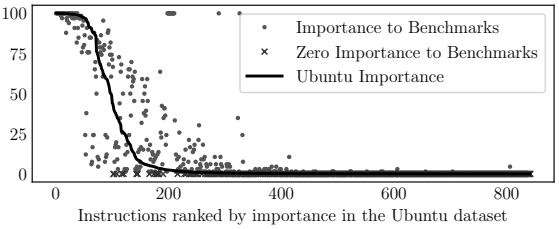

Figure 10: Importance of instructions in the Ubuntu dataset to the SPEC and PARSEC benchmark suites

Figure 7 shows instructions disassembled from SPEC and PARSEC, categorized by CPU feature set. Nine of the twenty-four categories of instructions in our dataset derived from Ubuntu applications (Figure 3) do not occur among the SPEC and PARSEC applications: `AES`, `ATOMIC`, `AVX`, `BMI1/BMI2`,

CLMUL, I/O, RAND NUM GEN, SMX, and VMX. Some of these are newer vector instructions, cryptography-related, privileged, or just too new to reasonably expect to see among the benchmark applications. What is particularly surprising is that ATOMIC instructions are missing.

5.1 Instruction Distribution

Figure 8 shows the most popular instruction mnemonics in the SPEC and PARSEC suites. The general trends from the larger dataset (Figure 2) hold. The distribution of instruction lengths for benchmarks also adheres to the general trend observed in the larger dataset (Figure 4). The total number of mnemonics observed among the benchmark applications is smaller than in the Ubuntu dataset: we see 281 distinct instruction mnemonics among the benchmark applications, compared to the 841 instructions seen among the Ubuntu applications.

Figure 9 shows how the instruction importance in these benchmarks compares to Ubuntu in general. In other words, we show the same left-to-right ordering of instructions as Figure 6, but plot the importance just within these suites. These results are somewhat skewed. The bottom left corner of Figure shows that some instructions of low global importance are used more often in the benchmarks: there is a long tail of instructions used in the benchmarks that are of low importance in our dataset (the dots at 0 importance on the right side). These are mostly vector instructions, which makes sense as these benchmarks are designed to stress the CPU. In addition, we found 4 vector instructions —PMOVZXWD, PHSUBW, PFADD, PMOVZXBW — used in the benchmarks that we had not seen in the Ubuntu dataset. This may be an artifact of the benchmarks being optimized for our test system.

Figure 10 further juxtaposes the Ubuntu instruction importance with the benchmark instruction importance. The crosses in the figure below the line represent instructions that are missing from the benchmarks, but are important to Ubuntu Linux 16.04 applications. The most prominent of these are floating-point control-flow instructions, such as LOOP, JO, and JRCXZ. The dots below the line are instructions under-represented in the benchmark, and those above the line are over-represented in the benchmarks (mostly vector instructions).

Observation: Vector instructions are over-represented among benchmarks, but important floating-point instructions are missing.

6 Related Work

While others [18, 22, 25] have previously studied the relative popularity of instructions in the x86 ISA, their datasets were drawn from a much smaller set of applications. Our dataset is drawn from a large number of applications, and is more representative. Lopes et al. [24] performed a chronological analysis of the x86 ISA usage in Windows and Ubuntu versions released over two decades (1995-2015), and found that many instructions were not used in the default installation. Although we see many low-importance instructions, we observe that almost every instruction is used by at least one package. Their dataset was derived from static analysis of each OS, but did not account for user-installed packages, only those installed by default. Blem et al. [7, 8] studied the differences and similarities between the ARM and x86 ISAs using dynamic analysis on the traces of benchmark workloads. They observed that x86 instruction lengths had low variance, and thus claimed that x86 decode is optimized for common instructions. However, based on our static analysis of all packages in the Ubuntu repository, we see that the instruction lengths vary over a normal distribution as discussed in §3.3. Wiecek et al. [35] studied many of the same trends discussed in §3 for the VAX-11 architecture: Most of our observations on x86-64 are similar to their findings. Rigger et al. [31] examined the usage of in-line assembly among open source projects on GitHub. While our analysis doesn't differentiate between compiler-generated and hand-written assembly, we observe that our findings are complementary to theirs: the relative importance of instructions in the x86-64 ISA is skewed.

Using the Ubuntu and Debian Popularity Contest data to prioritize implementation efforts has been a common theme [21, 34] in prior work; we use the same popularity data to weight instruction occurrence data.

7 Conclusions

This paper contributes a methodology and framework for understanding the relative importance of various components of the x86-64 ISA to applications. We present insights derived from the data we collected by analyzing 9,337 packages from Ubuntu Linux 16.04 APT repositories, and illustrate how developers can leverage this data to iteratively develop new binary tools.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was supported in part by NSF grants CCF-1452904, CNS-1149229, CNS-1700512, and a gift from VMware. This work was completed in part while Akshintala, Jain, Tsai, and Porter were at Stony Brook University, and while Tsai was at The University of California at Berkeley.

References

[1] Keith Adams and Ole Agesen. 2006. A Comparison of Software and Hardware Techniques for x86 Virtualization. *SIGARCH Comput. Archit. News* 34, 5 (Oct. 2006), 2–13. https://doi.org/10.1145/1168919.

1168860

[2] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. 2015. Virtual CPU Validation. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[3] Arm Ltd. [n. d.]. ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile | ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile – Arm Developer. https://developer.arm.com/products/architecture/cpu-architecture/r-profile/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile. https://developer.arm.com/products/architecture/cpu-architecture/r-profile/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile Accessed: 2018-12-11.

[4] Fabrice Bellard and the QEMU team. [n. d.]. QEMU. https://www.qemu.org/. Accessed: 2018-12-18.

[5] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D Hill, and David A Wood. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (May 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[7] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. 2013. Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA) (HPCA '13)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/HPCA.2013.6522302

[8] Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan, and Karthikeyan Sankaralingam. 2015. ISA Wars: Understanding the Relevance of ISA Being RISC or CISC to Performance, Power, and Energy on Modern Architectures. *ACM Transactions on Computer Systems* 33, 1 (March 2015), 3:1–3:34. https://doi.org/10.1145/2699682

[9] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. 2001. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*.

[10] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18)*. ACM, New York, NY, USA, 41–42. https://doi.org/10.1145/3185768.3185771

[11] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. 2012. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.* 30, 4, Article 12 (Nov. 2012), 51 pages. https://doi.org/10.1145/2382553.2382554

[12] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[13] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S Bharadwaj Yadavalli, and John Yates. 1998. FX! 32: A profile-directed binary translator. *IEEE Micro* 2 (1998), 56–64.

[14] Kemal Ebcioğlu and Erik R Altman. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. *ACM SIGARCH Computer Architecture News* 25, 2 (1997), 26–37.

[15] Free Software Foundation (FSF). [n. d.]. GCC, the GNU Compiler Collection. https://gcc.gnu.org/. https://gcc.gnu.org/ Accessed: 2019-4-26.

[16] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (2006), 1–17. https://doi.org/10.1145/1186736.1186737

[17] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. *SIGPLAN Not.* 51, 6 (June 2016), 237–250. https://doi.org/10.1145/2980983.2908121

[18] Amr Hussam Ibrahim, Mohamed Bakr Abdelhalim, Hanadi Hussein, and Ahmed Fahmy. 2011. An Analysis of x86-64 Instruction Set for Optimization of System Softwares. *Planning perspectives: PP* 152 (2011), 162. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.407.5071&rep=rep1&type=pdf

[19] Intel Corporation. 2017. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[20] R. Jagtap, S. Diestelhorst, A. Hansson, M. Jung, and N. When. 2016. Exploring system performance using elastic traces: Fast, accurate and portable. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 96–105. https://doi.org/10.1109/SAMOS.2016.7818336

[21] Bhushan Jain, Chia-Che Tsai, Jitin John, and Donald E Porter. 2014. Practical Techniques to Obviate Setuid-to-Root Binaries. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*.

[22] P Kankowski. 2004. x86 Machine Code Statistics. strchr. com: website. https://www.strchr.com/x86_machine_code_statistics Accessed: 2019-3-6.

[23] 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–. http://dl.acm.org/citation.cfm?id=977395.977673

[24] Bruno Cardoso Lopes, Rafael Auler, Luiz Ramos, Edson Borin, and Rodolfo Azevedo. 2015. SHRINK: Reducing the ISA Complexity via Instruction Recycling. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 311–322. https://doi.org/10.1145/2749469.2750391

[25] Charles Mutigwe, Johnson Kinyua, and Farhad Aghdasi. 2013. Instruction set usage analysis for application-specific systems design. *Int'l Journal of Information Technology and Computer Science* 7, 2 (2013). http://people.umass.edu/cmutigwe/research/ICETCIT-2013/Instruction%20Set%20Usage%20Analysis%20for%20Application-Specific%20Systems%20Design.pdf

[26] Trek Palmer, DD Zovi, and Darko Stefanovic. 2001. SIND: A framework for binary translation. *Department of Computer Science, University of New Mexico* (2001).

[27] David A. Patterson and John L. Hennessy. 2013. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[28] Mathias Payer, Tobias Hartmann, and Thomas R Gross. 2012. Safe loading-a foundation for secure execution of untrusted programs. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 18–32.

[29] Avery Pennarun, Bill Allombert, and Petter Reinholdtsen. 2018. Debian Popularity Contest. http://popcon.debian.org.

[30] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-End Verification of Processors with ISA-Formal. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 42–58.

[31] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. 2018. An Analysis of x86-64 Inline Assembly in C Programs. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '18)*. ACM, New York, NY, USA, 84–99. https://doi.org/10.1145/3186411.3186418

[32] Prateek Saxena, R Sekar, and Varun Puranik. 2008. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 74–83.

[33] The Ubuntu Web Team, Avery Pennarun, Bill Allombert, and Petter Reinholdtsen. 2018. Ubuntu Popularity Contest. http://popcon.ubuntu.com.

[34] 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 ACM European Conference on Computer Systems (EuroSys)*. London, United Kingdom.

[35] Cheryl A Wiecek. 1982. A case study of VAX-11 instruction set usage for compiler execution. In *ACM SIGARCH Computer Architecture News*, Vol. 10. ACM, 177–184.

[36] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries.. In *USENIX Security Symposium*. 337–352.

[37] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the USENIX Security Symposium*. 337–352.