Adapting Software Fault Isolation to Contemporary CPU Architectures

David Sehr Robert Muth Cliff Biffle Victor Khimenko Egor Pasko
Karl Schimpf Bennet Yee Brad Chen

{*sehr,robertm,cbiffle,khim,pasko,kschimpf,bsy,bradchen*}*@google.com*

Abstract

Software Fault Isolation (SFI) is an effective approach to sandboxing binary code of questionable provenance, an interesting use case for native plugins in a Web browser. We present software fault isolation schemes for ARM and x86-64 that provide control-flow and memory integrity with average performance overhead of under 5% on ARM and 7% on x86-64. We believe these are the best known SFI implementations for these architectures, with significantly lower overhead than previous systems for similar architectures. Our experience suggests that these SFI implementations benefit from instruction-level parallelism, and have particularly small impact for workloads that are data memory-bound, both properties that tend to reduce the impact of our SFI systems for future CPU implementations.

1 Introduction

As an application platform, the modern web browser has some noteworthy strengths in such areas as portability and access to Internet resources. It also has a number of significant handicaps. One such handicap is computational performance. Previous work [30] demonstrated how software fault isolation (SFI) can be used in a system to address this gap for Intel 80386-compatible systems, with a modest performance penalty and without compromising the safety users expect from Web-based applications. A major limitation of that work was its specificity to the x86, and in particular its reliance on x86 segmented memory for constraining memory references. This paper describes and evaluates analogous designs for two more recent instruction set implementations, ARM and 64-bit x86, with pure software-fault isolation (SFI) assuming the role of segmented memory.

The main contributions of this paper are as follows:

- A design for ARM SFI that provides control flow and store sandboxing with less than 5% average overhead,
- A design for x86-64 SFI that provides control flow and store sandboxing with less than 7% average overhead, and
- A quantitative analysis of these two approaches on modern CPU implementations.

We will demonstrate that the overhead of fault isolation using these techniques is very low, helping to make SFI a viable approach for isolating performance critical, untrusted code in a web application.

1.1 Background

This work extends Google Native Client [30].[1] Our original system provides efficient sandboxing of x86-32 browser plugins through a combination of SFI and memory segmentation. We assume an execution model where untrusted (hence sandboxed) code is multi-threaded, and where a trusted runtime supporting OS portability and security features shares a process with the untrusted plugin module.

The original NaCl x86-32 system relies on a set of rules for code generation that we briefly summarize here:

- The code section is read-only and statically linked.
- The code section is conceptually divided into fixed sized bundles of 32 bytes.
- All *valid* instructions are reachable by a disassembly starting at a bundle beginning.
- All indirect control flow instructions are replaced by a multiple-instruction sequence (*pseudo-instruction*) that ensures target address alignment to a bundle boundary.
- No instructions or pseudo-instructions in the binary crosses a bundle boundary.

All rules are checked by a verifier before a program is executed. This verifier together with the runtime system comprise NaCls trusted code base (TCB).

For complete details on the x86-32 system please refer to our earlier paper [30]. That work reported an average overhead of about 5% for control flow sandboxing, with the bulk of the overhead being due to alignment considerations. The system benefits from segmented memory to avoid additional sandboxing overhead.

Initially we were skeptical about SFI as a replacement for hardware memory segments. This was based in part on running code from previous research [19], indicating about 25% overhead for x86-32 control+store SFI, which we considered excessive. As we continued

[1] We abbreviate Native Client as "NaCl" when used as an adjective.

our exploration of ARM SFI and sought to understand ARM behavior relative to x86 behavior, we could not adequately explain the observed performance gap between ARM SFI at under 10% overhead with the overhead on x86-32 in terms of instruction set differences. With further study we understood that the prior implementations for x86-32 may have suffered from suboptimal instruction selection and overly pessimistic alignment.

Reliable disassembly of x86 machine code figured largely into the motivation of our previous sandbox design [30]. While the challenges for x86-64 are substantially similar, it may be less clear why analogous rules and validation are required for ARM, given the relative simplicity of the ARM instruction encoding scheme, so we review a few relevant considerations here. Modern ARM implementations commonly support 16-bit Thumb instruction encodings in addition to 32-bit ARM instructions, introducing the possibility of overlapping instructions. Also, ARM binaries commonly include a number of features that must be considered or eliminated by our sandbox implementation. For example, ARM binaries commonly include read-only data embedded in the text segment. Such data in executable memory regions must be isolated to ensure it cannot be used to invoke system call instructions or other instructions incompatible with our sandboxing scheme.

Our architecture further requires the coexistence of trusted and untrusted code and data in the same process, for efficient interaction with the trusted runtime that provides communications and portable interaction with the native operating system and the web browser. As such, indirect control flow and memory references must be constrained to within the untrusted memory region, achieved through sandboxing instructions.

We briefly considered using page protection as an alternative to memory segments [26]. In such an approach, page-table protection would be used to prevent the untrusted code from manipulating trusted data; SFI is still required to enforce control-flow restrictions. Hence, page-table protection can only avoid the overhead of data SFI; the control-flow SFI overhead persists. Also, further use of page protection adds an additional OS-based protection mechanism into the system, in conflict with our requirement of portability across operating systems. This OS interaction is complicated by the requirement for multiple threads that transition independently between untrusted (sandboxed) and trusted (not sandboxed) execution. Due to the anticipated complexity and overhead of this OS interaction and the small potential performance benefit we opted against page-based protection without attempting an implementation.

2 System Architecture

The high-level strategy for our ARM and x86-64 sandboxes builds on the original Native Client sandbox for x86-32 [30], which we will call NaCl-ARM, NaCl-x86-64, and NaCl-x86-32 respectively. The three approaches are compared in Table 1. Both NaCl-ARM and NaCl-x86-64 sandboxes use alignment masks on control flow target addresses, similar to the prior NaCl-x86-32 system. Unlike the prior system, our new designs mask high-order address bits to limit control flow targets to a logical zero-based virtual address range. For data references, stores are sandboxed on both systems. Note that reads of secret data are generally not an issue as the address space barrier between the NaCl module and the browser protects browser resources such as cookies.

In the absence of segment protection, our ARM and x86-64 systems must sandbox store instructions to prevent modification of trusted data, such as code addresses on the trusted stack. Although the layout of the address space differs between the two systems, both use a combination of masking and guard pages to keep stores within the valid address range for untrusted data. To enable faster memory accesses through the stack pointer, both systems maintain the invariant that the stack pointer always holds a valid address, using guard pages at each end to catch escapes due to both overflow/underflow and displacement addressing.

Finally, to encourage source-code portability between the systems, both the ARM and the x86-64 systems use ILP32 (32-bit Int, Long, Pointer) primitive data types, as does the previous x86-32 system. While this limits the 64-bit system to a 4GB address space, it can also improve performance on x86-64 systems, as discussed in section 3.2.

At the level of instruction sequences and address space layout, the ARM and x86-64 data sandboxing solutions are very different. The ARM sandbox leverages instruction predication and some peculiar instructions that allow for compact sandboxing sequences. In our x86-64 system we leverage the very large address space to ensure that most x86 addressing modes are allowed.

3 Implementation

3.1 ARM

The ARM takes many characteristics from RISC microprocessor design. It is built around a load/store architecture, 32-bit instructions, 16 general purpose registers, and a tendency to avoid multi-cycle instructions. It deviates from the simplest RISC designs in several ways:

- condition codes that can be used to predicate most instructions
- "Thumb-mode" 16-bit instruction extensions can improve code density

Feature	NaCl-x86-32	NaCl-ARM	NaCl-x86-64
Addressable memory	1GB	1GB	4GB
Virtual base address	Any	0	44GB
Data model	ILP32	ILP32	ILP32
Reserved registers	0 of 8	0 of 15	1 of 16
Data address mask method	None	Explicit instruction	Implicit in result width
Control address mask method	Explicit instruction	Explicit instruction	Explicit instruction
Bundle size (bytes)	32	16	32
Data embedded in text segment	Forbidden	Permitted	Forbidden
"Safe" addressing registers	All	`sp`	`rsp`, `rbp`
Effect of out-of-sandbox store	Trap	No effect (typically)	Wraps mod 4GB
Effect of out-of-sandbox jump	Trap	Wraps mod 1GB	Wraps mod 4GB

Table 1: Feature Comparison of Native Client SFI schemes. NB: the current release of the Native Client system have changed since the first report [30] was written, where the addressable memory size was 256MB. Other parameters are unchanged.

- relatively complex barrel shifter and addressing modes

While the predication and shift capabilities directly benefit our SFI implementation, we restrict programs to the 32-bit ARM instruction set, with no support for variable-length Thumb and Thumb-2 encodings. While Thumb encodings can incrementally reduce text size, most important on embedded and handheld devices, our work targets more powerful devices like notebooks, where memory footprint is less of an issue, and where the negative performance impact of Thumb encodings is a concern. We confirmed our choice to omit Thumb encodings with a number of major ARM processor vendors.

Our sandbox restricts untrusted stores and control flow to the lowest 1GB of the process virtual address space, reserving the upper 3GB for our trusted runtime and the operating system. As on x86-64, we do not prevent untrusted code from *reading* outside its sandbox. Isolating faults in ARM code thus requires:

- Ensuring that untrusted code cannot execute any forbidden instructions (e.g. undefined encodings, raw system calls).
- Ensuring that untrusted code cannot store to memory locations above 1GB.
- Ensuring that untrusted code cannot jump to memory locations above 1GB (e.g. into the service runtime implementation).

We achieve these goals by adapting to ARM the approach described by Wahbe et al. [28]. We make three significant changes, which we summarize here before reviewing the full design in the rest of this section. First, we reserve no registers for holding sandboxed addresses, instead requiring that they be computed or checked in a single instruction. Second, we ensure the integrity of multi-instruction sandboxing *pseudo-instructions* with a variation of the approach used by our earlier x86-32 system [30], adapted to further prevent execution of embedded data. Finally, we leverage the ARM's fully predicated instruction set to introduce an alternative data address sandboxing sequence. This alternative sequence replaces a data dependency with a control dependency, preventing pipeline stalls and providing better overhead on multiple-issue and out-of-order microarchitectures.

3.1.1 Code Layout and Validation

On ARM, as on x86-32, untrusted program text is separated into fixed-length *bundles*, currently 16 bytes each, or four machine instructions. All indirect control flow must target the beginning of a bundle, enforced at runtime with address masks detailed below. Unlike on the x86-32, we do not need bundles to prevent overlapping instructions, which are impossible in ARM's 32-bit instruction encoding. They are necessary to prevent indirect control flow from targeting the interior of pseudo-instruction and bundle-aligned "trampoline" sequences. The bundle structure also allows us to support data embedded in the text segment, with data bundles starting with an invalid instruction (currently `bkpt 0x7777`) to prevent execution as code.

The validator uses a fall-through disassembly of the text to identify valid instructions, noting the interior of pseudo-instructions and data bundles are not valid control flow targets. When it encounters a direct branch, it further confirms that the branch target is a valid instruction. For indirect control flow, many ARM opcodes can cause a branch by writing `r15`, the program counter. We forbid most of these instructions[2] and consider only explicit branch-to-address-in-register forms such as `bx r0` and their conditional equivalents. This restriction is consistent with recent guidance from ARM for compiler

[2]We do permit the instruction `bic r15, rN, MASK` Although it allows a single-instruction sandboxed control transfer, it can have poor branch prediction performance.

writers. Any such branch must be immediately preceded by an instruction that masks the destination register. The mask must clear the most significant two bits, restricting branches to the low 1GB, and the four least significant bits, restricting targets to bundle boundaries. In 32-bit ARM, the Bit Clear (`bic`) instruction can clear up to eight bits rotated to any even bit position. For example, this pseudo-instruction implements a sandboxed branch through `r0` in eight bytes total, versus the four bytes required for an unsandboxed branch:

```
bic r0, r0, #0xc000000f
bx r0
```

As we do not trust the contents of memory, the common ARM return idiom `pop {pc}` cannot be used. Instead, the return address must be popped into a register and masked:

```
pop { lr }
bic lr, lr, #0xc000000f
bx lr
```

Branching through LR (the link register) is still recognized by the hardware as a return, so we benefit from hardware return stack prediction. Note that these sequences introduce a data dependency between the `bx` branch instruction and its adjacent masking instruction. This pattern (generating an address via the ALU and immediately jumping to it) is sufficiently common in ARM code that the modern ARM implementations [3] can dispatch the sequence without stalling.

For stores, we check that the address is confined to the low 1GB, with no alignment requirement. Rather than destructively masking the address, as we do for control flow, we use a `tst` instruction to *verify* that the most significant bit is clear together with a predicated store:[3]

```
tst r0, #0xc0000000
streq r1, [r0, #12]
```

Like `bic`, `tst` uses an eight-bit immediate rotated to any even position, so the encoding of the mask is efficient. Using `tst` rather than `bic` here avoids a data dependency between the guard instruction and the store, eliminating a two-cycle address-generation stall on Cortex-A8 that would otherwise triple the cost of the added instruction. This illustrates the usefulness of the ARM architecture's fully predicated instruction set. Some predicated SFI stores can also be synthesized in this manner, using sequences such as `tsteq`/`streq`. For cases where the compiler has selected a predicated store that cannot be synthesized with `tst`, we revert to a `bic`-based sandbox, with the consequent address-generation stall.

[3]The `eq` condition checks the Z flag, which `tst` will set if the selected bit is clear.

We allow only base-plus-displacement addressing with immediate displacement. Addressing modes that combine multiple registers to compute an effective address are forbidden for now. Within this limitation, we allow all types of stores, including the Store-Multiple instruction and DMA-style stores through coprocessors, provided the address is checked or masked. We allow the ARM architecture's full range of pre- and post-increment and decrement modes. Note that since we mask only the base address and ARM immediate displacements can be up to $\pm$4095 bytes, stores can access a small band of memory outside the 1GB data region. We use guard pages at each end of the data region to trap such accesses.[4]

3.1.2 Stores to the Stack

To allow optimized stores through the stack pointer, we require that the stack pointer register (SP) always contain a valid data address. To enforce this requirement, we initialize SP with a valid address before activating the untrusted program, with further requirements for the two kinds of instructions that modify SP. Instructions that update SP as a side-effect of a memory reference (for example `pop`) are guaranteed to generate a fault if the modified SP is invalid, because of our guard regions at either end of data space. Instructions that update SP directly are sandboxed with a subsequent masking instruction, as in:

```
mov SP, r1
bic SP, SP, #c0000000
```

This approach could someday be extended to other registers. For example, C-like languages might benefit from a frame pointer handled in much the same way as the SP, as we do for x86-64, while Java and C++ might additionally benefit from efficient stores through `this`. In these cases, we would also permit moves between any two such data-addressing registers without requiring masking.

3.1.3 Reference Compiler

We have modified LLVM 2.6 [13] to implement our ARM SFI design. We chose LLVM because it appeared to allow an easier implementation of our SFI design, and to explore its use in future cross-platform work. In practice we have also found it to produce faster ARM code than GCC, although the details are outside the scope of this paper. The SFI changes were restricted to the ARM target implementation within the `llc` binary, and required approximately 2100 lines of code and table modifications. For the results presented in this paper we used

[4]The guard pages "below" the data region are actually at the top of the address space, where the OS resides, and are not accessible from user mode.

the compiler to generate standard Linux executables with access to the full instruction set. This allows us to isolate the behavior of our SFI design from that of our trusted runtime.

3.2 x86-64

While the mechanisms of our x86-64 implementation are mostly analogous to those of our ARM implementation, the details are very different. As with ARM, a valid data address range is surrounded by guard regions, and modifications to the stack pointer (`rsp`) and base pointer (`rbp`) are masked or guarded to ensure they always contain a valid address. Our ARM approach relies on being able to ensure that the lowest 1GB of address space does not contain trusted code or data. Unfortunately this is not possible to ensure on some 64-bit Windows versions, which rules out simply using an address mask as ARM does. Instead, our x86-64 system takes advantage of more sophisticated addressing modes and use a small set of "controlled" registers as the base for most effective address computations. The system uses the very large address space, with a 4GB range for valid addresses surrounded by large (multiples of 4GB) unmapped/protected regions. In this way many common x86 addressing modes can be used with little or no sandboxing.

Before we describe the details of our design, we provide some relevant background on AMD's 64-bit extensions to x86. Apart from the obvious 64-bit address space and register width, there are a number of performance relevant changes to the instruction set. The x86 has an established practice of using related names to identify overlapping registers of different lengths; for example `ax` refers to the lower 16-bits of the 32-bit `eax`. In x86-64, general purpose registers are extended to 64-bits, with an `r` replacing the `e` to identify the 64 vs. 32-bit registers, as in `rax`. x86-64 also introduces eight new general purpose registers, as a performance enhancement, named `r8` - `r15`. To allow legacy instructions to use these additional registers, x86-64 defines a set of new prefix bytes to use for register selection. A relatively small number of legacy instructions were dropped from the x86-64 revision, but they tend to be rarely used in practice.

With these details in mind, the following code generation rules are specific to our x86-64 sandbox:

- The module address space is an aligned 4GB region, flanked above and below by protected/unmapped regions of 10×4GB, to compensate for scaling (c.f. below)
- A designated register "RZP" (currently `r15`) is initialized to the 4GB-aligned base address of untrusted memory and is read-only from untrusted code.
- All `rip` update instructions must use `RZP`.

To ensure that `rsp` and `rbp` contain a valid data address we use a few additional constraints:

- `rbp` can be modified via a copy from `rsp` with no masking required.
- `rsp` can be modified via a copy from `rbp` with no masking required.
- Other modifications to `rsp` and `rbp` must be done with a pseudo-instruction that post-masks the address, ensuring that it contains a valid data address.

For example, a valid `rsp` update sequence looks like this:

```
%esp = %eax
lea (%RZP, %rsp, 1), %rsp
```

In this sequence the assignment[5] to `esp` guarantees that the top 32-bits of `rsp` are cleared, and the subsequent add sets those bits to the valid base. Of course such sequences must always be executed in their entirety. Given these rules, many common store instructions can be used with little or no sandboxing required. `Push, pop` and `near call` do not require checking because the updated value of `rsp` is checked by the subsequent memory reference. The safety of a store that uses `rsp` or `rbp` with a simple 32-bit displacement:

```
mov disp32(%rsp), %eax
```

follows from the validity invariant on `rsp` and the guard ranges that absorb the displacement, with no masking required. The most general addressing expression for an allowed store combines a valid base register (`rsp`, `rbp` or `RZP`) with a 32-bit displacement, a 32-bit index, and a scaling factor of 1, 2, 4, or 8. The effective address is computed as:

```
basereg + indexreg * scale + disp32
```

For example, in this pseudo-instruction:

```
add $0x00abcdef, %ecx
mov %eax, disp32(%RZP, %rcx, scale)
```

the upper 32 bits of `rcx` are cleared by the arithmetic operation on `ecx`. Note that any operation on `ecx` will clear the top 32 bits of `rcx`. This required masking operation can often be combined other useful operations. Note that this general form allows generation of addresses in a range of approximately 100GB, with the

[5] We have used the = operation to indicate assignment to the register on the left hand side. There are several instructions, such as `lea` or `movzx` that can be used to perform this assignment. Other instructions are written using ATT syntax.

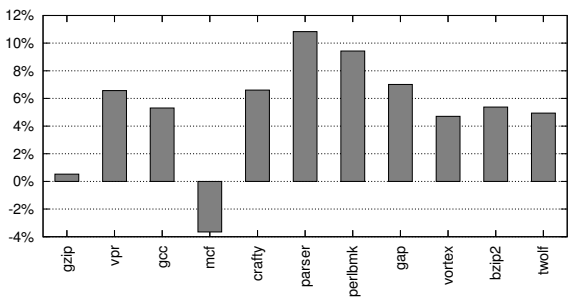

Figure 1: SPEC2000 SFI Performance Overhead for the ARM Cortex-A9.

valid 4GB range near the middle. By reserving and unmapping addresses outside the 4GB range we can ensure that any dereference of an address outside the valid range will lead to a fault. Clearly this scheme relies heavily on the very large 64-bit address space.

Finally, note that updates to the instruction pointer must align the address to 0 mod 32 and initialize the top 32-bits of address from RZP as in this example using `rdx`:

```
%edx = ...
and 0xfffffe0, %edx
lea (%RZP, %rdx, 1), %rdx
jmp *%rdx
```

Our x86-64 SFI implementation is based on GCC 4.4.3, requiring a patch of about 2000 lines to the compiler, linker and assembler source. At a high level, the changes include supporting the new call/return sequences, making pointers and longs 32 bits, allocating `r15` for use as `RZB`, and constraining address generation to meet the above rules.

4 Evaluation

In this section we evaluate the performance of our ARM and x86-64 SFI schemes by comparing against the relevant non-SFI baselines, using C and benchmarks from SPEC2000 INT CPU [12]. Our main analysis is based on out-of-order CPUs, with additional measurements for in-order systems at the end of this section. The out-of-order systems we used for our experiments were:

- For x86-64, a 2.4GHz Intel Core 2 Quad with 8GB of RAM, running Ubuntu Linux 8.04, and
- For ARM, a 1GHz Cortex-A9 (Nvidia Tegra T20) with 512MB of RAM, running Ubuntu Linux 9.10.

4.1 ARM

For ARM, we compared LLVM 2.6 [13] to the same compiler modified to support our SFI scheme. Figure 1 summarizes the ARM results, with tabular data in Table 2. Average overhead is about 5% on the out-of-order Cortex-A9, and is fairly consistent across the benchmarks. Increases in binary size (Table 3) are comparable at around 20% (generally about 10% due to alignment padding and 10% due to added instructions, shown in the rightmost columns of the table). We believe the observed overhead comes primarily from the increase in code path length. For `mcf`, this benchmark is known to be data-cache intensive [17], a case in which the additional sandboxing instructions have minimal impact, and can sometimes be hidden by out-of-order execution on the Cortex-A9. We see the largest slowdowns for `gap`, `gzip`, and `perlbmk`. We suspect these overheads are a combination of increased path length and instruction cache penalties, although we do not have access to ARM hardware performance counter data to confirm this hypothesis.

	x86-64 SFI	SFI vs. -m32	SFI vs. -m64	ARM SFI
164.gzip	16.0	0.82	16.0	0.53
175.vpr	1.60	-5.06	1.60	6.57
176.gcc	35.1	35.1	33.0	5.31
181.mcf	1.34	1.34	-42.6	-3.65
186.crafty	29.3	-8.17	29.3	6.61
197.parser	-4.07	-4.07	-20.3	10.83
253.perlbmk	34.6	26.6	34.6	9.43
254.gap	-4.46	-4.46	-5.09	7.01
255.vortex	43.0	26.0	43.0	4.71
256.bzip2	21.6	4.84	21.6	5.38
300.twolf	0.80	-3.08	0.80	4.94
geomean	14.7	5.24	6.9	5.17

Table 2: SPEC2000 SFI Performance Overhead (percent). The first column compares x86-64 SFI overhead to the "oracle" baseline compiler.

	ARM	ARM SFI	%inc.	%pad
164.gzip	73	90	24	13
175.vpr	225	271	20	13
176.gcc	1586	1931	22	14
181.mcf	84	103	23	12
186.crafty	320	384	20	12
197.parser	219	265	21	12
253.perlbmk	812	1009	24	14
254.gap	531	636	20	11
255.vortex	720	845	17	13
256.bzip2	74	92	24	13
300.twolf	289	343	19	11

Table 3: ARM SPEC2000 text segment size in kilobytes, with % increase and % padding instructions.

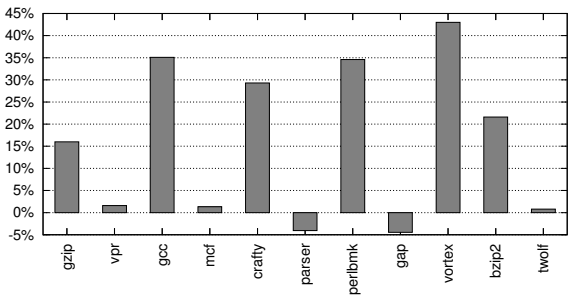

Figure 2: SPEC2000 SFI Performance Overhead for x86-64. SFI performance is compared to the faster of `-m32` and `-m64` compilation.

4.2 x86-64

Our x86-64 comparisons are based on GCC 4.4.3. The selection of a performance baseline is not straightforward. The available compilation modes for x86 are either 32-bit (ILP32, `-m32`) or 64-bit (LP64, `-m64`). Each represents a performance tradeoff, as demonstrated previously [15, 25]. In particular, the 32-bit compilation model's use of ILP32 base types means a smaller data working set compared to standard 64-bit compilation in GCC. On the other hand, use of the 64-bit instruction set offers additional registers and a more efficient register-based calling convention compared to standard 32-bit compilation. Ideally we would compare our SFI compiler to a version of GCC that uses ILP32 and the 64-bit instruction set, but without our SFI implementation. In the absence of such a compiler, we consider a hypothetical compiler that uses an oracle to automatically select the faster of `-m32` and `-m64` compilation. Unless otherwise noted all GCC compiles used the `-O2` optimization level.

Figure 2 and Table 2 provide x86-64 results, where average SFI overhead is about 5% compared to `-m32`, 7% compared to `-m64` and 15% compared to the oracle compiler. Across the benchmarks, the distribution is roughly bi-modal. For `parser` and `gap`, SFI performance is better than either `-m32` or `-m64` binaries (Table 4). These are also cases where `-m64` execution is slower than `-m32`, indicative of data-cache pressure, leading us to believe that the beneficial impact additional registers dominates SFI overhead. Three other benchmarks (`vpr`, `mcf` and `twolf`) show SFI impact is less than 2%. We believe these are memory-bound and do not benefit significantly from the additional registers.

At the other end of the range, four benchmarks, `gcc`, `crafty`, `perlbmk` and `vortex` show performance overhead greater than 25%. All run as fast or faster for `-m64` than `-m32`, suggesting that data-cache pressure does not dominate their performance. `Gcc`, `perlbmk` and `vortex` have large text, and we suspect SFI code-size increase may be contributing to instruction cache pressure. From hardware performance counter data, `crafty` shows a 26% increase in instructions retired and an increase in branch mispredicts from 2% to 8%, likely contributors to the observed SFI performance overhead. We have also observed that `perlbmk` and `vortex` are very sensitive to `memcpy` performance. Our x86-64 experiments are using a relative simple implementation of `memcpy`, to allow the same code to be used with and without the SFI sandbox. In our continuing work we are adapting a tuned `memcpy` implementation to work within our sandbox.

	-m32	-m64	SFI
164.gzip	122	106	123
175.vpr	87	81.3	82.6
176.gcc	47.3	48.0	63.9
181.mcf	59.5	105	60.3
186.crafty	60	42.6	55.1
197.parser	123	148	118
253.perlbmk	86.9	81.7	110
254.gap	60.5	60.9	57.8
255.vortex	99.2	87.4	125
256.bzip2	99.2	85.5	104
300.twolf	130	125	126

Table 4: SPEC2000 x86-64 execution times, in seconds.

	-m32	-m64	SFI
164.gzip	82	85	155
175.vpr	239	244	350
176.gcc	1868	2057	3452
181.mcf	20	23	33
186.crafty	286	257	395
197.parser	243	265	510
253.perlbmk	746	835	1404
254.gap	955	1015	1641
255.vortex	643	620	993
256.bzip2	98	95	159
300.twolf	375	410	617

Table 5: SPEC2000 x86 text sizes, in kilobytes.

4.3 In-Order vs. Out-of-Order CPUs

We suspected that the overhead of our SFI scheme would be hidden in part by CPU microarchitectures that better exploit instruction-level parallelism. In particular, we suspected we would be helped by the ability of out-of-order CPUs to schedule around any bottlenecks that SFI introduces. Fortunately, both architectures we tested have multiple implementations, including recent products with in-order dispatch. To test our hypothesis, we ran a subset of our benchmarks on in-order machines:

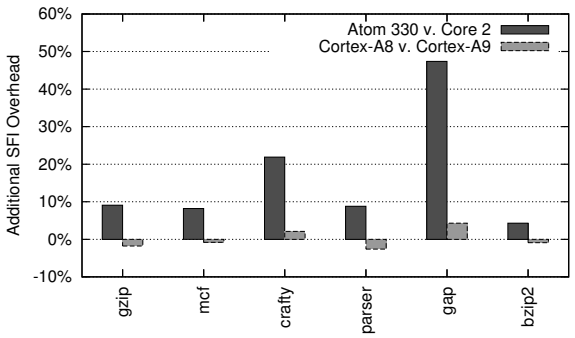

Figure 3: Additional SPEC2000 SFI overhead on in-order microarchitectures.

	Core 2	Atom 330	A9	A8
164.gzip	16.0	25.1	4.4	2.6
181.mcf	-42.6	-34.4	-0.2	-1.0
186.crafty	29.3	51.2	4.2	6.3
197.parser	-20.3	-11.5	3.2	0.6
254.gap	-5.09	42.3	3.4	7.7
256.bzip2	21.6	25.9	2.9	2.0
geomean	6.89	18.5	3.0	3.0

Table 6: Comparison of SPEC2000 overhead (percent) for in-order vs. out-of-order microarchitecture.

- A 1.6GHz Intel Atom 330 with 2GB of RAM, running Ubuntu Linux 9.10.
- A 500MHz Cortex-A8 (Texas Instruments OMAP3540) with 256MB of RAM, running Ångström Linux.

The results are shown in Figure 3 and Table 6. For our x86-64 SFI scheme, the incremental overhead can be significantly higher on the Atom 330 compared to a Core 2 Duo. This suggests out-of-order execution can help hide the overhead of SFI, although other factors may also contribute, including much smaller caches on the Atom part and the fact that GCC's 64-bit code generation may be biased towards the Core 2 microarchitecture. These results should be considered preliminary, as there are a number of optimizations for Atom that are not yet available in our compiler, including Atom-specific instruction scheduling and better selection of no-ops. Generation of efficient SFI code for in-order x86-64 systems is an area of continuing work.

The story on ARM is different. While some benchmarks (notably `gap`) have higher overhead, some (such as `parser`) have equally reduced overhead. We were surprised by this result, and suggest two factors to account for it. First, microarchitectural evaluation of the Cortex-A8 [3] suggests that the instruction sequences produced by our SFI can be issued without encountering a hazard that would cause a pipeline stall. Second, we suggest that the Cortex-A9, as the first widely-available out-of-order ARM chip, might not match the maturity and sophistication of the Core 2 Quad.

5 Discussion

Given our initial goal to impact execution time by less than 10%, we believe these SFI designs are promising. At this level of performance, most developers targeting our system would do better to tune their own code rather than worry about SFI overhead. At the same time, the geometric mean commonly used to report SPEC results does a poor job of capturing the system's performance characteristics; nobody should expect to get "average" performance. As such we will continue our efforts to reduce the impact of SFI for the cases with the largest slowdowns.

Our work fulfills a prediction that the costs of SFI would become lower over time [28]. While thoughtful design has certainly helped minimize SFI performance impact, our experiments also suggest how SFI has benefited from trends in microarchitecture. Out-of-order execution, multi-issue architectures, and the effective gap between memory speed and CPU speed all contribute to reduce the impact of the register-register instructions used by our sandboxing schemes.

We were surprised by the low overhead of the ARM sandbox, and that the x86-64 sandbox overhead should be so much larger by comparison. Clever ARM instruction encodings definitely contributed. Our design directly benefits from the ARM's powerful bit-clear instruction and from predication on stores. It usually requires one instruction per sandboxed ARM operation, whereas the x86-64 sandbox frequently requires extra instructions for address calculations and adds a prefix byte to many instructions. The regularity of the ARM instruction set and smaller bundles (16 vs. 32 bytes) also means that less padding is required for the ARM, hence less instruction cache pressure. The x86-64 design also induces branch misprediction through our omission of the `ret` instruction. By comparison the ARM design uses the normal return idiom hence minimal impact on branch prediction. We also note that the x86-64 systems are generally clocked at a much higher rate than the ARM systems, making the relative distance to memory a possible factor. Unfortunately we do not have data to explore this question thoroughly at this time.

We were initially troubled by the result that our system improves performance for so many benchmarks compared to the common `-m32` compilation mode. This clearly results from the ability of our system to leverage features of the 64-bit instruction set. There is a sense in which the comparison is unfair, as running a 32-bit binary on a 64-bit machine leaves a lot of resources idle.

Our results demonstrate in part the benefit of exploiting those additional resources.

We were also surprised by the magnitude of the positive impact of ILP32 primitive types for a 64-bit binary. For now our x86-64 design benefits from this as yet unexploited opportunity, although based on our experience the community might do well to consider making ILP32 a standard option for x86-64 execution.

In our continuing work we are pursuing opportunities to reduce SFI overhead of our x86-64 system, which we do not consider satisfactory. Our current alignment implementation is conservative, and we have identified a number of opportunities to reduce related padding. We will be moving to GCC version 4.5 which has instruction-scheduling improvements for in-order Atom systems. In the fullness of time we look forward to developing an infrastructure for profile-guided optimization, which should provide opportunities for both instruction cache and branch optimizations.

6 Related Work

Our work draws directly on Native Client, a previous system for sandboxing 32-bit x86 modules [30]. Our scheme for optimizing stack references was informed by an earlier system described by McCamant and Morrisett [18]. We were heavily influenced by the original software fault isolation work by Wahbe, Lucco, Anderson and Graham [28].

Although there is a large body of published research on software fault isolation, we are aware of no publications that specifically explore SFI for ARM or for the 64-bit extensions of the x86 instruction set. SFI for SPARC may be the most thoroughly studied, being the subject of the original SFI paper by Wahbe et al. [28] and numerous subsequent studies by collaborators of Wahbe and Lucco [2, 16, 11] and independent investigators [4, 5, 8, 9, 10, 14, 22, 29]. As this work matured, much of the community's attention turned to a more virtual machine-oriented approach to isolation, incorporating a trusted compiler or interpreter into the trusted core of the system.

The ubiquity of the 32-bit x86 instruction set has catalyzed development of a number of additional sandboxing schemes. MiSFIT [23] contemplated use of software fault isolation to constrain untrusted kernel modules [24]. Unlike our system, they relied on a trusted compiler rather than a validator. SystemTAP and XFI [21, 7] further contemplate x86 sandboxing schemes for kernel extension modules. McCamant and Morrisett [18, 19] studied x86 SFI towards the goals of system security and reducing the performance impact of SFI.

Compared to our sandboxing schemes, CFI [1] provides finer-grained control flow integrity. Whereas our systems only guarantee indirect control flow will target an aligned address in the text segment, CFI can restrict a specific control transfer to a fairly arbitrary subset of known targets. While this more precise control is useful in some scenarios, such as ensuring integrity of translations from higher-level languages, our use of alignment constraints helps simplify our design and implementation. CFI also has somewhat higher average overhead (15% on SPEC2000), not surprising since its instrumentation sequences are longer than ours. XFI [7] adds to CFI further integrity constraints such as on memory and the stack, with additional overhead. More recently, BGI [6] considers an innovative scheme for constraining the memory activity of device drivers, using a large bitmap to track memory accessibility at very fine granularity. None of these projects considered the problem of operating system portability, a key requirement of our systems.

The Nooks system [26] enhances operating system kernel reliability by isolating trusted kernel code from untrusted device driver modules using a transparent OS layer called the Nooks Isolation Manager (NIM). Like Native Client, NIM uses memory protection to isolate untrusted modules. As the NIM operates in the kernel, x86 segments are not available. The NIM instead uses a private page table for each extension module. To change protection domains, the NIM updates the x86 page table base address, an operation that has the side effect of flushing the x86 Translation Lookaside Buffer (TLB). In this way, NIM's use of page tables suggests an alternative to segment protection as used by NaCl-x86-32. While a performance analysis of these two approaches would likely expose interesting differences, the comparison is moot on the x86 as one mechanism is available only within the kernel and the other only outside the kernel. A critical distinction between Nooks and our sandboxing schemes is that Nooks is designed only to protect against unintentional bugs, not abuse. In contrast, our sandboxing schemes must be resistant to attempted deliberate abuse, mandating our mechanisms for reliable x86 disassembly and control flow integrity. These mechanisms have no analog in Nooks.

Our system uses a static validator rather than a trusted compiler, similar to validators described for other systems [7, 18, 19, 21], applying the concept of proof-carrying code [20]. This has the benefit of greatly reducing the size of the trusted computing base [27], and obviates the need for cryptographic signatures from the compiler. Apart from simplifying the security implementation, this has the further benefit of opening our system to 3rd-party tool chains.

7 Conclusion

This paper has presented practical software fault isolation systems for ARM and for 64-bit x86. We believe

these systems demonstrate that the performance overhead of SFI on modern CPU implementations is small enough to make it a practical option for general purpose use when executing untrusted native code. Our experience indicates that SFI benefits from trends in microarchitecture, such as out-of-order and multi-issue CPU cores, although further optimization may be required to avoid penalties on some recent low power in-order cores. We further found that for data-bound workloads, memory latency can hide the impact of SFI.

Source code for Google Native Client can be found at: `http://code.google.com/p/nativeclient/`.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, November 2005.

[2] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. *SIGPLAN Not.*, 31(5):127–136, 1996.

[3] ARM Limited. Cortex A8 technical reference manual. `http://infocenter.arm.com/help/index.jsp?topic=com.arm.doc.ddi0344/index.html`, 2006.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[5] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[6] M. Castro, M. Costa, J. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *2009 Symposium on Operating System Principles*, pages 45–58, October 2009.

[7] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software guards for system address spaces. In *OSDI '06: 7th Symposium on Operating Systems Design And Implementation*, pages 75–88, November 2006.

[8] B. Ford. VXA: A virtual architecture for durable compressed archives. In *USENIX File and Storage Technologies*, December 2005.

[9] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *2008 USENIX Annual Technical Conference*, June 2008.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2000.

[11] S. Graham, S. Lucco, and R. Wahbe. Adaptable binary programs. In *Proceedings of the 1995 USENIX Technical Conference*, 1995.

[12] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[13] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Masters Thesis, Computer Science Department, University of Illinois, 2003.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 1999.

[15] J. Liu and Y. Wu. Performance characterization of the 64-bit x86 architecture from compiler optimizations' perspective. In *Proceedings of the International Conference on Compiler Construction, CC'06*, 2006.

[16] S. Lucco, O. Sharp, and R. Wahbe. Omniware: A universal substrate for web programming. In *Fourth International World Wide Web Conference*, 1995.

[17] C.-K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, and R. Cohn. Profile-guided post-link stride prefetching. In *Proceedings of the ACM International Conference on Supercomputing, ICS'02*, 2002.

[18] S. McCamant and G. Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. Technical Report MIT-CSAIL-TR-2005-030, MIT Computer Science and Artificial Intelligence Laboratory, 2005.

[19] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium*, August 2006.

[20] G. Necula. Proof carrying code. In *Principles of Programming Languages*, 1997.

[21] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and J. Chen. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*, pages 49–64, July 2005.

[22] J. Richter. *CLR via C#, Second Edition*. Microsoft Press, 2006.

[23] C. Small. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, June 1997.

[24] C. Small and M. Seltzer. VINO: An integrated platform for operating systems and database research. Technical Report TR-30-94, Harvard University, Division of Engineering and Applied Sciences, Cambridge, MA, 1994.

[25] Sun Microsystems. Compressed OOPs in the HotSpot JVM. `http://wikis.sun.com/display/HotSpotInternals/CompressedOops`.

[26] M. Swift, M. Annamalai, B. Bershad, and H. Levy. Recovering device drivers. In *6th USENIX Symposium on Operating Systems Design and Implementation*, December 2004.

[27] U. S. Department of Defense, Computer Security Center. Trusted computer system evaluation criteria, December 1985.

[28] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[29] C. Waldspurger. Memory resource management in VMware ESX Server. In *5th Symposium on Operating Systems Design and Implementation*, December 2002.

[30] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009.