International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

SaBRe: Load-Time Selective Binary Rewriting

Paul-Antoine Arras* · **Anastasios Andronidis*** · **Luís Pina** · **Karolis Mituzas** · **Qianyi Shu** · **Daniel Grumberg** · **Cristian Cadar**

Received: date / Accepted: date

Abstract Binary rewriting consists in disassembling a program to modify its instructions. However, existing solutions suffer from shortcomings in terms of soundness and performance. We present *SaBRe*, a load-time system for selective binary rewriting. *SaBRe* rewrites specific constructs—particularly system calls and functions—when the program is loaded into memory, and intercepts them using plugins through a simple API. We also discuss the theoretical underpinnings of disassembling and rewriting. We developed two backends—for `x86_64` and `RISC-V`—which were used to implement three plugins: a fast system call tracer, a multi-version executor, and a fault injector. Our evaluation shows that *SaBRe* imposes little overhead, typically below 3%.

Keywords selective binary rewriting · `x86_64` · `RISC-V` · system call tracing · multi-version execution · fault injection

1 Introduction

Binary rewriting is widely used to implement security and reliability techniques, such as software fault isolation [68], sandboxing [67], multi-version execution [26], program optimization [3], and bounds checking [49].

The goal of binary rewriting is to add, delete and replace instructions in binary code. There are two main types of binary rewriting techniques: static and dynamic. In static binary rewriting, the binary file is rewritten on disk *before* the program executes, while in dynamic binary rewriting it is rewritten in memory *as* the program executes. With static approaches, the rewriting process does not incur any overhead during execution as it is performed before the program starts running. However, static binary rewriting is hard to get right: correctly identifying all the code in the program is ultimately reducible to the halting problem [24] in the presence of variable-length instructions and indirect jumps.

By contrast, dynamic binary rewriting modifies the code in memory, during program execution. This is typically accomplished by translating one basic block at a time and caching the results, with branch instructions modified to point to already translated code. Since translation is done at runtime, when the instructions are issued and the targets of indirect branches are already resolved, dynamic binary rewriting is not affected by the aforementioned issues for static binary rewriting. However, this style of translation is heavyweight and incurs a large runtime overhead.

In this paper, we present *SaBRe*, a system that implements a novel design point for binary rewriting. Unlike prior techniques, *SaBRe* operates at load-time, *after* the program is loaded into memory, but *before* it starts execution. Like static binary rewriting techniques, *SaBRe* rewrites the code in-place. However, *SaBRe*'s translation is done in memory, similarly to dynamic binary rewriting. This combination enables *SaBRe* to efficiently and safely rewrite all the code mapped into a process, including dynamically-loaded libraries, while increasing the program start time usually by less than 70ms. We note here that some techniques based on load-time binary rewriting exist—such as Xifer [13] in the context of software diversity—but they rely on specific features of the operating system's loader (see §6).

* equal contributions

P.-A. Arras, K. Mituzas, Q. Shu, work done while at Imperial College London · A. Andronidis, D. Grumberg, C. Cadar, Imperial College London · L. Pina, University of Illinois at Chicago, work done while at Imperial College London.

SaBRe is a *selective rewriter* that intercepts instructions of interest (e.g. system calls and function prologues) by relocating them and rewriting their original location with byte-long invalid instructions. *SaBRe* installs an invalid-instruction handler that identifies which instruction was intercepted by the value of the program counter at the invalid instruction. Using this scheme, *SaBRe* improves the state of the art over static techniques as it can retain the original semantics of the binary even in the presence of indirect jumps only known at runtime and jumps to the middle of existing instructions.

The goal of *SaBRe* is to provide a framework for implementing plugins for improving software reliability, such as tracers and fault injectors. Therefore, *SaBRe* is designed to take advantage of code generated by well-behaved compilers such as `gcc` and `LLVM`, even at high optimisation levels. Obfuscated and hand-crafted code, such as that found in malware, is out of scope.

A key optimization in *SaBRe* is to replace the relocated instructions with a direct jump to their handler code, based on the observation that code generated by modern compilers follows well-known patterns for certain types of constructs such as system calls and function prologues. This reduces *SaBRe*'s average runtime overhead from 13%-62% to only 0.3%-3% on our benchmarks.

We implemented two rewriting backends based on this design: one for `x86_64` and one for `RISC-V`. Both rewriters provide the same flexible API, which we used to implement three different *plugins*: a fast system call tracer, a multi-version execution system, and a fault injector.

In summary, our main contributions are:

1. A new design point for selective binary rewriting which translates code in memory in-place at load time, before the program starts execution.
2. An implementation of this approach for two architectures, `x86_64` and `RISC-V`. We make our implementation available as open source.
3. The implementation of a system call tracer based on *SaBRe*, which is *complete* (does not miss `vDSO` system calls) and *faster* (2x slowdown vs 28x average slowdown) when compared to the state of the art.
4. The implementation of a novel system-call fault injector, that finds bugs in existing libraries that mishandle error conditions from system calls (e.g. `glibc` mishandling an `mmap` that returns `NULL`).
5. A comprehensive evaluation using three plugins: the fast system call tracer and fault injector described above, and the reimplementation of a multi-version execution system.

At a high level, *SaBRe* starts by disassembling the binary into its constituent instructions and then selectively rewrites the desired instructions in memory, at load time. We first discuss the rewriting stage in §2 and then the disassembly stage in §3. We next present the implementation of *SaBRe* in §4, and extensively evaluate it in §5.

2 Rewriting

This section assumes that all instructions are accurately disassembled; we discuss this process in detail in §3. We start with a discussion of existing binary rewriting approaches (§2.1), present the standard interception in *SaBRe* via invalid instructions (§2.2) and then discuss a key optimization based on trampolines (§2.3).

2.1 Static vs. dynamic rewriting

Binary rewriting schemes are traditionally categorised as either static or dynamic [63]. The latter is performed at runtime and relies on actual execution of the program. It incurs a high overhead in both space and time, and therefore generally cannot be used in production (see also our experiments in §5.2). Moreover, it is useful for heavyweight instrumentation purposes but not particularly suited to the type of plugins based on selective rewriting.

On the other hand, static binary rewriting can be performed off-line on binary files, which makes it amenable to deployment. Nonetheless, statically rewriting a whole program is hard due to dynamic linking and position-independent code—which is now the default for most modern compiler targets—and the difficulty of creating a valid modified executable—which may require changing a large number of offsets. Therefore, we propose to statically rewrite programs *at load time*. This has several benefits:

1. Low overhead that only occurs on program startup—which makes it suitable for production.
2. Rewriting done after the program is loaded in memory, when effective addresses are known and dynamic linking and position-independent code are not of concern anymore.
3. Rewriting done in-memory, so there is no need to create a valid executable on disk.
4. Shared libraries can be rewritten transparently and selectively for each execution.

The state of the art in static rewriters can broadly be classified based upon the underlying technique used: insertion, trampoline or lifting [29].

Insertion [29]. The most straightforward rewriting technique is to insert instructions directly into the main execution stream. Unfortunately, this requires "stretching" the whole binary to accommodate added instructions. In other words, references have to be recomputed and program headers updated to reflect the shifted locations. This may be done extensively at link time to leverage relocation information and mapping symbols [57,59,53,14], or post-link-time [15]. Recent additional work [16,64] shows that in binaries that consist only of position independent code, this approach is also feasible.

Trampolines [27,32]. This approach inserts an unconditional jump to an out-of-line piece of code called a *trampoline*. Its role is to set up the environment in a way that enables a dedicated function, thereafter named *handler*, to be called transparently. However, to reach the trampoline, some control-transfer code has to be inserted at the target snippet's original location: a *detour*. Inserting a detour at an arbitrary location can be challenging, as we discuss in §2.3.

Lifting [46]. This approach translates binary code into a higher-level representation, rewrite it, and then reassemble it. The main advantage of binary lifting is to expose high-level constructs like function arguments and return values as well as symbols rather than memory locations. Unfortunately, these representations are usually tied to a specific compiler (e.g. LLVM bitcode), and translating and then recompiling code is a costly operation.

2.2 Standard interception in *SaBRe*

In general, rewriting includes three possible operations: adding, removing and replacing instructions. We discuss all these operations in terms of replacing an original instructions snippet O with another snippet R, where both O or R could have size 0. Let $A(o)$ and $S(o)$ be functions that map an object o into its start address and size, respectively. When $S(O) = 0$, $A(O)$ indicates the offset in the code where R should be added. When $S(R) = 0$, it means that O should be removed.

We have three main cases:

1. $S(O) = S(R)$: simply overwrite O with R
2. $S(O) > S(R)$: insert R and pad the remaining space with `nop`s
3. $S(O) < S(R)$: insert an illegal instruction and catch the SIGILL signal

In the last case, R cannot be inserted in place because it is larger than O. In *SaBRe*, the standard way to deal with this situation is to replace O with an illegal instruction (e.g. `UD0` for `x86_64`) and pad the remaining space with `nop`s. The exact illegal instruction picked is arbitrary but will be used at runtime to determine which R to insert. Then *SaBRe* installs a handler to catch the SIGILL signal triggered by the CPU attempting to decode this illegal instruction. This signal handler first checks whether the cause of the SIGILL is a legitimate illegal instruction, in which case it simply redirects execution to the default handler. Otherwise, the signal handler simply executes the R corresponding to the illegal instruction inserted, before returning to normal execution. In general, since there can be only one SIGILL handler per process and the number of binary sequences guaranteed by an ISA to encode illegal instructions[1]is limited, this approach does not scale. However, as *SaBRe* is a *selective* rewriter, it is only concerned with a limited number of instruction types; furthermore, the optimisation presented next removes most of these illegal instructions. Besides, in case this scheme had to be generic, the solution would be to associate each R not with a specific illegal instruction but rather with the location to be rewritten, i.e. the value of the program counter that raised the signal.

2.3 Trampoline-based optimisation

Interception via illegal instructions is expensive, as it involves a context switch to the kernel and back. A key optimisation in *SaBRe* is to replace as many illegal instructions as possible with direct jumps to the handling code. The optimization takes advantage of the types of instructions *SaBRe* is primarily designed to intercept, particularly system calls, functions and vDSO calls, for which compilers generate patterns that *SaBRe* can take advantage of.

The optimization replaces illegal instructions with detours via trampolines, which we briefly described in §2.1. The main challenge is to find space in the main instruction stream for the unconditional jump to the trampoline. In a CISC context (e.g. `x86_64`), a single instruction suffices to encode the jump. However, with RISC architectures (e.g. `RISC-V`), the restricted encoding space usually requires at least two instructions: one to load the target address into a register, plus one to actually perform the jump. Hence we will use the term *jump snippet* from now on.

When a detour is required, depending on the relative sizes of the original snippet O and the jump snippet J, some surrounding instructions may need to be moved to the trampoline. Thus the following two cases emerge:

[1] Excluding encodings reserved for future use or undefined.

1. $S(O) \geq S(J)$: insert J and (possibly) pad the remaining space with `nop`s
2. $S(O) < S(J)$: relocate as many neighbouring instructions as necessary to accommodate J

The trampoline then comprises up to six parts:

1. *Preamble* (optional): code from O that is to execute before the *handler* and had to be relocated
2. *Pre-processing*: ABI-dependent code to ensure transparency
3. Call to the *handler*
4. *Post-processing*: ABI-dependent code to restore the original state
5. *Postamble* (optional): code from O that is to execute after the *handler* and had to be relocated
6. Jump back to the main instruction stream

On Linux `x86_64` for instance, the pre-processing consists in adjusting the stack pointer in order to preserve the red zone and aligning it on a 16-byte boundary. The *red zone* is a 128-byte memory area located below the stack pointer, i.e. it is contiguous to the top of the stack without being part of it [40, p.18-19]. It is an optimisation mandated by the ABI to save stack-adjustment instructions since it can be used as a scratch space for temporary data. Regular function calls may clobber the red zone but system calls preserve it. Likewise, the alignment is required by the ABI, whenever a function is called, as an optimisation to enable compilers to emit vector instructions. Another important aspect is register preservation: *SaBRe* stores them on the stack just before calling the handler (see §4.3).

Not all instructions can be safely relocated. If any such instructions are encountered, *SaBRe* does not perform the trampoline-based optimization and falls back to the standard interception scheme, which guarantees the soundness of the approach. In particular, there are three main classes of instructions that pose challenges:

Instructions with side effects. In most ISAs, certain instructions exhibit some kind of dependency. For instance, condition codes set by an earlier instruction may be used to determine whether a later conditional branch should be taken. Interposing a snippet in-between may therefore cause the original condition code to be overwritten which might in turn result in erroneous branching. In `x86_64`, condition codes are stored in a special status register called RFLAGS; they can be pushed onto and popped from the stack via dedicated instructions [28, p.79-82]. In ARM's compressed instruction set (Thumb), there is no encoding space for per-instruction condition codes. Instead Thumb provides the `if-then` (`IT`) instruction, which allows to specify a condition that applies to the next four instructions [2]. Thus relocating part of an IT block would result in the condition being applied to the wrong instructions. In addition, some instructions cannot be inserted into an IT block.

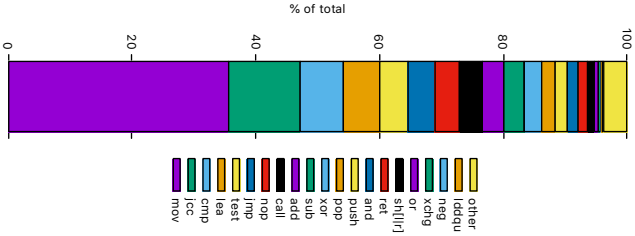


Fig. 1: Static instruction mix in `glibc` 2.28.

Table 1: Instruction white list for `x86_64`.

ADD	OR	AND	SUB	XOR	CMP
LEA	NOP	SHL	SHR	NOT	NEG
MUL	DIV	TEST	XCHG	MOV	

In general, whatever the ISA, most of the code only uses a handful of different operations. To illustrate, we carried out a frequency measurement of `x86_64` instructions in the `glibc` library. The results are plotted in Figure 1. The ten most frequent instructions represent 85 % of the mix. As few as twenty instructions cover almost 97 % of the code.

Therefore, we maintain a white list of instructions known to be safely-relocatable, which has several advantages. First, even though there will be a lot of false negatives, these should only represent a tiny fraction of the whole instruction mix and thus be very rare. Second, this approach allows us to be conservative by leaving out instructions whose safety is not clear. This eliminates false positives, i.e. potentially unsafe instructions will never be relocated. Third, it is much easier to implement a short white list rather than a long black list.

Drawing on the `glibc` instruction mix, Table 1 presents the white list for `x86_64`. This list covers 73 % of the `glibc` code and, in our experience, suffices to relocate all instructions around system calls, function and vDSO prologues.

Jump targets. Control transfers are of two types: direct or indirect. The latter is the least frequent: 20% in SPEC JVM98 and 9% in SPEC INT95 C [34] Direct control flow changes have the destination address statically encoded into the instruction, usually as a PC-relative displacement. In *SaBRe*, all direct jump destinations are recorded during disassembling and then checked against whenever a snippet is considered for relocation at rewriting time. However, there are situa-

tions (e.g. higher-order functions and virtual methods) where the actual target is not known at compile time. In this case, a level of indirection is required: instead of directly specifying the destination, the instruction has a register operand. The designated register holds the target address computed at runtime, which depends on the program's inputs. As a result, statically deriving the destination of indirect control transfers is reducible to the halting problem [24]. Similarly, it is not possible to determine whether an arbitrary memory location is the target of an indirect jump.

This poses an interesting theoretical challenge to our rewriting scheme as some relocated instructions may become unreachable. For instance, let us suppose that we relocate a snippet S comprising three instructions $I_{1..3}$, and replace them with a jump. If I_2 was the target of a control transfer, the relocation would make it unreachable. Instead, control would land in the middle of the jump instruction which would obviously result in an incorrect execution. However, in the context of system calls, functions and vDSO, we can make the following observations. First, system calls have well-defined prologue and epilogue: the former loads arguments according to the ABI-specific calling convention, while the latter tests the return value and stores it either as a result or an error code. Second, named functions and vDSO are detoured at the prologue rather than at call site. In both cases, we can rely on the fact that a control transfer does not end up in the middle of these well-defined sequences.

PC-relative addresses. To facilitate shared-library loading and address-space layout randomisation (ASLR), most ISAs support PC-relative addressing. As its name stands, it is a dedicated memory addressing mode that allows to designate a location as an offset from the current PC value. Obviously, if a PC-relative instruction is relocated to a trampoline, the displacement will become invalid.

For system calls, due to their standard prologue and epilogue, we have not encountered such instructions. However, since function detouring offers less flexibility in the choice of instructions to relocate, we enhanced *SaBRe* to handle PC-relative addressing. The new displacement is easily obtained, as follows. Let D and L denote the displacement and the instruction location respectively. Then the new displacement is $D_{\text{new}} = D_{\text{old}} - (L_{\text{new}} - L_{\text{old}})$.

3 Disassembly

The goal of the disassembling stage is to recover the program code from the bytes in the binary file. This is a challenging task, as the disassembler is not aware of the underlying semantics of the bytes read from the binary. See Appendix A for more details.

In the context of load-time binary rewriting, we need to statically disassemble the code. There are two main algorithms available:

1. **Linear sweep.** The simplest solution is to start from the first instruction and then sweep through the code until the end. The main advantage is obviously the ease of implementation and the low overhead. However, linear sweep suffers from a notable shortcoming: it does not attempt to make any distinction between actual code and possibly embedded data. Furthermore, it is oblivious to most kinds of instruction overlapping. In addition, the algorithm must be pointed to the start of a legitimate instruction, which is non-trivial for variable-length ISAs. Nonetheless, traditional linear sweep can be improved to help code discovery by leveraging relocation information [52] or pattern matching of well-known constructs.
2. **Recursive traversal.** Another option is to scan the code recursively by analysing and following its control flow. This translates into much more complex heuristics compared to linear sweep but enables skipping mixed-in data and exploring possibly hidden execution paths. Nevertheless, recursive traversal has its own shortcomings, the main one being its inability to reliably handle indirect control transfers [31]. In fact, indirect jumps and function calls are hard to analyse statically because the effective destination addresses are computed at runtime. The usual workaround is to use speculative disassembly, i.e. to sweep through unreachable areas of executable segments in case they might be indirectly targeted [18].

For load-time disassembly, the two criteria for adequate disassembly are coverage and performance. Linear sweep wins in terms of performance because it only requires a single decoding pass. In terms of coverage, even though code discovery is theoretically difficult, previous work has shown that well-behaved `x86_64` compilers never mix code and data [1]. More importantly, the same work has highlighted that only linear sweep consistently achieves 100% coverage.

We next discuss the sufficient conditions for accurate disassembly and their feasibility in practice.

Proposition 1 *Sufficient conditions for adequate coverage:*

1. *Within a segment, instructions are tessellated, i.e. there is neither gap nor overlap between them.*
2. *The start address and size of executable segments are known.*

Proposition 2 *Sufficient conditions for accurate disassembly:*

1. *adequate coverage*
2. *code is immutable*

The proofs for these propositions are included in the appendix. We next discuss the feasibility of each condition.

Tessellation. Without this assumption, it is impossible to rule out over-coverage; and under-coverage can only be avoided at the price of disassembling every single offset [4]. This precludes linear sweep on code that relies on techniques like instruction overlapping and data interleaving. However, prior work has shown that `gcc` and `clang` produce tessellated code by default for `x86_64` [1]. As a matter of fact, they never insert data into executable segments nor do they emit overlapping instructions precluding hand-written assembly (e.g. `glibc`). Even jump tables are stored with all other read-only data in a separate section. Indeed, the Executable and Linkable Format (ELF)[2] specification has different sections for code (`.text`) and data (`.bss`, `.data` and `.rodata`).[3] Thus, in `x86_64` and `RISC-V`, jump tables are placed in the `.rodata` section.

However, a notable exception is ARM, with two constructs that break the tessellation assumption: *literal pools* of constants embedded in the text segment and *in-function jump tables*. Thus, supporting ARM in *SaBRe* would be possible at the price of some additional constraints on the disassembler, for instance reading mapping symbols[4] or relocation information [52]—which is expensive and not fail-safe since they can be stripped—or listing and analysing instructions that may precede such constructs (e.g. `LDRLS`).

Start and size of code segment. In practice, the locations and sizes of all segments are required by the operating system's loader to map them into memory. This information is therefore provided as metadata in the final compiled and linked binary, whatever the executable file format. Besides, the *executability* of a segment is always decidable. Most modern CPUs support an `NX` bit in their page table to mark entries that hold non-executable data—by default, everything is executable. The operating system sets this bit, if necessary, when the memory page is mapped into the process space. This is also reflected in the executable file format. For instance in ELF, program headers list code segments with their start address and size, and with flag `X` (eXecutable).[5] Given tessellation, disassembling all `X`-flagged segments excludes under-coverage.

Immutability. Most applications do not modify their code during execution, and the operating system typically restricts write permissions on executable pages—a widespread scheme commonly known as *write XOR execute*. However, one legitimate instance of self-modifying code is just-in-time (JIT) compilation, which *SaBRe* does not support.

4 Implementation

In this section, we present the architecture of *SaBRe* and its practical usage.

SaBRe is built using a modular architecture that maximises flexibility, as illustrated by Figure 2. The *backbone* of *SaBRe* comprises the loader and the rewriter. In addition, *SaBRe* requires two modules, a *backend* and a *plugin*. The backend gathers all the ISA-specific code: the disassembler and the binary-code emitter. The plugin implements the exact purpose of the rewriting, e.g. tracing system calls, injecting faults, etc. Note that plugins are loosely coupled with the backbone by a well-defined API, which enables third-party plugins to be added to *SaBRe* easily.

The current implementation of *SaBRe* targets the Linux operating system. The backbone comprises 2108 LOC in C; the `x86_64` backend has 1333 LOC split between C (963) and ASM (410). The API and some optional support code (data structures to speed up the traversal of scanned libraries by the rewriter) account for an additional 2016 LOC in C. The resulting `x86_64` binary is only 47 KiB split between 33 KiB of code and 14 KiB of data. The binary for `RISC-V` is smaller: 26 KiB of code and 14 KiB of data, making up a total of 40 KiB. Therefore, *SaBRe* can be used in small embedded systems with stringent memory constraints.

SaBRe is available as open source at:
`https://github.com/srg-imperial/sabre`.

4.1 Backbone

SaBRe's *loader* is a dynamically linked executable whose only dependency is the C runtime library (`libc`). It provides the entry point to *SaBRe*'s execution and only runs at load time. Upon startup, its first task is to

[2] ELF is the default file format for executable and object files on UNIX derivatives.

[3] Mach-O and PECOFF have similar sections.

[4] The ARM ELF specification requires that *mapping* symbols should be emitted to identify inline transitions between code and data, notably at literal pool and jump table boundaries.

[5] In systems without an OS, where static loading is preferred, this information can be found in the linker script.

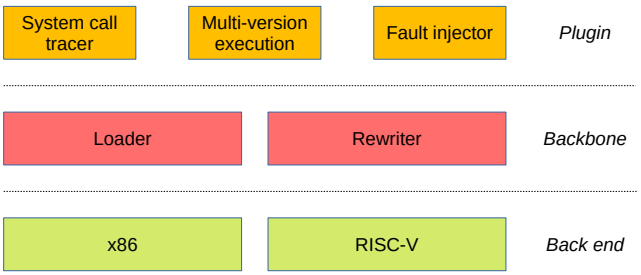


Fig. 2: *SaBRe* architecture.

load the plugin and perform its specific initialisation, as detailed in §4.3. Then, the loader scans the memory map of its own process, which includes only private and `libc` segments, to avoid rewriting them later. Next, it loads the user application (thereafter called *client*) into the same process and scans the memory map again to rewrite the executable segments of both the client binary and the dynamically linked libraries known to have system calls.[6] Lastly, the loader rewrites the stack for the client and jumps to its entry point.

If the client is statically linked, then the entry point is the binary itself. By the time the client receives control from the loader, all system calls have already been rewritten since libraries (including `libc`) are incorporated.[7] Otherwise, if the client is dynamically linked (which is the default on Linux), the entry point is the operating system's dynamic loader (`ld.so` on Linux). In this case, *SaBRe*'s loader rewrites `ld.so` to intercept library loading and rewrite the libraries before any system call has been issued. Thus, all libraries needed by the program (including third-party plugins dynamically loaded by the client using `dlopen`), are systematically and transparently rewritten at load time, ensuring that no system call can be missed. Besides, child processes are also instrumented through the interception of the `clone` system call (see Section 4.2).

SaBRe's *rewriter* is built into the loader's binary and is called just once at load time. This ensures the runtime overhead is only incurred by the trampoline and the handler. First, the rewriter handles the virtual dynamic shared object (vDSO). The vDSO is an optimised user-space implementation for some frequent and safe system calls (e.g. `gettimeofday`). Its main point is to avoid the overhead resulting from context switches to kernel space. Rewriting the vDSO segment is safe because each process has its own private copy, so it only entails the same precautions as rewriting other mappings. Second, the rewriter examines each library known to contain system calls or selected functions. For each such library, the rewriter searches the symbol table (if present) for the names of functions to be intercepted, and the corresponding prologues are detoured. Then, the rewriter scans the `.text` section for system calls using the ISA-specific disassembler (see §4.2). As all system calls are always rewritten, it is up to the plugin's handler to treat some of them specially, e.g. by filtering system call numbers.

In both cases, the patching mechanism works as follows. The rewriter scans the target memory range, first to collect branch-target addresses (see §2.3), then to actually rewrite. The rewriter keeps an in-memory buffer of the last few instructions disassembled so that they can quickly be relocated whenever a system call is encountered. The rewriter searches for candidates for relocation both backwards and forwards from the system call. If not enough space can be made to accommodate the detour (e.g. 5 bytes on `x86_64`), the target snippet is replaced by a short illegal instruction (e.g. `UD0` on `x86_64`). At runtime, if triggered, the illegal instruction results in a signal that is caught internally by *SaBRe* so that the execution is then redirected to the plugin's system call handler.

SaBRe is designed to detour selected functions named by the user. To this end, *SaBRe* relies on the symbol table to map a function name into the address of its first instruction. For this reason, if the client binary is stripped of its symbol table or when inlining is used, function interception is not possible, except for functions exported for dynamic linking.

4.2 Backends

For each intercepted instruction I, the disassembler provides *SaBRe* with the following information:

1. A pointer to the instruction following I, from which *SaBRe* computes I's size;
2. I's addressing mode, in particular whether it is PC-relative;
3. I's side effects (e.g. the flags that it sets);
4. I's opcode category: control flow, system call, etc.

SaBRe does not depend upon third-party libraries for disassembling, due to portability and efficiency concerns, but the built-in disassemblers are as sound as a standard one. Currently, *SaBRe* is able to disassemble two ISAs: `x86_64` and `RISC-V`. The backend is also in charge of all aspects that involve emitting assembly or binary code. This includes: (1) the jump to the trampoline, (2) the trampoline itself, and (3) the pre-handlers.

[6] We perform this optimisation because most applications perform system calls via a handful of system libraries, namely: `libc`, `librt`, `libpthread` and `libresolv`.

[7] In the current implementation, libraries `dlopen`-ed by statically-linked binaries are not rewritten. However, this situation is extremely rare.

Pre-handlers are hand-written assembly functions that are called from the trampoline. They perform the following operations in turn: save machine state (registers and flags), call the plugin-specific handler, and restore machine state when the plugin returns.

SaBRe also handles two system calls in a special way: `clone` and `rt_sigreturn`. Intercepting `clone` poses a special challenge for *SaBRe* as after the system call is issued by the kernel, the child thread is created with a fresh stack and no information on how *SaBRe* can return execution back to the caller of the `clone` system call (usually the libc library). *SaBRe* solves this issue by carrying the return address to the trampoline that issued the system call. For all other system calls, this is unnecessary as all information of the return addresses are stored in the stack. So when a `clone` system call is detected, *SaBRe* jumps to the explicit return address of our trampoline, which in turn jumps back to the hard-coded address of the rewritten system call.

The second system call that *SaBRe* needs to handle in a special way is `rt_sigreturn`. This system call returns execution to an application provided pointer which was given to the kernel from `sigaction` in order to return from a signal handler. `rt_sigreturn` never returns and when it is issued execution is directly resumed by the kernel to the application provided pointer. This creates issues as *SaBRe* has tampered with the stack while calling the trampolines and handlers. Thus *SaBRe* restores back the stack pointer to the value it had prior the jump to the trampoline, just before `rt_sigreturn` is issued.

4.2.1 x86_64 backend.

Because it has variable-length instructions, the `x86_64` ISA is relatively compact. Detours in `x86_64` require 5 bytes for an unconditional jump: 1 for the opcode, 4 for a 32-bit displacement that covers a ±2 GiB range in address space. Our empirical evidence shows that this is sufficient to reach the trampoline.

Due to the complexity of the ISA, resorting to a full-featured disassembler would be costly. However, as we just need to be able to quickly traverse the code until an instruction of interest is found, our implementation does not need to fully disassemble the instructions being skipped.

4.2.2 RISC-V backend.

Reduced instruction sets may require several consecutive instructions to perform a jump. In `RISC-V`, there are two possibilities. First, if the trampoline is within a 20-bit displacement (i.e. 1MiB range), then a single 4-byte instruction suffices. Otherwise, two 4-byte instructions can reach the required ±2 GiB address range.

In the theoretical case where the trampoline cannot be fitted within the required address range, *SaBRe* aborts. We never encountered this case in practice.

4.3 Plugins

Plugins are compiled as shared objects that are dynamically loaded at runtime via `dlopen`. *SaBRe* provides a well-defined application programming interface (API) to orchestrate the interaction with the backbone. This API, designed with both flexibility and simplicity in mind, only requires two functions to be implemented: the *initialisation*, called once at load time; and the *system call handler*, called at runtime whenever a system call is intercepted. The plugin can also implement three additional, optional types of functions. First, the plugin can provide special handlers for vDSO calls (up to four on `x86_64`). By default, *SaBRe* treats such calls as regular system calls, thus redirecting them to the system call handler. Second, the plugin can intercept selected functions, by associating handlers to function-library or function-binary couples. In other words, the user can decide to intercept a function belonging to either a library (in case of dynamic linking) or the binary itself by filling a dedicated data structure with the name of the function and the name of the library or binary. Finally, the plugin can provide a post-initialisation function that *SaBRe* calls *after* the client is loaded (as opposed to the initialisation function called *before* the client is loaded). This may be used, for instance, to have different system call handlers between load time and runtime.

Besides bootstrapping the plugin, the initialisation function must perform the following two tasks: (1) process the plugin's command-line arguments, and (2) register the system-call and function handlers with *SaBRe*. At interception time, the handler receives the system call number and the arguments. It can therefore decide to actually issue the system call surrounded by some pre-/post-processing, for instance, or even not issue the system call at all.

4.4 Limitations

The *SaBRe* tool has a number of limitations; we enumerate the most important ones here. (1) As discussed in the introduction, *SaBRe* is not designed to handle self-modifying code. (2) By default, only system

calls, vDSO and function prologues are supported, although *SaBRe* may easily be extended, as we did for the multi-version execution plugin (see §5.3). (3) *SaBRe* relies on the symbol table to map a function name into the address of its first instruction, so if the client binary is stripped of its symbol table or when inlining is used, function interception is not possible. (4) Libraries loaded at runtime (via `dlopen`) by statically-linked binaries are not seen by the rewriter. (5) In the case of dynamic linking, stack unwinding information is lost due to the simplified loading scheme of the client; this will be fixed in a future version, at least for position independent executables (PIE), by using `dlmopen()`.

Also, due to the cohabitation of two executables in the same process space, some usually trivial operations may be unsafe. The main point of contention here is libc's memory allocation functions (`malloc()`, `free()`, etc.). To maintain isolation and ensure the client's behaviour is preserved, *SaBRe*'s loader has its own copy of libc. However, because of the heap and the thread-local storage being shared with the client's libc, combined with the frequent runtime switches between the client and the plugin, the latter cannot rely on libc's `malloc`. The technical issue is that the `FS register` cannot be shared between the client and the plugin or else memory operations will overlap, which will lead to memory corruption. Therefore, in the current implementation, it is recommended that the plugin should either not make any calls to `malloc` functions (both directly and indirectly) or provide its own implementation. We currently have a functioning workaround that switches the `FS register` on every jump between the client and the plugin, but it introduces three extra system calls as switching the `FS register` requires communication with the kernel. We are working on a much more performant alternative that will be implemented in the next version of *SaBRe*.

5 Evaluation

We first present a set of experiments that demonstrate the efficiency of *SaBRe* in terms of load-time and interception overhead (§5.1), and then show its versatility by illustrating how it can be used to build various types of plugins (§5.2–5.4).

5.1 Overhead

To benchmark *SaBRe*'s load-time and interception overhead, we implemented an *identity plugin* that intercepts all system calls and vDSO, and simply reissues them in the handler. The *identity plugin* was tested both without and with the trampoline-based optimisation.

Experimental setup. We tested the identity plugin with four widely-used, high-performance network servers.

Nginx [44] is a popular reverse-proxy server, often used as an HTTP web server, load balancer, or cache. ***Lighttpd*** *[35]* is a lightweight web server optimised for high-performance environments. We benchmarked Nginx 1.16.1 and Lighttpd 1.4.54 with *wrk* 4.1.0, a modern HTTP benchmarking tool [65]. Both Nginx and Lighttpd servers are configured to serve a 2KiB file containing random data, with protocol-level compression enabled. *wrk* is transferring this file for 3 minutes using one thread and 40 open connections. We also use a warm-up period of 5s.

Redis [50] and ***Memcached*** [41] are high-performance, in-memory key-value data stores, used by many well-known services. We benchmarked Redis 5.0.7 and Memcached 1.5.20 with *memtier* 1.2.17, a Redis/Memcached benchmarking tool [42]. Both servers are started with an empty store. *memtier* issues the same number of GET and SET operations with values of 100 bytes, for 3 minutes using 3 threads and 30 open connections. We also introduced a warm-up period of 5s.

We compiled all servers with default compiler optimisation options, i.e. `-O2`. All experiments were conducted on a machine equipped with two 2.50 GHz Intel Xeon E5-2450 v2 CPUs (8 physical cores, 16 logical cores per CPU), with 188 GiB of RAM, and running 64-bit Ubuntu 18.04.3 (kernel version 4.18.0-21-generic, `glibc` version 2.27-3ubuntu1).

Rewriting statistics. We first report some statistics about the rewriting process on `x86_64`. In `glibc` version 2.28, *SaBRe* is able to rewrite all 404 `SYSCALL` instructions by means of a detour. The size of the trampoline (without preamble and postamble) is 5 instructions for functions and 10 for system calls. The only instance where a `UD` instruction and a signal are necessary is for `RDTSC` (see §5.3). This happens in the run-time initialisation of statically-linked binaries because both the preamble and the postamble comprise PC-relative instructions.

Load-time overhead. To benchmark *SaBRe*'s load-time overhead, we introduced an `exit(0)` at the top of the `main` function of the benchmarked applications, and then used `perf stat` with `chrt -f 99` and 2000 iterations to measure execution times. Results in Table 2 show that *SaBRe* with trampolines introduces a delay of less than 65ms during the loading phase of an application, but the absolute time is still negligi-

Table 2: Load-time overhead.

	Load time (ms)			Number of libraries	Binary size (MiB)
	Native	SIGILL	Trampoline		
Lighttpd	0.6	27.3	64.4	8	2.6
Nginx	0.6	25.7	62.0	8	4.6
Memcached	0.4	27.1	63.8	5	0.9
Redis	0.8	27.2	64.7	7	8.4

ble even for short-running interactive applications, as response times under 100ms are usually imperceptible to users [9]. *SaBRe* under the SIGILL mode performed better as it is much simpler to simply replace instructions with interrupts rather introducing trampolines. *SaBRe* shows no significant performance difference between different binary sizes and number of libraries, as scanning the assembly code in memory for targets is very efficient. The performance is mainly dominated by all the other technical details *SaBRe* needs to handle (see §4).

Interception overhead. Saturating the application's CPU utilization is important to identify the highest possible interception overhead. Table 3 reports worst-case overheads for native applications and the identity plugin under both SIGILL and trampoline modes. All tests were conducted at 100% utilization for the servers for a 3-minute execution, and the table reports the time and overhead per request. As can be seen, the runtime overhead when solely using SIGILL interception is high, varying from 13.20% for Nginx to 65.28% for Redis. When the trampoline optimisation is used, the overhead decreases substantially, at under 2.8% in all cases, a two orders of magnitude improvement.

Table 4 shows various statistics for the 3-minute benchmark execution when using trampolines. We report the time spent in the kernel, as applications with little kernel time should see little *SaBRe* overhead. Indeed, the time spent in the kernel roughly correlated with the overhead of *SaBRe*. We also report the total number of requests as well as the number of syscalls, vDSO call and RDTSC instructions. As expected, the total overhead roughly correlates with the sum of system calls and vDSO calls (the number of RDTSC instructions is insignificant in all cases).

We also explore the overhead at lower CPU utilisation levels. We do this only for Memcached and Redis, as the *memtier* benchmarking tool allows us to control the utilisation level (unlike *wrk* that we use for Lighttpd and Nginx). The results are presented in Table 5 and show that the overhead decreases substantially at utilisation levels of less than 100%.

We next compare with two open-source systems that provide functionality similar to *SaBRe*: syscall_intercept and LiteInst.

Comparison with syscall_intercept. The library syscall_intercept[8] is a run-time system call intercepting library that provides a low-level interface for hooking Linux system calls in user space. This is achieved by hotpatching the machine code of the standard C library in the memory of a process. We compare *SaBRe* to syscall_intercept due to the similar goals of the two projects.

syscall_intercept comes with some important limitations. First, syscall_intercept does not support the `clone` syscall and thus multithreaded applications are out of scope. Indeed we observed instant crashes in all of our server benchmarks except Lighttpd. Second, syscall_intercept only rewrites syscalls inside the `libc` library and thus other syscalls might be ignored. Lighttpd depends on libraries that have additional system calls, for example `librt`. Syscalls in libraries outside of `libc` are common: In a quick scan on our Ubuntu system we found that 20% of system libraries issue syscalls.

In our Lighttpd benchmark we avoid triggering system calls outside of `libc` in order to compare the two systems fairly. syscall_intercept showed a +7% overhead compared to +0.9% of *SaBRe*. In terms of load time, syscall_intercept showed a 230.35ms overhead compared to 64.4ms for *SaBRe*.

Inspection of the syscall_intercept code base revealed some contributing factors to the observed performance overhead. The trampolines leading to the user-provided interception routine perform additional unnecessary work compared to *SaBRe*, such as saving and restoring registers used by SIMD instructions (which we think would be better done as part of the user-provided handler when needed), as well as doing unavoidable additional checks related to syscall_intercept's implementation of logging. Furthermore, *SaBRe* employs a more optimised trampoline system using only two jumps to get to the user-provided system call handler as opposed to syscall_intercept which uses three or four jumps to get to the user-provided system call handler.

[8] https://github.com/pmem/syscall_intercept

Table 3: Overheads of trampolines vs SIGILL for a 3-minute benchmark execution using the identity plugin.

	Native	SIGILL		Trampoline	
	Time (ms/req)	Time (ms/req)	Overhead	Time (ms/req)	Overhead
Lighttpd	1.14	1.85	62.3%	1.15	0.9%
Nginx	3.94	4.46	13.2%	3.95	0.3%
Memcached	0.89	1.38	55.0%	0.90	1.0%
Redis	0.72	1.19	65.3%	0.74	2.8%

Table 4: Various statistics for a 3-minute benchmark execution with trampolines.

	Kernel time	Total requests	Total syscalls	Total vDSO	Total RDTSC
Lighttpd	69.3%	6.2M	75.4M	187k	0
Nginx	21.7%	1.8M	11.1M	91k	1
Memcached	79.2%	18.0M	37.8M	541k	6
Redis	74.2%	21.6M	45.3M	45.3M	3

Table 5: Performance difference per CPU utilisation (trampolines, identity plugin).

	CPU utilisation	Throughput (reqs/s)	Normal execution (ms/req)	Identity plugin (ms/req)	Runtime overhead
Memcached	100%	102,503	0.89	0.90	1.0%
	99%	99,394	0.58	0.59	0.5%
	70%	51,625	0.57	0.57	0.4%
Redis	100%	120,107	0.72	0.74	2.8%
	99%	110,324	0.54	0.55	1.3%
	85%	47,901	0.52	0.52	0.3%

We also note that syscall_intercept is initialized during early runtime while *SaBRe* is initialized during load-time and thus can intercept a larger set of functionality e.g. the loader itself and libraries that come before syscall_intercept in `.init` and `.preinit`.

Comparison with LiteInst. LiteInst [11] is an instruction-punning framework for x86-64. LiteInst also suffers from some important limitations. First, it only provides function-call interception and not syscall interception, even though this is likely a limitation of the implementation. Second, like syscall_intercept, it does not scan for the target function calls outside of the target binary, i.e. functionality inside external libraries is not intercepted. And third, LiteInst only provides passive probes that cannot skip over the intercepted syscalls and functions as in the case of *SaBRe*.

Given these limitations, we couldn't use our syscall interception experiments involving network servers. Instead, we used a micro-benchmark that prints messages through calling a local function 1000 times for a single run. We measured the average performance overhead of intercepting these function calls, using 1000 repetitions. This synthetic micro-benchmark simulates a worst-case scenario for both tools, as interceptions are very often and execution time is dominated by the loader due to the short life of the execution. In absolute numbers, the native version has a load-time of 0.2ms with a total execution time of 0.3ms.

LiteInst's average total execution time was 7.7ms with 7.5ms average load-time and 0.2ms of average intercepting time for the local function call. *SaBRe* showed an average total execution time of 52.4ms with 52.2ms average load-time and 0.2ms of average intercepting time. As expected both tools are significantly slower than the native version. *SaBRe* has similar performance characteristics for intercepting function calls, while the larger load-time (6.9x slower) is expected due to the fact that *SaBRe* scans both the binary and its libraries, and rewrites all syscalls even if the user does not provide any syscall handlers.

5.2 System call tracer

System calls are the main interface between user and kernel spaces. For that reason, when it comes to understanding how a given application interacts with the kernel, the ability to monitor the system calls it issues is paramount. On Linux, the most prominent tool to achieve this is `strace`.[9] Although widely used, it suffers from several shortcomings, all related to its underlying technique: the `ptrace` system call. First, it requires the existence of a separate process: the tracer. Second, in

[9] https://strace.io/

terms of performance, for each system call issued by the traced process, `ptrace` raises two signals and triggers four context switches. This is necessary to allow the tracer to get access to both the arguments and the return value of each system call. In addition, the tracer needs to attach to and detach from the traced process, but this only happens once and is therefore usually negligible in terms of overhead. Third, in terms of coverage, system calls that go through vDSO—thus bypassing the kernel—cannot be intercepted.[10]

Experimental setup. In order to improve on these issues, we introduce `sbrtrace`, a system call tracer that leverages load-time binary rewriting. We implemented `sbrtrace` as a *SaBRe* plugin that mimics `strace`'s output. We measured their respective overheads with both the `x86_64` and `RISC-V` backends, averaged over 10 runs. To compare against dynamic binary instrumentation frameworks, we also implemented an equivalent tool using Pin [38]. Pin does not support `RISC-V`, so we only compared against it on the `x86_64` benchmarks. We also considered comparing against DynamoRIO, but the available open-source DynamoRIO is tied to a very old GLibc version on Linux systems and we could not run it under our Ubuntu distribution.

Benchmarks for `x86_64` were run natively under Linux 5.4.0 on a machine powered by an Intel Core i3-8100 CPU (4 cores running at 3.6 GHz) and 32 GiB of DDR4 RAM running at 3200 MHz. Due to lack of hardware, `RISC-V` benchmarks were run on a QEMU instance emulating a single-core machine with 2 GiB of DRAM propping up Linux 5.4.0.

The benchmark we used issues `read` and `write` system calls as fast as possible using GNU `dd`. All the values provided hereafter are means over ten measurements.

Figure 3a shows the performance results for `x86_64`. In this experiment, slightly more than one million system calls were issued, equally split between `read`s and `write`s. The native execution spends 10.04 s in system calls and 14.40 s in user space, for a total of 24.45 s of wall-clock time on average.

The slowest tracer was the one based on Pin. It took 49x longer to run than the native execution. This overhead was largely due to Pin's rewriting and lookup of translated basic blocks which incurs a significant run-time penalty. It also introduces many more system calls which are required to allocate the additional executable pages that are necessary to host the rewritten code. Note however that the Pin tracer runs for over 20 minutes, so the benchmark is sufficiently long-running to warm up its internal code trace caches and thus does not unduly penalise it compared to the other approaches.

Due to the overhead introduced by `ptrace`, `strace` spends 57x longer in the kernel, but only 17x longer in user space. Overall `strace` introduces a 28x slowdown. By contrast, `sbrtrace` introduces only a 2x slowdown with 3x longer in the kernel and 1.3x longer in user space. The cost of intercepting system calls and formatting the output is responsible for the additional time compared to native execution. The formatting cost is exactly one `write system call` per original system call.

Since release 5.3, `strace` can arrange to be notified by `ptrace` only when specific system calls are issued using a `seccomp` sandbox [10]. This drastically reduces the overhead incurred by strace if it is used to trace only rarely occurring system calls. For example, we ran the `dd` benchmark, using `strace` with `seccomp`, but only tracing the `brk` system call which is only called three times. The results are now vastly different, `strace` with `seccomp` runs in 25.60 s split between 13.48 s in user space and 12.11 s in system calls, which is very similar to the native execution and significantly faster then `strace` which takes 469.88 s to complete the same benchmark. Similarly, when `sbrtrace` is configured to only trace the same `brk` system calls it is able to complete the benchmark in 25.00 s, 15.37 s of which are spent in user space and the remaining 9.62 s are spend in system calls.

For `RISC-V`, the experiment involved a smaller workload—10 KiB instead of 25 MiB—because the emulation imposes a significant run-time overhead. In this case, the total number of system calls issued was slightly above 20 k. Still, the results are very similar to `x86_64`, as depicted by Figure 3b. `strace` and `sbrtrace` are respectively 70x and 5x slower than native.

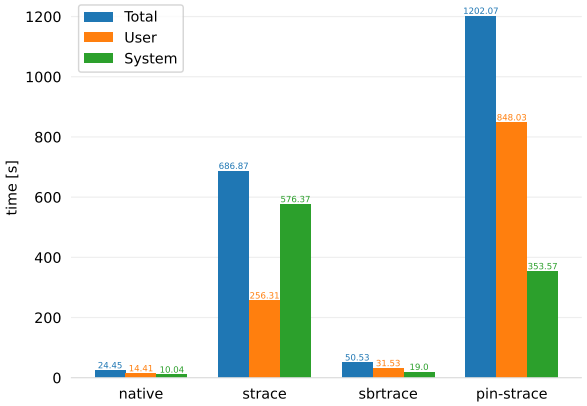
Finally, we remind the reader that `sbrtrace` also intercepts the vDSO calls, which is out of reach for `strace`.

5.3 Multi-Version Execution

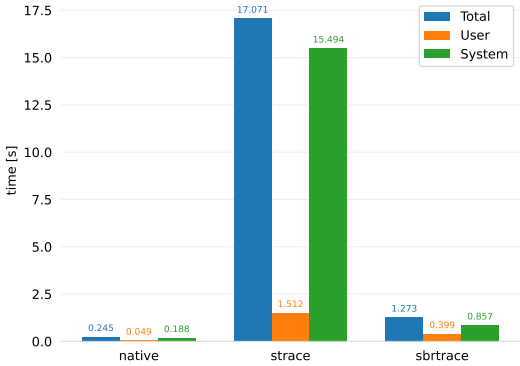
Multi-version execution (MVX) is a paradigm in which multiple versions of a program are run concurrently, with their execution synchronised and virtualised to appear as a single entity to the outside world. The technique has numerous applications in improving the security and reliability of software systems by combining diversity and fault-tolerance [51, 25, 66, 26].

Varan [26] proposed a novel decentralised architecture for MVX systems that leverages a record-replay

[10] vDSO can be disabled but, depending on the application, this may incur a significant overhead.



(a) N=25MB on `x86_64`.



(b) N=10KB on `RISC-V`.

Fig. 3: Execution times of `dd if=/dev/zero of=/dev/null bs=1 count=N`.

strategy based on an in-memory ring buffer. Several other recent MVX systems [47,30,66,58] incorporate elements of this architecture. In Varan, a *leader* version performs the actual system calls, vDSO and `RDTSC` instructions, and writes their results into an in-memory buffer, while multiple follower versions do not perform the calls and `RDTSC` instructions and instead read their results from the buffer.

We reimplemented Varan on top of *SaBRe* by splitting it in half: the part that intercepts system calls, vDSO and `RDTSC` instructions is built on top of *SaBRe*, while the actual MVX functionality of Varan resides in a separate plugin.

Results. First, the separation of the monolithic Varan into two resulted in an immediate increase in code quality. This separation of concerns meant that the Varan plugin did not have to worry about interception and binary rewriting, and also meant that it could now use third-party libraries such as `libc`, which was not previously possible before due to the engineering implications of its monolithic architecture. One particular area of improvement was support for multi-threading. While the original Varan supported multi-threaded applications, the support for this degraded over time, to a point that Varan could not properly support multithreading anymore. Separating it made it easier to debug and maintain this feature, and multi-threading works again well with the new *SaBRe*-based implementation.

To keep leader and follower executions consistent, we had to augment *SaBRe* to intercept `RDTSC` instructions. The `RDTSC` instruction counts the number of cycles since last reset. We also had to intercept the `__libc_start_main` function to allow proper initialisation of the plugin.

In terms of performance, the differences between the monolithic and *SaBRe*-based Varan are insignificant.

Table 6: SaBRe MVX overhead results (in ms), compared to Varan.

	Only leader			Leader & Follower		
Application	SaBRe	Varan	Overhead	SaBRe	Varan	Overhead
Lighttpd	1.78	1.79	-0.56%	1.92	1.93	-0.52%
Nginx	4.53	4.54	-0.22%	4.65	4.63	0.43%
Redis	0.91	0.91	0.32%	1.31	1.28	0.69%

Table 6 compares the performance of the monolithic and *SaBRe*-based Varan using the same benchmarks as in Section 5.1 (from which we removed Memcached, due to the multi-threading issues of the monolithic Varan discussed above). We show two scenarios commonly used in multi-version execution, one in which only the leader is run (e.g., as sometimes used in Mvedsua [48]) and one in which the leader and one follower are run (as sometimes used in FreeDA [47]). In both scenarios, the performance differences are insignificant, in the range of -0.56%–0.69%.

5.4 Fault injector

We have built a simple fault injector plugin that aims to test application resiliency in face of system-related error conditions. Error-handling paths for conditions related to starvation of resources in the system are not thoroughly tested. Therefore, we aim to check how well the application responds to conditions such as running out of memory or disk space. *SaBRe* lends itself perfectly to the task, as it allows us to intercept and modify system calls in an unobtrusive manner. Furthermore, this was achieved with minimal programming effort: the plugin is roughly 300 LOC long.

We employed a simple, configurable and extensible scheme for simulating system call failures. Each system call is assigned to families relevant to its functionality. The categories are device management, file-descriptor handling, network operations, process management, and memory management. These allowed us to categorise 286 system calls out of 333. The remaining system calls were either ones that are never allowed to fail and were left untouched (22 of them), or those that could not be neatly categorised and were thus put into an uncategorised family (25 of them).

Users can configure the fault injector by assigning failure probabilities to each family on the command line. These probabilities are used by the system call handler when it conducts a fault-injection campaign to determine whether or not the current system call should fail. In the event of a failure, an appropriate error code is returned by the handler, and the failure is logged to the relevant output stream. Beyond logging the failure, the occurrence number of the failing system call is recorded to aid with debugging.

Experimental setup. To get a broad overview of the effectiveness of the plugin, we applied it to the GNU Coreutils [20], containing well-known utilities such as `ls` and `mkdir`. Although we were able to use the test suite bundled directly with GNU Coreutils, it relies heavily on `gdb` scripts and function interception via the dynamic linker, all of which complicate the execution of the utilities and made the results harder to analyze. This is why we decided to drive the testing of the GNU Coreutils with the test suite of Busybox [8], an implementation of these utilities targeting embedded devices. Busybox's test suite relies only on the utilities themselves and is thus easier to execute. We ran the test suite with failures enabled for each family in isolation with a 20% failure probability, and then we added a final run with failures enabled for all families at a rate of 10%.

Results. We found three distinct types of issues, ranging from mild (cryptic error reporting messages) to severe (crashes):

a) Cryptic error messages: In general, error reporting varied significantly in quality. Many messages were cryptic, especially for a general user (e.g. ": Bad file descriptor"), and in some cases both cryptic and misleading, e.g. some applications claimed to be unable to write output, when the failure logs indicated that only reads had failed.

b) Lack of resiliency: Many system calls cannot be interrupted whilst they are handled. If this happens, `EINTR` is reported, and the client application is expected to retry the system call. However, we often found that many of the tested utilities immediately exited in this situation and merely printed the integer value of `EINTR` to standard error, instead of retrying the system call.

c) Crashes: In the face of unusual memory management conditions, many of the tested utilities immediately crashed with a segmentation fault instead of exiting gracefully. Notably, we found that most applications could not survive a failure in `mmap()` when mapping in libc pages during dynamic loading, which we think is acceptable. However, some applications crashed when the first call to `malloc()` failed. Specifically, when `malloc()`

is first called, the size of the uninitialised data segment is expanded using the `brk()` system call. This is done to provide malloc with an arena from which to allocate memory to the user: the heap. When this particular invocation of `brk()` fails, many application failed to detect this and crashed immediately.

6 Related Work

In 2009, Google developed the secure computing mode (seccomp) sandbox [21], as part of its Native Client (NaCl) project [67]. To the best of our knowledge, seccomp sandbox is currently the only pre-existing load-time selective binary rewriting implementation, although the idea was first proposed back in 1997 [19]. We implemented *SaBRe* by borrowing some code from NaCl, especially the x86_64 disassembler and some parts of the system call rewriter. This was considerably reworked to meet our needs and directly incorporated into the *SaBRe* codebase. However, *SaBRe* differs significantly from seccomp sandbox in several respects. First and foremost, seccomp sandbox has no flexibility with regard to how system calls are handled, and does not provide an API. Second, the isolation required by sandboxing introduces a significant runtime penalty. Third, seccomp sandbox specifically targets x86_64 and has no modularity. More broadly, *SaBRe* is a rewriting system suitable for various applications, while seccomp sandbox is solely aimed at sandboxing.

The following paragraphs offer an overview of projects that relate in some way to *SaBRe*. For a broader and more comprehensive review of the state of the art in binary rewriting, the reader is referred to Wenzl et al. [63].

As regards static binary rewriting, i.e. modifying a binary file ahead of execution, most techniques fall into one of three categories: trampoline, insertion or lifting (see §2.1). *SaBRe* belongs to the former, together with Detours [27], Dyninst (static) [5,12], BIRD [43], PEBIL [32], STIR [62], Multiverse [4] and E9Patch [17]. Detours only works at function level and thus cannot intercept individual instructions. BIRD partly relies on dynamic speculative disassembly, which incurs a significant runtime overhead. Both of them are specifically aimed at Windows binaries. PEBIL duplicates entire functions, even if just a couple of instructions are instrumented. STIR goes further and replicates whole segments. Multiverse leverages an expensive brute-force approach to disassemble every byte offset in the text section. Most importantly, none of them can rewrite the shared libraries needed by the program without additional effort from the user. Since all system calls usually lie in such libraries (particularly `libc`), this makes all of those tools unfit to the task. By contrast, Dyninst is able to rewrite libraries transparently but it suffers from the problems of static approaches, as discussed in §A.2. In fact, the programmer's guide states this limitation, alerting the developer that Dyninst finds targets of indirect jumps only by matching binary code against known patterns generated by popular compilers [12]. *SaBRe*, which works after the code is loaded and all addresses are resolved, can detect problematic indirect control-flow transfers by padding detours with invalid instructions. Diablo [57] works at link time but requires a patched toolchain and does not support dynamically-linked binaries, which greatly limits its usability on modern Linux systems. E9Patch improves on instruction punning (see below) by introducing instruction eviction and physical page grouping, all three techniques being agnostic to control flow. LLBT [54] is a static binary translation tool for ARM that lifts code into LLVM intermediate representation before retargeting a different ISA. As mentioned earlier, this is inefficient for binaries produced by other compilers. Additional recent work [56,60,61,16,64] has seen the emergence of compiler-agnostic variations on binary lifting. But these techniques rely on additional assumptions (typically the availability of relocation data) or are restricted to some binary forms (e.g. 64-bit PIC [16]). Furthermore, the static analysis involved is still heavyweight and it is not clear how efficiently a large number of libraries can be handled by such tools.

As regards dynamic binary rewriting, i.e. modifying instructions at runtime, the most prominent tools include: SecondWrite [18], Dyninst (dynamic) [7], DynamoRIO [6], Pin [38], MAMBO [22], ADORE [37] and LiteInst [11]. SecondWrite performs rewriting on LLVM bitcode, which is costly to do dynamically due to the binary lifting process. Dyninst relies on a separate process and `ptrace`[11] to achieve this at runtime, which incurs a high overhead in both space and time. DynamoRIO and Pin have been reported to impose runtime penalties of at least 20% and 54% respectively [39], just to maintain supervised execution, without any additional instrumentation, while *SaBRe*'s overhead on the identity plugin is less than 3% (see §5.1). Most dynamic rewriters have to pay similar costs, due to their common underlying technique: code caching. Thus, instead of being rewritten in place, the code is copied to a separate scratch space—the cache—where all indirect control flows have to be fixed up. Obviously, this also has a significant impact on memory and energy consumption. MAMBO, which specifically targets ARM, has a

[11] See §5.2 for why `ptrace` is costly. Note that unlike `strace`, Dyninst only relies on `ptrace` for rewriting, not for interception.

built-in mechanism to intercept system calls; but, although it has lower overhead than other dynamic binary modification tools, it still suffers from the cost of maintaining a code cache. *SaBRe* boasts a much lighter footprint on resources than all aforementioned techniques, which makes it the only practical option for binary rewriting on small embedded systems. The only two exceptions here are ADORE and LiteInst. Rather than code caching, the ADORE system leverages trampolines to redirect the execution of small portions of code at runtime, with very low overhead. However, it is restricted to a very specific application—runtime data cache prefetching—and is optimised accordingly. Similarly to *SaBRe*, LiteInst relies on trampolines, but it pairs this common technique with instruction punning, which allows it to work also with indirect jumps and short instructions We have directly compared with LiteInst in Section 5.1.

Besides, although all the aforementioned projects operate in user space, extended Berkeley Packet Filtering (eBPF) can be used to intercept system calls in the kernel. However, as eBPF programs are actually running inside the kernel, there are security implications. Therefore, eBPF programs have to be statically analysable for correctness by the eBPF verifier, which means expressiveness is limited due to the language not being Turing-complete. In particular, loops are forbidden, pointer arithmetic is restricted, accessible memory has fixed size and the instruction count is bounded.[12] In addition, as vDSO calls do not go through the kernel, they cannot be captured by eBPF. `vltrace`[13] is a system call tracer that relies on eBPF. It has several system requirements and dependencies, including a recent, suitably configured kernel and third-party libraries. In contrast, *SaBRe* is self-contained and works with any kernel. Besides, the Linux `perf-trace`[14] tool leverages kernel probes to monitor system calls. But it similarly requires a suitably configured kernel and must be run with superuser privileges.

Finally, a widely used technique to intercept function calls is library interposition (see e.g. Xifer [13]).[15] The idea is to define in a separate library one or several functions with the same name as existing ones and bind them at load time so that they are called instead of the original ones. This works well for named functions with the same restrictions as with *SaBRe*—symbols have to be exported—but with additional constraints. First, only function calls that go through the procedure linkage table (PLT) are interposable. Second, library interposition will not work with statically-linked binaries since it relies on the dynamic loader's doing the final link. Third, `LD_PRELOAD` cannot be used with `setuid` programs for security reasons.

[12] Unfortunately, despite all these constraints, eBPF is not exempt from vulnerabilities: CVE-2017-16995 allowed to bypass the eBPF verifier so as to get unlimited read/write access within the kernel.

[13] https://github.com/pmem/vltrace

[14] https://perf.wiki.kernel.org/

[15] On Linux, this is usually achieved by setting the `LD_PRELOAD` environment variable.

7 Conclusion

In this paper, we have introduced *SaBRe*, a lightweight load-time system for selective binary rewriting. *SaBRe* enables adding, removing and modifying instructions in process memory, both in the program itself and in libraries. We described the main theoretical challenges for accurate rewriting, including those for disassembly with a linear-sweep scheme, and rewriting with a trampoline-based approach. On the practical side, we used the modular plugin architecture of *SaBRe* and its flexible API to build backends for `x86_64` and `RISC-V`, and plugins for system call tracing, multi-version execution and fault injection, which have been evaluated on real applications. The overhead of bare interception, without any added instrumentation, ranges between 0.3 % and 2.8 %. The MVX plugin, which is the most complex, performs on par with the state-of-the-art monolithic Varan system. By targeting the `RISC-V` architecture, not relying on third-party libraries and keeping the main binary below 50 KiB, *SaBRe* is also a better fit for embedded systems under strong memory constraints than any other solution in the literature.

Finally, we remind the reader that *SaBRe* is open source and can be found at:
`https://github.com/srg-imperial/sabre`.

References

1. Andriesse, D., Chen, X., Van Der Veen, V., Slowinska, A., Bos, H.: An In-depth Analysis of Disassembly on Full-scale x86/x64 Binaries. In: Proc. of the 29th USENIX Security Symposium (USENIX Security'16) (2016)
2. Arm Ltd: Conditional execution in Thumb state. `https://developer.arm.com/documentation/dui0473/m/condition-codes/conditional-execution-in-thumb-state`
3. Bansal, S., Aiken, A.: Automatic generation of peephole superoptimizers. In: Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06) (2006)
4. Bauman, E., Lin, Z., Hamlen, K.: Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In: Proc. of the 25th Network and Distributed System Security Symposium (NDSS'18) (2018)
5. Bernat, A.R., Miller, B.P.: Anywhere, any-time binary instrumentation. In: Proc. of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'11) (2011)

6. Bruening, D., Zhao, Q., Amarasinghe, S.: Transparent dynamic instrumentation. In: Proc. of the 8th International Conference on Virtual Execution Environments (VEE'12) (2012)
7. Buck, B., Hollingsworth, J.K.: An API for Runtime Code Patching. The International Journal of High Performance Computing Applications **14**(4), 317–329 (2000)
8. Busybox. https://busybox.net/
9. Card, S., Moran, T., Newell, A.: The model human processor: an engineering model for human performance. Handbook of Perception and Human Performance **2**, 1–35 (1986)
10. Chaignon, P.: Introducing strace –seccomp-bpf. https://pchaigno.github.io/strace/2019/10/02/introducing-strace-seccomp-bpf.html (2019)
11. Chamith, B., Svensson, B.J., Dalessandro, L., Newton, R.R.: Instruction punning: Lightweight instrumentation for x86-64. SIGPLAN Notices **52**(6), 320–332 (2017)
12. Computer Science Department at University of Wisconsin-Madison and Computer Science Department at University of Maryland: Dyninst Programmer's Guide, Release 10.1 (2019). URL https://github.com/dyninst/dyninst/blob/v10.1.0/dyninstAPI/doc/dyninstAPI.pdf
13. Davi, L.V., Dmitrienko, A., Nürnberger, S., Sadeghi, A.R.: Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In: Proc. of the 8th ASIAN ACM Symposium on Information, Computer and Communications Security (ASIACCS'13) (2013)
14. De Sutter, B., De Bus, B., De Bosschere, K.: Link-time binary rewriting techniques for program compaction. ACM Transactions on Programming Languages and Systems (TOPLAS) **27**(5), 882–945 (2005)
15. Debray, S., Muth, R., Watterson, S.: Software power optimization via post-link-time binary rewriting. Tech. rep., University of Arizona (2001)
16. Dinesh, S., Burow, N., Xu, D., Payer, M.: Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In: Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'20) (2020)
17. Duck, G.J., Gao, X., Roychoudhury, A.: Binary rewriting without control flow recovery. In: Proc. of the Conference on Programing Language Design and Implementation (PLDI'20) (2020)
18. ElWazeer, K., Anand, K., Kotha, A., Smithson, M., Barua, R.: Scalable variable and data type detection in a binary rewriter. In: Proc. of the Conference on Programing Language Design and Implementation (PLDI'13) (2013)
19. Fisher, J.A.: Walk-time techniques: Catalyst for architectural change. Computer **30**(9), 40–42 (1997)
20. GNU Coreutils. https://www.gnu.org/software/coreutils/
21. Google Seccomp Sandbox for Linux. https://code.google.com/archive/p/seccompsandbox/
22. Gorgovan, C., d'Antras, A., Luján, M.: Mambo: A low-overhead dynamic binary modification tool for ARM. ACM Transaction on Architecture and Code Optimization (TACO) **13**(1) (2016)
23. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 5 edn. Morgan Kaufmann (2011)
24. Horspool, R.N., Marovac, N.: An Approach to the Problem of Detranslation of Computer Programs. The Computer Journal **23**(3), 223–229 (1980)
25. Hosek, P., Cadar, C.: Safe software updates via multi-version execution. In: Proc. of the 35th International Conference on Software Engineering (ICSE'13) (2013)
26. Hosek, P., Cadar, C.: Varan the Unbelievable: An efficient N-version execution framework. In: Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15) (2015)
27. Hunt, G., Brubacher, D.: Detours: Binary interception of Win32 functions. In: Proc. of the 3rd USENIX Windows NT Symposium (USENIX NT'99) (1999)
28. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual (2021). URL https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
29. Kim, T., Kim, C.H., Choi, H., Kwon, Y., Saltaformaggio, B., Zhang, X., Xu, D.: RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications. In: Proc. of the 33rd Annual Computer Security Applications Conference (ACSAC'17) (2017)
30. Koning, K., Bos, H., Giuffrida, C.: Secure and efficient multi-variant execution using hardware-assisted process virtualization. In: Proc. of the 2016 International Conference on Dependable Systems and Networks (DSN'16) (2016)
31. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proc. of the 13th USENIX Security Symposium (USENIX Security'04) (2004)
32. Laurenzano, M.A., Tikir, M.M., Carrington, L., Snavely, A.: PEBIL: Efficient static binary instrumentation for Linux. In: Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2010) (2010)
33. LeDoux, C., Sharkey, M., Primeaux, B., Miles, C.: Instruction Embedding for Improved Obfuscation. In: Proc. of the 50th Annual Southeast Regional Conference (ACM-SE'12) (2012)
34. Li, T., Bhargava, R., John, L.K.: Adapting branch-target buffer to improve the target predictability of Java code. ACM Transaction on Architecture and Code Optimization (TACO) **2**(2), 109–130 (2005)
35. Lighttpd. http://www.lighttpd.net/
36. Linn, C., Debray, S.: Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In: Proc. of the 10th ACM Conference on Computer and Communications Security (CCS'03) (2003)
37. Lu, J., Chen, H., Fu, R., Hsu, W.C., Othmer, B., Yew, P.C., Chen, D.Y.: The performance of runtime data cache prefetching in a dynamic optimization system. In: Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03) (2003)
38. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proc. of the Conference on Programing Language Design and Implementation (PLDI'05) (2005)
39. Majlesi-Kupaei, A., Kim, D., Anand, K., ElWazeer, K., Barua, R.: RL-bin, robust low-overhead binary rewriter. In: Proc. of the 2nd Workshop on Forming an Ecosystem Around Software Transformation (FEAST 2017) (2017)
40. Matz, M., Hubička, J., Jaeger, A., Mitchell, M., Lu, H., Girkar, M.: System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) (2018)
41. Memcached. http://memcached.org/
42. memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark

43. Nanda, S., Li, W., Lam, L.C., cker Chiueh, T.: BIRD: Binary interpretation using runtime disassembly. In: Proc. of the 4th International Symposium on Code Generation and Optimization (CGO'06) (2006)
44. Nginx. `https://nginx.org/`
45. Nylander, E.: Improved code obfuscation through automatic construction of hidden execution paths. Master's thesis, Lund University (2014)
46. O'sullivan, P., Anand, K., Kotha, A., Smithson, M., Barua, R., Keromytis, A.D.: Retrofitting security in COTS software with binary rewriting. In: Proc. of the IFIP International Information Security Conference (SEC'11) (2011)
47. Pina, L., Andronidis, A., Cadar, C.: FreeDA: Incompatible stock dynamic analyses in production. In: Proc. of the 2018 ACM International Conference on Computing Frontiers (CF'18) (2018)
48. Pina, L., Andronidis, A., Hicks, M., Cadar, C.: Mvedsua: Higher availability dynamic software updates via multi-version execution. In: Proc. of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19) (2019)
49. Prasad, M., Chiueh, T.: A binary rewriting defense against stack based buffer overflow attacks. In: Proc. of the 2003 USENIX Annual Technical Conference (USENIX ATC'03) (2003)
50. Redis. `https://redis.io/`
51. Salamat, B., Jackson, T., Gal, A., Franz, M.: Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In: Proc. of the 4th European Conference on Computer Systems (EuroSys'09) (2009)
52. Schwarz, B., Debray, S., Andrews, G.: Disassembly of executable code revisited. In: Proc. of the 9th Working Conference on Reverse Engineering (WCRE'02) (2013)
53. Schwarz, B., Debray, S., Andrews, G., Legendre, M.: Plto: A link-time optimizer for the Intel IA-32 architecture. In: Proc. of the 2001 Workshop on Binary Translation (WBT-2001) (2001)
54. Shen, B.Y., Hsu, W.C., Yang, W.: A retargetable static binary translator for the ARM architecture. ACM Transaction on Architecture and Code Optimization (TACO) **11**(2), 18 (2014)
55. Smith, J., Nair, R.: Virtual machines: versatile platforms for systems and processes. Elsevier (2005)
56. Smithson, M., ElWazeer, K., Anand, K., Kotha, A., Barua, R.: Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In: Proc. of the 20th Working Conference on Reverse Engineering (WCRE'13) (2013)
57. Van Put, L., Chanet, D., De Bus, B., De Sutter, B., De Bosschere, K.: Diablo: a reliable, retargetable and extensible link-time rewriting framework. In: Proc. of the 5th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT'05) (2005)
58. Volckaert, S., Coppens, B., Voulimeneas, A., Homescu, A., Larsen, P., Sutter, B.D., Franz, M.: Secure and efficient application monitoring and replication. In: Proc. of the 2016 USENIX Annual Technical Conference (USENIX ATC'16) (2016)
59. Wall, D.W.: Global register allocation at link time. In: Proc. of the 1986 SIGPLAN Symposium on Compiler Construction (SIGPLAN'86) (1986)
60. Wang, R., Shoshitaishvili, Y., Bianchi, A., Machiry, A., Grosen, J., Grosen, P., Kruegel, C., Vigna, G.: Ramblr: Making Reassembly Great Again. In: Proc. of the 24th Network and Distributed System Security Symposium (NDSS'17) (2017)
61. Wang, S., Wang, P., Wu, D.: Uroboros: Instrumenting stripped binaries with static reassembling. In: Proc. of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16) (2016)
62. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In: Proc. of the 19th ACM Conference on Computer and Communications Security (CCS'12) (2012)
63. Wenzl, M., Merzdovnik, G., Ullrich, J., Weippl, E.: From hack to elaborate technique—a survey on binary rewriting. ACM Comput. Surv. **52**(3) (2019)
64. Williams-King, D., Kobayashi, H., Williams-King, K., Patterson, G., Spano, F., Wu, Y.J., Yang, J., Kemerlis, V.P.: Egalito: Layout-agnostic binary recompilation. In: Proc. of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20) (2020)
65. wrk. `https://github.com/wg/wrk`
66. Xu, M., Lu, K., Kim, T., Lee, W.: Bunshin: Compositing security mechanisms through diversification. In: Proc. of the 2017 USENIX Annual Technical Conference (USENIX ATC'17) (2017)
67. Yee, B., Sehr, D., Dardyk, G., Chen, J.B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native Client: A sandbox for portable, untrusted x86 native code. In: Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'09) (2009)
68. Zhao, L., Li, G., Sutter, B.D., Regehr, J.: Armor: Fully verified software fault isolation. In: Proc. of the 11th International Conference on Embedded Software (EMSOFT'11) (2011)

A Appendix

We start by providing some formal definitions (§A.1), then assess the main challenges involved (§A.2) and finally demonstrate the propositions underpinning our approach (§A.3).

A.1 Definitions

Definition 1 An *instruction word* is a finite sequence of executable bytes. A *data word* is a finite sequence of non-executable bytes, which may include padding or junk bytes.

Definition 2 A *program* is a finite sequence of both instruction and data words. The *code* of a program is the subsequence derived by deleting data words.

Definition 3 A *code snippet* refers to a sequence of one or several instructions.

We model the *memory* M as a one-dimensional space. Each *memory object* is characterised by its *start address* and its *size*, which determine a connected (or contiguous) subset of M. Let $A(o)$ and $S(o)$ be functions that map an object o into its start address and size, respectively. The distance between two objects is the difference between their start addresses.

Definition 4 A *segment* is a connected (or contiguous) subset of the memory space M.

Each segment has a set of permissions attached to it, usually a combination of *Read*, *Write* and *eXecute*. All instructions must be mapped to executable segments.

Definition 5 The *code coverage* is the proportion of instructions that the disassembler is able to decode. *Under-coverage* occurs when some instructions are missed and thus the program is not fully disassembled. *Over-coverage* occurs when data words are interpreted as instructions. When neither occurs, we have *adequate* coverage.

Note that both types of inadequate coverage can manifest simultaneously, i.e. we could have both under-coverage (some instructions are missed) and over-coverage (some data words are misinterpreted as instructions). Also, note that adequate coverage is not sufficient to guarantee the *accuracy* of the disassembly, as immutability of the code is also required (see Proposition 2).

A.2 Disassembly challenges

The main challenges of disassembling are:

Code discovery. One of the core challenges of disassemblers is to distinguish between data and instruction words. This major theoretical challenge is known since 1978 [24] and is now referred to as the *content-classification* or *code-discovery* problem [18,55] or simply *code discovery*. It is proven to be reducible to the halting problem [24]. The fundamental reason behind the undecidability is that for a snippet to be proven as code it has to be reachable through some execution path. However, some paths may involve indirect jumps whose target addresses can not be known before execution.

On the practical side though, there are solutions to make the distinction between code and data obvious. One of them is to embed some metadata into the memory representation. For instance, one possible convention would be for instruction and data words to have their first bit set to 0 and 1 respectively. This scheme has several major drawbacks, including the waste of encoding space, which is why no commercial ISA uses it. Another, more realistic, design is segregation, either physical or logical. Physical segregation is embodied by Harvard architectures [23, p. L-4] in which code and data have their own separate memories. However, most real-world computers (aside from embedded devices)[16] have taken a hybrid Harvard/Von Neumann approach characterised by a unified memory, mixing both instructions and data. For Von Neumann architectures, logical segregation can be used instead. This can be achieved by having the kernel enforce some partitioning of the address space exposed to the user. Another option is for the executable file format to mandate code and data to be stored in separate segments (as discussed later). In either case, segregation requires cooperation from the compiler as it is the only one to know the original program's semantics.

If the disassembler misinterprets data as code, there are two possible outcomes: (1) the decoded bytes happen to represent a valid instruction, and (2) there exists no such instruction encoding.

[16] In the embedded world, the situation is slightly different. On the one hand, most microprocessors rely on a pure Von Neumann architecture with either no or a single cache and one bus because of its simpler design and reduced footprint. On the other hand, microcontrollers stick to separate flash memory for code and RAM for data.

Case (1) results in over-coverage. If the considered architecture has fixed-length instructions (e.g. MIPS), this is benign. However, if instructions have variable length (e.g. `x86_64`) then an additional hazard is to *derail*, i.e. to desynchronise from the instruction stream. In this situation, the disassembler derives erroneous start addresses for subsequent legitimate instructions and fails to decode them properly. This might happen because such ISAs do not impose any alignment in order to optimise code density.

In case (2) the disassembler skips invalid bytes until it synchronises back to the instruction stream. In the process of coming back on track, the disassembler might temporarily reach a fake execution path, i.e. a sequence of bytes that happen to encode legal instructions. This translates into both under-coverage (since legitimate instructions are missed) and over-coverage (as extra instructions in the fake path are disassembled). However, prior work shows it is rare to miss more than three genuine instructions on `x86_64` [36].

Instruction overlapping and embedding. On ISAs with variable-length instructions, an additional issue is the possibility of having a given snippet encode several execution paths [45]. This is possible if there is some amount of overlap between neighbouring instructions. More formally, there may exist two instruction sequences i and j such that $A(i) < A(j)$ but $A(i) + S(i) > A(j)$, i.e. the last few bytes of i are also the first bytes of j. In this case, a disassembler might only discover the main execution path, namely the one comprising i. On the other hand, j may only be reached through a jump and thus belongs to the hidden execution path.

A variation on overlapping is instruction *embedding* [33] or aliasing, wherein $A(i) < A(j)$ but $A(i) + S(i) \geq A(j) + S(j)$ i.e. some bytes of i happen to encode j entirely. While general overlap across several instructions is only used in obfuscated code, some rare examples of embedding can be found in `glibc` [1], which is the most widespread implementation of the C standard and run-time library among Linux distributions. The purpose of the `glibc` embedding is to prepend an optional lock to a regular instruction, so that this lock can be skipped through a conditional jump.

A.3 Proofs

Proposition 1 *Sufficient conditions for adequate coverage:*

1. *Within a segment, instructions are tessellated, i.e. there is neither gap nor overlap between them.*
2. *The start address and size of executable segments are known.*

Proof Let us denote by I_a the actual set of instructions in the program and by I_d the set of instructions found by the disassembler. We want to prove that $I_\mathrm{d} = I_\mathrm{a}$, i.e both $I_\mathrm{d} \subset I_\mathrm{a}$ and $I_\mathrm{d} \supset I_\mathrm{a}$.

($\subset$) Instructions are tessellated; in particular, there is no gap between them. This ensures that nothing other than code can be encountered in executable segments. Tessellation also guarantees the absence of overlap between instructions. This in turn ensures that no extra instruction can be encountered by derailing into a fake execution path (see Section A.2). Therefore, I_d shall be a subset of I_a.

($\supset$) The start address and size of executable segments are known. Linear sweep guarantees that no instruction on the main execution path in executable segments can be missed. Besides, instructions are tessellated; in particular, there is no overlap between them. Thus there cannot exist hidden execution paths. Moreover, all instructions need to lie in executable segments. Therefore, I_d is necessarily a superset of I_a.

Adequate coverage is not enough to guarantee the accuracy of the disassembly, i.e. its full correctness across the execution.

Proposition 2 *Sufficient conditions for accurate disassembly:*

1. *adequate coverage*
2. *code is immutable*

Proof Adequate coverage means that all instructions are appropriately decoded at the time they are read by the disassembler. Then, under this assumption, the only way by which the disassembly can be inaccurate is if the code changes after it has been disassembled. Code immutability therefore guarantees that any program adequately covered by the disassembler will result into an accurate disassembly ever after.