A Binary-level Thread Sanitizer or *Why Sanitizing on the Binary Level is Hard*

Joschua Schilling[1], Andreas Wendler[2], Philipp Görz[1], Nils Bars[1], Moritz Schloegel[1], and Thorsten Holz[1]

[1]CISPA Helmholtz Center for Information Security
[2]Friedrich-Alexander-Universität Erlangen-Nürnberg

Abstract

Dynamic software testing methods, such as fuzzing, have become a popular and effective method for detecting many types of faults in programs. While most research focuses on targets for which source code is available, much of the software used in practice is only available as closed source. Testing software *without* having access to source code forces a user to resort to binary-only testing methods, which are typically slower and lack support for crucial features, such as advanced bug oracles in the form of *sanitizers*, i. e., dynamic methods to detect faults based on undefined or suspicious behavior. Almost all existing sanitizers work by injecting instrumentation at compile time, requiring access to the target's source code.

In this paper, we systematically identify the key challenges of applying sanitizers to binary-only targets. As a result of our analysis, we present the design and implementation of BINTSAN, an approach to realize the data race detector TSAN targeting binary-only Linux x86-64 targets. We systematically evaluate BINTSAN for correctness, effectiveness, and performance. We find that our approach has a runtime overhead of only 15% compared to source-based TSAN. Compared to existing binary solutions, our approach has better performance (up to 5.0× performance improvement) and precision, while preserving compatibility with the compiler-based TSAN.

1 Introduction

A current central challenge of dynamic software testing is the lack of better and more widely available *bug oracles*, i. e., methods used to identify software faults [4]. Most dynamic software testing techniques, like the widely successful fuzzing, cannot discover many types of faults but rely on such bug oracles. Thus, dynamic bug-finding methods, called *sanitizers*, are often applied to uncover more types of faults. Generally speaking, these sanitizers are commonly implemented as runtime checks embedded into the program [52]. Sanitizers identify and uncover faulty behavior during program execution that cannot be detected otherwise. Over the years, various sanitizers for different types of software faults have been developed, and some of them are widely used in practice [52]. However, most sanitization techniques are unavailable on the binary level, limiting the capability of observing software faults to timeouts and crashes. This affects fuzzers as well as the dynamic testing of targets, which cannot be fuzzed easily.

While naive dynamic testing supports targets without source code, more advanced approaches, such as fuzzing, rely on compile-time instrumentation to steer the testing process. Without source code, they fall back to significantly more costly mechanisms that collect feedback at runtime [17, 22, 23, 25, 36, 42, 57, 60], resulting in severe performance penalties. Only recently, research has started to investigate how compiler optimization-like techniques can be used to enhance binary-only fuzzing [33, 34], but other problems remain. Crucially, most existing binary-only sanitizing approaches [6, 11, 35, 50] often rely on slow dynamic instrumentation approaches. Thus, these new advances in fast binary-only fuzzing highlight the need for appropriately fast, statically instrumented binary-only sanitizers. A dedicated effort to port such sanitizers to the binary level is needed, as undertaken by RETROWRITE [12], which allows for instrumenting binary targets with the popular *Address Sanitizer* (ASAN) instrumentation. To the best of our knowledge, there have been no further attempts at porting other sanitizers, such as MSAN (detector for uninitialized memory reads), UBSAN (undefined behavior detector), or TSAN (data race detector) to the binary level using static binary instrumentation.

In this paper, we study the most popular sanitization methods in practice from a binary-only perspective. First, we systematically analyze the challenges of porting these methods to the binary level, uncovering several obstacles that must be addressed. More specifically, we demonstrate that it is likely infeasible to transfer UBSAN to the binary level. For MSAN, we find that an implementation would be very complex, incompatible with its source-based version, and it would likely incur a significant performance overhead. On the other hand, a binary port of TSAN is challenging but generally feasible and would bring state-of-the-art data race detection with the speed of static instrumentation to binary-only targets. Based on this analysis, we present the design, implementation, and evaluation of BINTSAN, an efficient and effective thread sanitizer on the binary level for Linux x86-64 targets. Our evaluation shows that BINTSAN can uncover data races on the binary level, while also maintaining a low overhead, outperforming existing binary-level solutions, and exhibiting fewer false positives.

In summary, we make the following key contributions:

- First, we systematically analyze the challenges preventing us from porting sanitization methods to the binary level. We define several success criteria and highlight how existing solutions fit these criteria.
- Based on this analysis, we demonstrate how to overcome these challenges by designing and implementing BINTSAN, our prototype of a binary-only version of the popular thread sanitizer TSAN. Thereby, we introduce novel heuristics to identify atomic operations and propose several optimizations to minimize the performance impact of binary sanitizers.
- We evaluate our prototype implementation and find that BINTSAN outperforms existing tools in terms of performance and error detection, while preserving compatibility with compiler-based TSAN.

We open source our implementation, evaluation scripts, and data at https://github.com/CISPA-SysSec/binary-tsan.

2 Challenges for Binary Sanitizers

We now introduce and discuss two main technical barriers that need to be overcome by binary sanitization methods, define success criteria for binary sanitizers, and assess how existing sanitizers may be ported to achieve the outlined criteria.

2.1 Assessment of Technical Barriers

Barrier I: Information Loss. The first barrier is the information loss caused by the compilation process. Binary sanitizers need to reconstruct this lost information to determine the correct locations for sanitization code, while maintaining the semantics of the binary. The reconstruction often depends on heuristics, which usually lead to inaccuracies.

Typical categories of information lost during compilation are control flow information, types, memory order, signedness, and debug information. This information loss is caused by conceptual differences between the source language and the target architecture. Undefined behavior, for example, is a concept used in high-level languages like C/C++ to guide compiler optimizations. However, there is no concept of undefined behavior on the binary level, as only concrete instructions are emitted.

Barrier II: Conceptual Differences in Program Representations. The second barrier encompasses the conceptual differences in the program representations. Different representations of a program (source code, intermediate representation (IR), and assembly) have different properties, which can either simplify or impede static analysis and hence need to be considered during sanitizer development. In the scope of this work, we focus on x86-64 binaries; thus, we focus on conceptual differences between x86-64 assembly and IRs, on which source-based sanitizers operate.

In practice, this results in several conceptual differences: While an IR can support an infinite number of registers, physical hardware has a limited number of registers. Furthermore, an IR using static single-assignment form (SSA) ensures that every register is assigned exactly once, while assembly cannot offer this property. This fundamentally impacts binary analysis, as the register state needs to be considered. Furthermore, an IR usually has a reduced instruction set (RISC), while many instruction set architectures support a complex instruction set (CISC). The x86-64 architecture, for example, supports nearly a thousand unique mnemonics and over three thousand instruction variants [30]. A binary sanitizer must support every instruction, so assembly is often lifted to an IR, which can cause inaccuracies.

2.2 Success Criteria

Based on the previously described challenges, we identify the following five success criteria for binary sanitizers:

Correctness. The semantics of the target program must be retained and not be affected by the introduced instrumentation.

Effective Error Detection. A binary sanitizer should effectively detect issues and potential vulnerabilities. While this may be done differently from its source-based counterpart, both methods should detect the same bugs and minimize false positives and false negatives.

Performance. While multiple dynamic instrumentation-based sanitization tools for binaries exist [11, 16, 35], one crucial weakness of these dynamic approaches is a considerable performance degeneration [34]. However, to a smaller degree, this is also a problem for tools based on static binary instrumentation, which can introduce unnecessary instrumentation that incurs overhead during binary rewriting and runtime. In any case, a binary sanitizer should strive to reduce runtime overhead.

Compatibility. A binary sanitizer should integrate with other tools. Ideally, the sanitizer should be compatible with already existing fuzzing frameworks, pipelines, or ecosystems, such that it can be seamlessly incorporated into existing tooling. This includes compatibility with source-based counterparts for projects with only partly available source code. The resulting method encourages a wider adaption within the community [52]. This is especially important for sanitizers, which depend on continuous information flow, i. e., information propagated from one part of the program to another.

Scalability. Binary sanitizers should be applicable to as many commercial off-the-shelf (COTS) binaries as possible, including large binaries, binaries with exception handling, and binaries that have been obfuscated or stripped. Another significant difference to source-based sanitizers is that binaries commonly lack debug symbols. This results in additional challenges for the user of a binary sanitizer, especially when the binary sanitizer is used to aid the debugging process. Thus, binary sanitizers should support debug symbols if available but should also be functional for targets without them.

2.3 Assessment of Source-Based Sanitizers

Based on the discussed challenges and success criteria, we now analyze the feasibility of porting the four most popular source-based sanitizers to the binary level. While various sanitizers have been proposed in the literature [52], we focus on the four static-rewriting-based sanitizers included in modern compilers [20]. Their availability, ease of use, and continued maintenance lead to a wide adoption within the community. Namely, these four methods are address sanitizer (ASAN) [48], undefined behavior sanitizer (UBSAN) [49], memory sanitizer (MSAN) [53], and thread sanitizer (TSAN) [46, 47].

ASAN can be applied well to binary targets in general, as its instrumentation is relatively lightweight, and the ASAN runtime library implements most logic. For example, handling heap red zones is part of the runtime library and does not require any binary instrumentation. While a binary port of ASAN is generally feasible, the loss of information impairs the effectiveness of a binary version of ASAN, as can be seen for RETROWRITE [12]. For example, RETROWRITE cannot sanitize global variables or individual stack objects. While RETROWRITE comes with its own AFL instrumentation for fuzzing and is compatible with ASAN, its rewriting framework limits its applicability to PIC Linux binaries without C++ exception handling. Still, it causes a performance overhead of 50-70% [12].

UBSAN is different from all other sanitizers, as it uses a multitude of small individual checks, which all need an independent implementation. However, the major issue with a binary version of UBSAN is that while undefined behavior can be used for optimization during compilation, this concept no longer exists on the binary level. Instead, the original intent needs to be reconstructed to decide if the source code's behavior was undefined. This is a challenging, error-prone task. According to our analysis, out of 28 checks provided by UBSAN, only ten can be recreated, even if only partially, for binary targets. This already assumes using heuristics to differentiate between signed and unsigned types. A complete list of these checks can be found in Appendix A. The majority of the (partly) possible checks are related to integer overflows. Other checks require knowledge of the source code, e. g., for alignment, correct usage of compiler builtins, or function signatures to correctly resolve indirect calls. Hence, we expect that the error detection capabilities of a binary-level UBSAN are so limited that we consider a binary port of UBSAN to be likely infeasible.

MSAN features a large runtime library for uninitialized memory detection. However, MSAN's instrumentation is rather complex. The primary reason is that uninitialized memory needs to be correctly tracked throughout the program's execution. Memory state is stored in so-called *shadow memory*. Shadow propagation is used to update the memory state if the memory is accessed or altered. On the binary level, this is even more complex since the contents of registers can influence the initialization state of memory. A typical example is loading a possibly uninitialized memory into a register and storing the register's content in another uninitialized memory location. Thus, to implement shadow propagation, the state of every register and its contents has to be considered. As the concept of shadow registers is not part of the source-based MSAN version, this would cause a compatibility break. More importantly, a CISC architecture like x86-64 is unsuitable for shadow register propagation, as it would require providing specific instrumentation for *all* instructions that may change the value and initialization state of the contents of registers, which is a significant part of the instruction set. Thus, we expect a binary version of MSAN to have a severe performance overhead and require a lot of effort to port its error detection mechanisms.

TSAN requires rather lightweight instrumentation, as only memory accesses have to be instrumented, and the detection logic is delegated to a runtime library. However, the memory order of atomic operations is lost during compilation. The six possible memory orders in C++ [40] result in three different sequential models that are used in practice: In ascending order of guaranteed properties, these are *relaxed*, *acquire-release*, and *sequential-release*. These sequential models differ substantially from the memory consistency model used by x86-64. The most popular memory model that is consistent with this architecture is called the *total store ordering* (TSO) model [37]. This memory model is generally as strong as the acquire-release sequential model, but weaker than the sequentially consistent model. Similarly, the information on atomicity itself is often lost during compilation. Therefore, heuristics must be developed to identify atomic variables to compensate for the loss of information. Overall, we believe a binary port of TSAN to be feasible. While such an implementation would partly have to rely on heuristics to overcome the aforementioned challenges, it would bring state-of-the-art data race detection to the binary level. This includes all benefits of modern-day TSAN, like fast static instrumentation with only minor performance overheads, a combination of lockset and happens-before analysis, and the support of high-level and low-level synchronization mechanisms, which leads to lower false positive rates compared to other tools. In the following section, we present our design and the prototype implementation of a binary-level thread sanitizer called BINTSAN. As part of our evaluation, we assess how well our approach fulfills the previously defined success criteria.

Our analysis suggests that the information loss and conceptual differences between binary executables and source code make a binary-level implementation of UBSAN likely impossible and one of MSAN very challenging. However, the concepts behind the sanitizers ASAN and TSAN can be applied on the binary level.

3 BINTSAN: Binary Thread Sanitizer

In the following, we introduce the design and implementation of BINTSAN, a binary-level method for thread sanitization.

3.1 High-level Overview

The architecture of our approach and the prototype implementation of BINTSAN is shown in Figure 1. The input to our approach is the binary file to be instrumented, which is done using binary rewriting techniques. An important design principle is to use the TSAN runtime library as much as possible, since reusing the runtime library allows BINTSAN to be fully compatible with the source-based TSAN. The sanitizer instrumentation is implemented as an extension of the binary rewriting framework. By following this approach, we can better integrate with existing fuzzing tooling, e. g., further instrumentation plugins can be added, such as coverage feedback for a fuzzer. Finally, the resulting instrumented binary is dynamically linked against the TSAN runtime library and can be used to test a target for potential data races.

3.2 Binary Rewriting Framework

Since binary rewriting is an established technique, we chose an existing binary rewriting framework that provides the functionality to address the challenges discussed in Section 2. As the framework can have a major impact on fulfilling the success criteria of the resulting sanitizer, the choice needs to be carefully weighed. For our prototype, we identified the following key criteria for an ideal framework:

1. supports arbitrary rewriting of x86-64 Linux ELF binaries and supports rewriting of PIC and non-PIC binaries,
2. refrains from heuristic analysis methods for rewriting that may lead to incorrect binaries,
3. is applicable to COTS binaries, including C++ exception handling,
4. supports analysis routines used to guide sanitization, e. g., call graph and control flow graph reconstruction, register analysis, etc.,

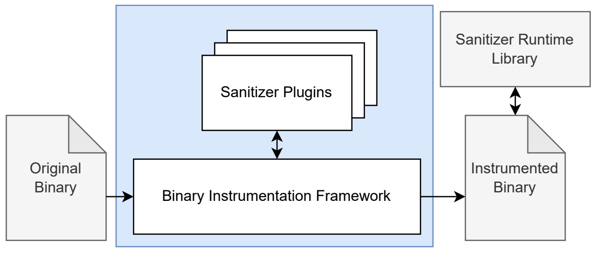

Figure 1: A high-level overview of the interactions between the different components of the binary thread sanitizer.

5. incurs only a minimal runtime performance overhead,
6. supports integration with existing fuzzing tools (e. g., AFL coverage feedback instrumentation), and
7. can use debug information (if present).

We initially considered the following six static rewriting frameworks for BINTSAN: E9patch [13, 19], Egalito [56], Multiverse [2], Ramblr [54], RETROWRITE [12], and ZIPR [21, 24]. While we could not find any framework that satisfies all of our criteria, we chose ZIPR since it came closest. There are two criteria that ZIPR does not fulfill. The first is using heuristics, which can cause issues with specific binaries. Additionally, ZIPR does not handle debug information. To support debug information, we extended the ZIPR framework accordingly. Thus, our implementation is built upon a modified fork of ZIPR[1]. Note that this custom ZIPR version is not required to run BINTSAN; we have successfully tested BINTSAN with various unmodified versions of ZIPR as well.

3.3 Data Race Detection

To implement data race detection logic for BINTSAN, we base our implementation on the runtime library of the source-based TSAN [46]. Here, a hybrid approach of two analyses is used to maximize detection and minimize false positives [31, 46, 47]: In a *lockset analysis* [44], all locks held by a thread are tracked, and data races are detected as two threads accessing the same memory location (at least one writing), without them both holding the same lock. In a *happens-before analysis* [28], the partial ordering of events in a program is tracked. Happens-before relations are created by synchronization events, or within a thread due to execution order. A data race is detected if two accesses from different threads happen to the same memory (at least one writing) and are not ordered with respect to each other.

As the `clang` version of TSAN only provides a static library, we use the dynamic library implemented for `gcc` (version 11.2; note that the library, and thus BINTSAN, is compatible with other versions, as the interface is stable). When linked against an application, system calls and com-

[1]based on commit 96b868a4ce6a703db4703164d3b1250625d80a3e

```
mov    cs:Global, edi
retn
```

Listing 1: Exemplary compilation of function assigning first argument to a global variable.

```
 1  push   rbx
 2  mov    ebx, edi
 3  mov    rdi, [rsp+0x8]
 4  call   __tsan_func_entry
 5  mov    edi, offset Global
 6  call   __tsan_write4
 7  mov    cs:Global, ebx
 8  call   __tsan_func_exit
 9  pop    rbx
10  retn
```

Listing 2: Same function with TSAN instrumentation.

mon C/C++ standard library functions are hooked. This is done to check memory accesses via these calls for data races and to identify thread creation and synchronization primitives, which thus can be detected without modifying the standard library.

3.4 Instrumentation

To enable a data race detection functionality compatible with TSAN in BINTSAN, we must correctly instrument the respective system calls and standard library functions on the binary level. In addition to the sanitization instrumentation, TSAN adds instrumentation to output meaningful call stacks in case a data race is detected. Both types of instrumentation are mimicked by BINTSAN as closely as possible.

We now take a look at how Listing 1 is instrumented by TSAN, which can be seen in Listing 2. Both listings show the generated assembly code for a function that sets a global variable to the argument given to the function. The original function does so with only two assembly instructions. In contrast, the same function compiled with TSAN contains ten instructions with three different function calls to the TSAN library. Lines 3, 4, and 8 are the call stack instrumentation to manage the shadow stack. Lines 5 and 6 are the instrumentation for the memory access itself. They are inserted before the original instruction that writes the global variable in Line 7. Line 5 loads a pointer to the global variable into the argument register *rdi*. The following function call to the TSAN library in line 6 checks the memory access for a data race. The remaining instructions are needed for state saving. Lines 1 and 9 save and restore the state of the register *rbx*; thus, it can be freely used in between. In line 2, *rbx* can now be used to save the function parameter so that *rdi* can be used as a parameter for the call stack instrumentation call in line 4.

During instrumentation, BINTSAN iterates over all instructions and identifies all operands that access memory. For each operand, the corresponding instrumentation code is inserted. Furthermore, three distinct properties are identified and reported to the runtime library for each memory access. These are the *address of the access*, the *type of access* (read or write), and the *number of affected bytes*. The access type and the number of affected bytes are statically known and are passed to the runtime library implicitly via the inserted instrumentation. For example, the function `__tsan_write4` reports a 4-byte write access. On the other hand, the memory access address is not statically known and is passed as the first argument of this function.

Another challenge is the highly complex x86-64 instruction set, which contains various memory access methods. Currently, BINTSAN covers the use of explicit operands, conditional memory operands, and string instructions. These are handled by recreating the condition check or, respectively, calculating the length of the accessed memory for string operations. The prototype implementation of BINTSAN does not handle a few instructions prone to race conditions. I/O instructions like `in` or `out` are not handled, as these instructions are privileged and cannot be used in user-space applications. Furthermore, instructions like `maskmovdqu`, which have implicit memory operands but do not conform to a specific instruction class, are not handled either. In our experience, these instructions are rarely used in practice.

3.5 State Saving

As part of the instrumentation, special care has to be taken to ensure that the semantics of the original program flow are not changed. After each instrumentation point, the same state as before the instrumentation has to be restored. This is especially difficult in BINTSAN's case, as the added instructions usually contain calls to the TSAN runtime library, which may arbitrarily change the program's state. To avoid interference of our state changes with the normal program execution, BINTSAN stores various registers at the beginning of each instrumentation point and restores the original values after exiting the instrumentation code. This includes all general purpose registers that are not maintained during function calls according to the System V ABI [32], the RFLAGS registers, and the XMM registers `xmm0` to `xmm15`. Other registers, like the FPU registers, are not saved and restored, as the TSAN runtime library does not use them.

The state-saving mechanism can potentially create significant performance overhead. To minimize this overhead, BINTSAN carefully selects the registers that must be saved and restored. BINTSAN includes multiple optimizations, as explained in Section 3.9. These optimizations include minimizing the set of used registers in the runtime library as well as a custom dead register analysis, which identifies registers that are not alive at the point of instrumentation and, therefore, do not need to be saved and restored.

3.6 Call Stack Instrumentation

The previously described memory access instrumentation is sufficient to detect data races. However, to produce error messages with meaningful stack traces for such data races, BINTSAN also needs to support the custom shadow stack implementation of the TSAN runtime library. This requires tracking the current call stack for each thread in the runtime library by calling `__tsan_func_entry` for each function entry. Respectively, a call to `__tsan_func_exit` is performed on function exit, removing the topmost object from the shadow stack. BINTSAN supports three types of exits from a function: If the function exits via a `ret` instruction, a function call is directly embedded before the *return statement*.

If the function exits via a `jmp` instruction, a typical result of tail-call optimization, the `jmp` instruction is converted into a regular function call and return instruction. Then regular call stack instrumentation can be applied. However, this transformation is only applied if the called function receives no arguments via the stack. This property is checked via our custom dead register analysis. If one of the argument registers is dead directly before the function call, it is not used, meaning the stack is not used for arguments. However, as this check has to be performed for the general purpose and the XMM registers, it only works with the custom dead register analysis, which—contrary to the STARS analysis implemented in ZIPR—covers the XMM registers. If BINTSAN cannot confirm that the function does not have stack arguments, the call stack instrumentation is omitted for this function.

The last way to exit a function is via *stack unwinding*. In this case, BINTSAN creates and inserts a new landing pad (where appropriate), which performs the necessary call to the `__tsan_func_exit` function.

3.7 Atomic Operations

In x86-64 assembly, some instructions can have the *lock* prefix, which guarantees that these instructions are executed atomically. We make sure that these instructions are detected by BINTSAN as atomical. Similarly, BINTSAN detects the instruction `xchg`, which is always executed atomically (i. e., regardless of whether it has a *lock* prefix or not).

While it is possible to detect atomic operations in many cases, compilers also create assembly code from which it is impossible to deduce the presence of atomic operations. This is due to the differences in memory model and atomic operations between high-level languages like C++ and x86-64. This causes multiple problems for BINTSAN concerning atomic operations. First, in practice, atomic operations are generally not involved in actual data races, as multiple atomic operations on the same variable are usually intended by the programmer. As a result, these are not reported by TSAN. This special handling cannot be mimicked by BINTSAN if atomic operations are not detected, which may lead to false positives. Atomic operations can also create synchronization events. For example, spinlocks act like other locks that block a thread while waiting on a signal from another thread. Therefore, TSAN considers atomic operations for "happens-before" relations. However, if BINTSAN fails to detect such an atomic operation, the synchronization event is missed, which may again lead to false positives.

Another problem is caused by the reliance of TSAN's runtime library on the knowledge of the used memory order by atomic operations. This information is used to create the correct synchronization events and is available during compilation. However, BINTSAN, which uses the same runtime library, cannot always correctly infer the used memory order based on the binary executable. If inference fails, atomic instructions are treated as having the *acquire*, *release*, or *acquire-release* memory orders, depending on the instruction. The drawback of using the *acquire-release* model is that synchronization events not present in the C++ code may be detected, which may lead to false negatives.

Finally, TSAN requires that all atomic operations are executed within the TSAN library itself. The C++ atomic operation `compare exchange` requires two memory orders. This operation compares the value in memory with a given value and exchanges it for a new value if the two are equal. Since the exchange only happens in some cases, the TSAN library has to execute the operation to know the correct memory order to use. For this reason, the execution of all atomic operations is performed within the TSAN runtime library itself. Therefore, BINTSAN replaces the original atomic instruction with a call to the library function. Special handling ensures that the correct registers are passed as arguments and that correct flags are replicated after the function is executed.

3.8 Heuristics

To minimize the impact of undetected atomic instructions, BINTSAN implements three heuristics to identify occurrences of atomic instructions. First, the *initialization of a static variable* is thread-safe by default. To implement thread safety efficiently, compilers like `gcc` and `clang` introduce a guard variable containing the initialization status. The status ensures that only the first thread enters the initialization section. Our first heuristic is used to detect accesses to these guard variables and mark them as atomic operations with the memory order *acquire*.

Memory accesses to the *same memory* (i. e., same local variable) within a function are all marked as atomic if accessed atomically at least once. This heuristic is based on the expectation that if a developer accesses a variable atomically, they likely have considered concurrency problems and added atomic operations to all memory accesses via the variable. This also holds for the C++ `std::atomic` variables, as any access to them is atomic by default. However, this heuristic is only applied if the instruction without the *lock* prefix can have possibly been generated from an atomic operation in C/C++.

Finally, the last heuristic focuses on detecting *spinlocks*. These are identified in the binary by searching for natural loops with at least one read memory access that also fulfill the following criteria: There must be no writing memory accesses and function calls (except for `usleep`) in the loop. All registers that are used to compute the locations of the memory accesses must be loop invariant to ensure that the read memory access reads from the same memory during every loop iteration. If such a spinlock is found, all reading memory accesses within the spinlock are treated as atomic load operations with the memory order *acquire*.

To create the correct synchronization events, the unlocking of the spinlock also needs to be detected. This manifests as simple memory writes in the binary executable, making detection difficult for general cases. However, typical barrier implementations insert the unlock memory write in the same function as the spinlock itself. This allows BINTSAN to detect the unlock memory write, which accesses the same memory location, and declare it as atomic with memory order *release*.

3.9 Optimizations

To minimize the performance overhead for BINTSAN, we design and implement several general optimizations in addition to the previously mentioned technique-specific optimizations.

3.9.1 Skipping Instrumentation

The first optimization is to skip instrumentation for memory accesses that can be statically determined to be thread-safe. This is always the case for *constant memory*. As this memory will never be written to, it can never cause a data race. Therefore, BINTSAN skips instrumentation of all accesses to read-only memory segments.

Similarly, variables only used inside a single thread cannot cause data races. An example of this is function local variables, which are never passed to another thread. However, this is hard to determine in general due to pointer aliasing. Hence, we only consider cases where a function never leaks any pointers to stack variables. To detect leakage, we implemented a function local analysis, identifying if pointers to stack variables are stored in memory or passed as an argument to another function call. If not, all stack accesses within the function are considered thread-safe, and instrumentation can be safely skipped by BINTSAN.

Finally, BINTSAN can also safely omit *compiler generated memory accesses* such as implicit memory operands of stack instructions (e. g., push, pop, enter, leave, call, and ret) and stack canaries. Canaries are detected as accesses to the thread-local storage at the constant offset *0x28*, which is used by both `gcc` and `clang`.

3.9.2 Custom Dead Register Analysis

As described in Section 3.5, BINTSAN needs to correctly restore the original register values after each instrumentation point. To reduce the resulting overhead, the second optimization we implement is a custom dead register analysis. A register is considered dead within a function once it is no longer read in any execution path. ZIPR already implements a register analysis called STARS. However, this analysis does not consider any calling convention or ABI, and labels function calls as reads and writes for all registers. Therefore, we implemented a more aggressive, custom dead register analysis for the System V ABI [32], which, contrary to the STARS analysis, also observes SSE registers.

A simplified dead register analysis works by tracking which registers are dead before and after each instruction and propagating this information within a function. If an instruction reads a register, it has been alive before the instruction, as its value is used. If a register is written, it has been dead before the instruction, as its value is not used. Note that there are instructions that read and write the same register, typically in a read-modify-write operation like the `add` instruction.

The System V ABI specifies that the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9` are used to pass function arguments of the integer class, while `r10` may be used for passing a static chain pointer, which is used by some languages like Ada. Thus, without further information, all of these registers are considered to be read by the function call and thus have been alive before. Therefore, the registers written by a function call that were dead before the call are more challenging to identify. While the calling convention suggests that all caller-saved registers may be overwritten, this is not always true. The problem is that whole-program optimization of modern compilers may identify these cases and consider these registers alive. Therefore, read operations may still happen after the function call if the compiler detects that the register's value cannot have been overwritten during the function call. Therefore, BINTSAN determines the possibly written registers of a function with a preliminary depth-first search on the call graph. For each function, all writing instructions, as well as the called functions, are considered. If the target of a function call cannot be identified statically, due to virtual function calls or calls to libraries, all registers are considered to be potentially written.

Jumps to other functions, typically a result of tail-call optimization, are considered and handled as regular function calls. Indirect jumps are considered to write all registers. As the return value of a function is, according to the System V ABI, passed via the `rax`, `rdx` or `xmm0` register, the `ret` instruction is considered to read these registers.

Note that our custom dead register analysis relies on the System V ABI. If a compiler optimization breaks this ABI, our analysis may not work with it. In such cases, it can be disabled.

3.9.3 Custom Compiled Runtime Library

The SSE registers `xmm0` to `xmm15`, which are used for floating point operations and SSE vector instructions, create a major performance problem for binary sanitizers due to their number and size. In total, 256 bytes are required to save all registers on the stack. Since the TSAN library itself does not require these registers, BINTSAN, in its default configuration, uses a specially compiled version of the TSAN runtime library that limits the use of XMM registers. The compiler option `-mno-sse` prohibits the use of the XMM registers, which allows BINTSAN to skip the saving and restoring of these registers for all TSAN library functions that are compiled with this flag. It is not possible to compile the whole runtime library like this, since library functions like `printf()`, which are required for error reporting, require the SSE registers. Therefore, BINTSAN compiles all performance-critical TSAN functions without SSE registers and only saves and restores these registers if actual data races are reported. Note that for compatibility, BINTSAN can also be used with the regular runtime library, in which case BINTSAN saves and restores all SSE registers.

In summary, we envision that BINTSAN can serve both as a guideline for binary sanitizers and as a low-overhead, static rewriting-based data race detection tool with support for atomic operations for binary targets.

4 Evaluation

With our evaluation, we aim to answer the following five research questions

- **RQ1:** How well is BINTSAN applicable to COTS binaries? Are rewritten binaries still correct?
- **RQ2:** Is BINTSAN effective in detecting data races? What are the tradeoffs in handling atomic operations?
- **RQ3:** How well does BINTSAN perform? How does it compare to compiler-based TSAN or dynamic binary instrumentation tools?
- **RQ4:** What is the impact of the optimizations used in our implementation?
- **RQ5:** Is BINTSAN compatible with other established tools like TSAN [46] or AFL++ [17]?

4.1 Hardware and Environment

Our evaluation was performed on an Intel Xeon Gold 5320 CPU and 256 GB of RAM with SSD memory as backing storage running on Ubuntu 22.04.1 LTS. If not stated otherwise, we use `gcc` version 11.2.0 for compilation. Similarly,

Table 1: List of all applications that were instrumented and tested. We list the size in number of instructions, functions, and memory accesses, the compiler optimization level, the programming language, and if the application is open source.

	Target	#Ins	Func	MemAcc	Opt	Lang	OSS
SPEC CPU 2017	505.mcf_r	4,578	44	1,399	O3	C++	✓
	523.cpuxalan_r	1,077,063	16,977	303,881	O3	C++	✓
	525.x264_r	127,358	539	40,023	O3	C	✓
	531.deepsjeng_r	18,065	112	4,794	O3	C++	✓
	541.leela_r	45,566	416	11,527	O3	C++	✓
	548.exchange2_r	31,148	15	15,207	O3	Fortran	✓
	557.xz_r	33,721	385	8,030	O3	C++	✓
PARSEC	blackscholes	1,214	5	330	O3	C	✓
	bodytrack	107,833	1,330	33,030	O3	C++	✓
	dedup	26,073	129	7,331	O3	C	✓
	ferret	127,357	784	38,979	O3	C	✓
	fluidanimate	5,752	34	1,624	O3	C++	✓
	freqmine	16,370	77	6,892	O3	C++	✓
	raytrace	990,246	5,217	290,585	O3	C++	✓
	streamcluster	8,195	36	2,185	O3	C++	✓
	swaptions	13,969	46	5,420	O3	C++	✓
	vips	115,6099	6,022	306,129	O3	C++	✓
Real-World Programs	QtNotepad	2,851	35	415	O2	C++	✓
	axel	12,026	105	4,186	O2	C	✓
	bifrost	151,442	595	46,237	O3	C++	✓
	bsdtar	136,872	1,347	31,200	O2	C	✓
	csv_parser	5,820	34	1,454	O2	C++	✓
	dns-discovery	1,045	19	241	O3	C	✓
	ffmpeg	3,765,360	19,195	973,942	O3	C	✓
	hdiffz	127,323	914	33,287	O2	C++	✓
	libQt5Core	816,790	10,781	20,0467	O3	C++	✓
	libSwell	147,354	860	38,534	O2	C++	✓
	libbifrost	234,629	1,410	64,182	O2	C++	✓
	lrzip	51,524	375	11,718	O2	C	✓
	lz4	60,892	263	14,351	O3	C	✓
	pbz2	10,307	97	2,154	O3	C++	✓
	pigz	23,678	124	6,024	O3	C	✓
	pixz	6,274	88	2,145	O2	C	✓
	plzip	26,532	821	8,371	O0	C++	✓
	rar	115,383	1,123	28,829	N/A	C++	✘
	reaper	1,965,335	13,331	488,487	N/A	Unknown	✘
	sqlite3	397,932	4,089	189,186	O2	C	✓
	unrar	51,767	715	11,094	O2	C++	✓
	x265	12,004	110	2,985	O3	C++	✓
	xz	9,474	118	1,454	O2	C	✓
	zstd	247,014	991	71,057	O3	C	✓

we use the corresponding version of the source-based thread sanitizer [46] (TSAN v2) for comparison. We also compare against Helgrind [11] in version 3.18.1, the only other actively maintained dynamic data race detection tool for binary targets to our knowledge. AFL-style coverage instrumentation is performed within the ZAFL [34] pipeline (commit f03ce79).

4.2 Data Set

We evaluate BINTSAN on 41 applications listed in Table 1. Further information on their type and version is shown in Table 5 in the appendix. This is the dataset that we use throughout our evaluation unless stated otherwise. The dataset contains the two benchmarks *SPEC CPU 2017* and *PARSEC*, which consist of real-world applications that have been widely used in academic work [10, 15, 29, 51, 58] before, and extend it with further real-world programs.

More specifically, we use the PARSEC benchmark suite in version 3 [3], consisting of benchmark programs designed to evaluate multithreaded processors and processor simulations. We remove the subjects *canneal*, *facesim*, and *x264* from our evaluation, as they terminate with a segfault even without source or binary instrumentation. In addition, we use the SPEC CPU 2017 benchmark in version 1.1.0 [7]. Here, we use seven programs of the SPECrate 2017 Integer Suite (*intrate*), which contains integer CPU benchmark programs based on real-world applications. We omit the targets *gcc_r*, *perlbench_r*, and *omnetpp_r*, as they are incompatible with the used binary instrumentation framework ZIPR. In addition to both benchmarks, we add further real-world applications to cover a wide spectrum of use cases. We gathered our targets from the literature [26], and searched GitHub for the phrases "parallel", "multithreaded", and "command line". We statically check whether the binaries include multithreading and also dynamically confirm that it is actually used when we call the target. We include projects with and without available source code, as well as binaries with and without graphical user interfaces (including stand-alone programs as well as libraries). Furthermore, we also cover complex applications like ffmpeg (21 MB, 3.7M instructions). We focused mainly on open-source software to directly compare BINTSAN to its source-based counterpart TSAN. To model realistic usage scenarios of real-world binaries, we compile all targets as suggested by the developers. Notably, this usually includes higher compiler optimization levels like *-O2* and *-O3*.

4.3 Correctness and Scalability

To evaluate the correctness and scalability (**RQ1**) of BINTSAN, we use automated test cases in combination with manual analysis of our real-world data set. We use the GitHub project c-testsuite [8] (commit 5c72756) for repeated tests and regression testing to ensure the correctness of the instrumented binaries. The c-testsuite contains 220 small C programs as well as the expected outputs of the programs. As these programs are rather small, we extended the automated test set with the Linux core utilities in version 9.1. Most of these programs have around 200 to 1,000 lines of C code. Again, these programs come with test cases that, apart from two exceptions, successfully work with BINTSAN. The first exception is a test case that internally uses Valgrind, which is incompatible with TSAN. The second exception is a known bug in the TSAN library specific to the coreutils library function `aligned_alloc()` [45]. Beyond these artificial test cases and test suites, we manually test BINTSAN extensively with our real-world data set during the scope of our evaluation, which contains a broad spectrum of use cases, as discussed previously. We do not find any differences in behavior after instrumentation, apart from performance and the introduced TSAN behavior.

4.4 Data Race Detection

To evaluate data race detection (**RQ2**), we use the test cases of the source-based TSAN from the LLVM project. These test various features, including unaligned memory data races, atomic operations in various configurations, and deadlock detection. We apply these test cases to our BINTSAN prototype implementation to evaluate whether BINTSAN can correctly identify data races. The test cases are from version 15.0.0 RC1 (commit fd8fd9e) of the LLVM project, which included 330 individual test configurations. To create a baseline, we compile the test cases with the TSAN version based on `gcc` 11.2.0, of which BINTSAN uses the runtime library. As the `gcc` version is older than the implemented LLVM test cases, some test cases are not correctly detected by the baseline TSAN; thus, we exclude those from the evaluation. Of the test cases detected by the baseline, 22 failed when using BINTSAN. Manual analysis revealed seven different reasons for this, with the number in brackets specifying the number of occurrences:

1) Inlined Functions (2x): The source-based TSAN prevents inlining of functions that generate a `rep` string instruction, as for example `memcpy()`. As BINTSAN does not influence the compilation, BINTSAN cannot prevent this. As a result, the inlined function does not appear in the generated stack trace, which causes two test cases to falsely report a failure even though BINTSAN correctly detects the data race itself.

2) Atomic Load/Store (5x): BINTSAN uses heuristics to identify atomic load and store operations. These heuristics failed for five test cases.

3) Unknown Memory Order (6x): Multiple test cases use specific memory orders for synchronization. As it is impossible to infer the used memory order from the binary, BINTSAN uses its default memory order for all cases and therefore fails six test cases that expect a different memory order.

4) Annotations (2x): Two test cases used *happens-before* annotations that BINTSAN cannot infer from the binary.

5) Computed Program Counter (2x): Two test cases use functions from the thread sanitizer library that require a program counter. Since the test cases perform computations with the program counter, ZIPR cannot statically translate the addresses correctly, and the computed program counters are invalid in the rewritten binary.

6) Variable Suppression (1x): Some tests use TSAN's error suppression feature, which allows developers to temporarily suppress known bugs. The functionality is implemented in the runtime library but depends on the names of all objects referenced in the suppression being present in the debug information of the executable. While BINTSAN can provide this information for function names, variable names are not included in the generated symbol table, which causes one test case to fail.

7) VTable Access (4x): The source-based TSAN contains special handling for vtables to facilitate debugging. Memory accesses to vtables are represented in the binary as regular *mov* instructions and, therefore, indistinguishable from reading function pointers out of structs. Thus, it is not feasible for BINTSAN to distinguish memory accesses to vtables, which is why BINTSAN cannot mimic this behavior.

Additionally, some test cases focus on testing specific functionalities in the runtime library, that are irrelevant for BINTSAN. To compensate for this, we add 17 custom test cases that specifically test BINTSAN functionality (e. g., atomic instructions, `rep` string instructions, tail-call transformation, etc.).

In addition to these artificial test cases, we also evaluate the data race detection capabilities of BINTSAN on the real-world applications listed in Table 1. For the real-world programs with available source code, we compare against the results of the source-based TSAN. To create inputs for the programs, we either use fuzzing or manual testing. In both cases, we run each instrumented version of the program against the same inputs. For GUI programs, we use a sequence of user interactions as input. Libraries are indirectly tested as they are included and used within tested applications. As data races caused by thread interleaving are usually non-deterministic, we execute each input three times for every tool in the comparison. The reports of all data races of each tool are then combined, automatically parsed, fingerprinted, and deduplicated. Note that we only consider data races in our evaluation and neglect all other reports, as TSAN and BINTSAN cover a slightly different set of issues than Helgrind.

A fingerprint consists of the type of access (read or write) and the function names of the functions from which the initial as well as the race producing data access originated. We limit the stack trace to the first function. This allows us to group together data races that occur and originate in the same functions but are called in different contexts. The fingerprints are then used to remove duplicates and establish false positive and false negative ratings. We consider all data races detected by BINTSAN but not by TSAN as false positives. Accordingly, we consider all issues found by TSAN but not by BINTSAN as false negatives.

We apply this process to all targets in our data set. The combined results of all runs are summarized in Table 2. Note that this table only lists targets where either BINTSAN or TSAN detected data races. During our benchmark evaluation, a total of 958 data races are detected by BINTSAN in 12 targets. 117 of these are identically identified by TSAN. For most applications, the output of TSAN and BINTSAN are identical or very similar. However, we find that ffmpeg is an outlier. It has the largest code base, which leads to the most detected issues. Also, BINTSAN produces by far the most false positives and false negatives for this program. We attribute this finding to the following three reasons: First, ffmpeg shows a high variance between the three different runs in detected data races for all tools in the comparison, which is an indicator of high non-determinism due to thread-interleaving. Furthermore, a manual analysis of the fingerprints and associated error messages showed that our measuring is often not able to match the identical issues detected in TSAN and BINTSAN. This is caused by the function containing the data race being omitted from the stack trace of BINTSAN, due to compiler optimizations like inlining or tail-call optimization. The last reason for the large number of false positives and false negatives for ffmpeg is likely the heavy use of atomic operations.

Table 2: Data Races (#DR) detection results of BINTSAN, TSAN and Helgrind with the used type of test case generation.

Target	Type	TSAN	BINTSAN			Helgrind			StdDev
		#DR	#DR	FP	FN	#DR	FP	FN	
ffmpeg	Fuzzing	290	95	830	195	74	2,120	216	13.5
bodytrack	Benchmark	6	1	6	5	1	9	5	0.0
QtNotepad	Manual	3	3	1	0	3	3	0	0.6
streamcluster	Benchmark	4	4	0	0	4	1	0	0.0
axel	Manual	4	3	0	1	3	3	0	0.6
vips	Benchmark	3	3	0	0	2	2	1	0.6
bifrost	Fuzzing	3	2	4	1	2	21	1	0.0
lrzip	Fuzzing	2	2	0	0	2	2	0	0.0
rar	Fuzzing	N/A	1	N/A	N/A	0	N/A	N/A	0.0
unrar	Fuzzing	1	1	0	0	0	0	1	0.0
ferret	Benchmark	1	1	0	0	0	2	1	0.0
fluidanimate	Benchmark	1	1	0	0	1	1	0	0.0
Sum		318	117	841	202	92	2,164	225	

Compared to Helgrind, BINTSAN matches or outperforms its data race detection in every aspect for all targets that we tested. We attribute the very high number of false positives produced by Helgrind to its inability to correctly identify atomic operations.

4.5 Atomic Operations Heuristics

Apart from inaccurate measurements, the heuristics implemented for atomic operations are the main cause of false positives and false negatives in BINTSAN. Therefore, we designed an experiment to count how many atomic operations are detected by BINTSAN for each target (**RQ2**). We performed this experiment on all real-world applications listed in Table 1. Note that the results in Table 3 omit targets without findings by our heuristics. The table shows how often each type of atomic operation is detected, either by the lock prefix and the `xchg` or by our heuristics. This is compared to the count of atomic instructions seen in LLVM-IR (Atomic-IR), when the target is compiled with `clang` 14.0.0. Note that this comparison is not exact, due to compiler differences and optimizations, which can add or remove atomic instructions.

To reduce this inaccuracy, we manually inspect a randomly selected subset of 20 identified atomic operations for each

Table 3: Atomic operations detected by BINTSAN on real-world targets. Atomics shows atomics detected via the lock prefix and xchg instructions. IR-Atomics shows the number of atomic instructions in LLVM-IR. Targets without findings by our heuristics are omitted.

Target	Atomics	Same Mem	SV Guard	Spinlock Acc	Total	IR-Atomics
libQt5Core	8,292	5,711	6	0	14,009	21,174
libbifrost	175	89	12	26	302	267
bifrost	163	88	1	26	278	190
ffmpeg	145	38	0	13	196	223
reaper	106	10	0	12	128	N/A
QtNotepad	57	51	0	0	108	112
hdiffz	12	0	0	0	12	20
lrzip	0	4	0	1	5	0
rar	0	4	0	1	5	N/A
bsdtar	2	0	0	0	2	0
sqlite3	1	0	0	0	1	34
x265	0	0	0	1	1	0
zstd	1	0	0	0	1	0

heuristic to verify if the heuristics identified the atomic operation correctly. As all manually inspected atomic operations are confirmed to be correct, we expect that most of the atomic operations are correctly detected. However, our results also show that the heuristics are prone to some false positives, as they detect atomic operations for targets without atomic instructions in LLVM-IR like lrzip and x265.

4.6 Performance

Due to the challenges described in Section 2, one major success criterion for binary sanitizers is matching the performance of source-based sanitizers. As a guiding principle when developing BINTSAN, we accepted a lower instrumentation performance if this tradeoff would result in faster runtime performance. We evaluate both runtime and instrumentation performance (**RQ3**) with the help of two benchmark suites. Note that the subject *freqmine* does not support `pthread` parallelization and is thus removed from our runtime tests. For all runtime evaluations of PARSEC subjects, the provided *simsmall* input sets were used.

4.6.1 Instrumentation Performance

We use BINTSAN to instrument the subjects of both benchmark suites. To get accurate results, we restrict compilation and instrumentation to a single process. On average, the instrumentation takes around five times as long as the compilation of the subject. For TSAN, which can be directly compared with respect to compilation times, an overhead of around 7% is observed. In general, we found that instrumentation using BINTSAN scales linearly with the number of assembler instructions, which enables instrumentation of large binaries. During instrumentation, we observe an average binary file growth of 237%. Note that about two thirds of that growth can be attributed to BINTSAN, while the rest is caused by the ZIPR instrumentation process itself. An overview of all targets can be found in Table 6 in the appendix.

4.6.2 Runtime Performance

To evaluate the runtime performance of BINTSAN, all benchmark targets are executed 5 times with 4 threads. The average of these runs is calculated and referenced as the evaluation result throughout this section. This measurement is performed for each subject without any instrumentation, with source-based TSAN compilation, with binary level instrumentation using our prototype BINTSAN, and with Helgrind to represent dynamic instrumentation tools.

Figure 2 shows the performance comparison of TSAN, BINTSAN, and Helgrind for each subject. The results indicate that, as expected, TSAN has the lowest average runtime overhead by a factor of 18.0× compared to the non-instrumented binary. It is closely followed by BINTSAN, which has an average slow-down of 20.7× (1.15× overhead compared to TSAN). The dynamic instrumentation tool Helgrind has a much higher and less predictable overhead, ranging from 17.1× to 491.1×. It produces an average slow-down of 104.4× compared to the non-instrumented binary and is thus about five times slower than BINTSAN. Note that we have excluded Helgrind's result for *streamcluster*, as it timeouts after 1000x the base execution time.

4.7 Impact of Optimizations

To minimize performance impact, we do not instrument memory accesses that can statically be proven to be thread-safe, as discussed in Section 3.9. As BINTSAN supports multiple types of memory accesses, we examine the respective impact of this performance optimization separately (**RQ4**); the results can be seen in Table 4. On average, 22.6% of all memory accesses can be safely omitted. By far the most common case where our optimizations can be applied are accesses to stack local variables (SLV); 94.8% of the identified omissions can be accounted to SLVs. However, we expect this ratio to change for other real-world targets, as most of our targets were compiled without stack canaries (SC), which accounts for 5.0% of omitted memory accesses. Constant memory reads (CMR) are the least common optimization. To ensure our optimization works as expected and does not exclude non-thread-safe memory accesses, we manually inspected 20 randomly selected accesses for each optimization and found no wrong categorizations. The performance impact of our custom dead register analysis can be seen in Figure 3. For comparison, we also show the performance impact using ZIPR's STAR register analysis. Similarly to our runtime evaluation, we execute the targets five times and present the average values. The average speed-up of the STAR analysis is 15.7%, while our custom dead register analysis features a speed-up of 24.5%. Finally, the total performance overhead of BINTSAN without the default compiled TSAN runtime library is 25.3x. This is about 22.0% higher than with the custom compiled runtime

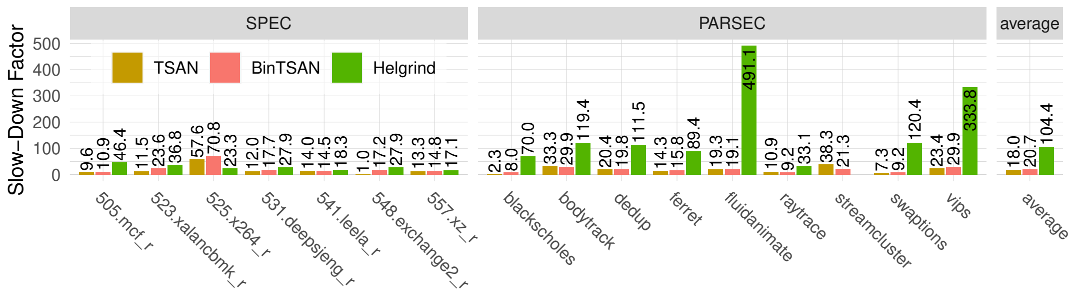

Figure 2: Comparison of the performance overhead of different data race detection tools. Values show the factor of the execution time compared to unmodified subjects. Average shows the geometric mean.

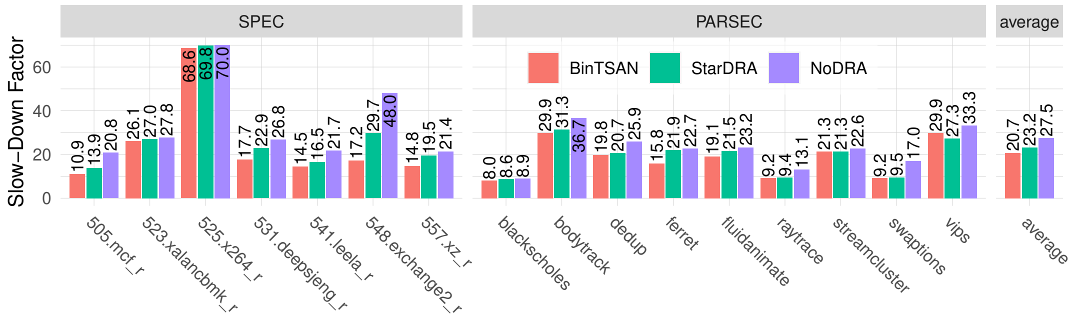

Figure 3: Performance impact of the custom DRA. Average shows the geometric mean.

library. As expected, this optimization performed very evenly through our data set.

Table 4: Impact of our optimizations detecting thread-safe memory accesses on real-world targets. MemAcc shows all memory accesses. We count accesses to stack canaries (SC), stack local variables (SLV), and constant memory reads (CMR).

Target	MemAcc	SC	SLV	CMR	Total Uninstr.
plzip	8,371	388	5,054	1	5,443 (65.02%)
sqlite3	189,186	3,588	90,529	3	94,120 (49.75%)
dns-discovery	241	20	94	0	114 (47.3%)
axel	4,186	108	1,859	1	1,968 (47.01%)
pixz	2,145	121	757	2	880 (41.03%)
libSwell	38,534	0	14,454	0	14,454 (37.51%)
zstd	71,057	1,181	20,503	24	21,708 (30.55%)
lz4	14,351	244	3,653	0	3,897 (27.15%)
ffmpeg	973,942	0	201,312	0	201,312 (20.67%)
hdiffz	33,287	979	5,413	350	6,742 (20.25%)
libQt5Core	200,467	13,157	26,031	131	39,319 (19.61%)
xz	1,454	0	199	0	199 (13.69%)
reaper	488,487	0	65,992	0	65,992 (13.51%)
unrar	11,094	0	1,473	0	1,473 (13.28%)
rar	28,829	0	3,673	0	3,673 (12.74%)
x265	2,985	52	262	64	378 (12.66%)
bsdtar	31,200	1,186	2,436	0	3,622 (11.61%)
pigz	6,024	0	694	0	694 (11.52%)
QtNotepad	415	28	14	0	42 (10.12%)
libbifrost	64,182	1,776	3,618	336	5,730 (8.93%)
lrzip	11,718	0	953	0	953 (8.13%)
csv_parser	1,454	0	99	0	99 (6.81%)
bifrost	46,237	1,072	2,042	12	3,126 (6.76%)
pbz2	2,154	0	143	0	143 (6.64%)

4.8 Compatibility

During our experiments, we implicitly also evaluated the compatibility (**RQ5**) of BINTSAN. More precisely, to show the compatibility of BINTSAN with source-based TSAN, we instrumented libraries like *libQt5Core* or *libSwell* with TSAN, while the program targets using these libraries were instrumented with BINTSAN. This worked without issues and produced similar results to instrumenting everything with BINTSAN. We also used the compatibility with other established fuzzing tools in our evaluation, as we used Zafl's [34] coverage instrumentation and fuzzing toolchain to fuzz our targets in order to create test cases for the data race detection experiment.

In summary, our evaluation shows that BINTSAN incurs only moderate performance overhead compared to source-based TSAN (15%), while significantly outperforming comparable binary-oriented tools such as Helgrind in terms of both performance (5× less overhead) and data race detection capability, with 32% more true data races detected.

5 Discussion and Future Work

In the following, we discuss potential limitations of the prototype implementation of BINTSAN, identify threats to our evaluation's validity, and outline directions for future work on this topic.

Limitations of TSAN Since BINTSAN uses the TSAN runtime library for race condition detection, it inherits the limitations of TSAN. This includes the lack of support for Windows and the restriction to user-space applications. However, an experimental version of TSAN is in development that aims to support kernel code in the future [27].

Limitations of ZIPR Similarly, as BINTSAN depends on the rewriting framework ZIPR, its limitations are inherited. For example, if ZIPR cannot correctly rewrite the binary executable, the inserted instrumentation by BINTSAN also fails or produces faulty binaries. This is more likely if the target program contains obfuscation, DRM mechanisms, self-modifying code, or handwritten assembly. Further, ZIPR does not support some legacy compiler features like dynamic exceptions, which were used by `gcc` before C++11.

Limitations of BINTSAN There are also limitations of BINTSAN itself. Some advanced features of the x86-64 architecture are not supported. This includes transactional memory instructions, for which BINTSAN will produce false positives. Similarly, our prototype does not support 512-bit AVX instructions, as they rarely occur in practice and require high engineering effort to implement. A limitation of the static instrumentation approach taken by BINTSAN is that programs that generate and execute code, like JIT compilers, will not work, as the generated code is not instrumented by BINTSAN. Furthermore, the custom dead register analysis implemented in BINTSAN assumes that function calls conform to the System V ABI [32], which may not always be correct. However, this optimization can be disabled. Finally, `clang` uses a newer version of TSAN than `gcc`, with better runtime performance. However, as our chosen approach requires a dynamic library, which is not available via `clang`, we are limited to the `gcc` implementation.

Threats to Validity A typical internal threat to validity is sample selection. We mitigated this threat by relying on multiple established benchmarks that have previously been used in research. Furthermore, we included real-world targets that have been used in studies about data race detection before. To cover the threat of inappropriately representing the population in general, we relied on a diverse set of targets, which includes applications of different sizes, with and without GUIs, libraries, as well as open-source and closed-source applications from various domains, in which parallel programming is popular (compression, multimedia processing, databases, network applications, and simulations). Another possible threat to internal validity are potential errors in the processing required to evaluate the large amounts of data created by BINTSAN and the comparison tools in our evaluation. While we cannot exclude errors in our experimental infrastructure, we mitigate this threat in two ways. First, we used manual analysis to confirm our quantitative findings, and second, we make our experimental infrastructure and scripts publicly available.

Future Work Our analysis shows that porting source-based sanitizers to the binary level is likely infeasible for UBSAN and requires excessive engineering effort for MSAN. Consequently, we do not see a binary version of either sanitizer as a promising next step. Instead, we expect that static binary sanitizers that aim to detect similar problems as UBSAN and MSAN need to break compatibility with their source-based counterparts and create their own runtime detection mechanisms. A similar approach could also further increase the effectiveness of BINTSAN. We hope to encourage work on these binary-level-focused sanitizers with our paper.

6 Related Work

Previous work on binary equivalents of source-based sanitizers focused on ASAN [9, 12, 16, 55]. For example, RETROWRITE [12] implements ASAN-style instrumentation, compatible with the compiler-based ASAN, and uses a custom binary rewriting framework. Contrary to compiler ASAN, RETROWRITE is not able to differentiate between individual elements on the stack and cannot instrument global variables. Due to the limits of RETROWRITE's lightweight rewriting approach, it supports only PIC Linux binaries without C++ exception handling. Also, RETROWRITE requires an existing symbol table or external tools for function identification. In total, it achieves an overhead of 50-70% compared to the compiler-based ASAN.

A different approach is followed by Chen et al. [9], who leverage hardware features in combination with static binary rewriting to detect spatial and temporal memory violations in heap, stack, and global regions. They rely on the upcoming memory tagging features of ARM to achieve low overheads of 1.82×. However, this new approach breaks compatibility with ASAN, is limited to AArch64 binaries, and cannot be applied to other sanitizers. Another approach is to lift the binary to LLVM-IR and then use compiler-based sanitizers. Altinay et al. [1] use this to apply ASAN to binaries. However, this approach suffers from similar limitations as RETROWRITE and can likely not be generalized to other sanitizers due to loss of information. As the lifting process does not restore any kind of multithreading, it is not suitable for TSAN.

Regarding data race detection, the current approaches can be grouped into two major categories. Static methods [5, 14, 18, 39] rely on ahead-of-time analysis to identify data races. These approaches are often resource intensive and produce many false positives due to approximations. Dynamic approaches [31, 38, 43, 44, 47, 59] detect data races during runtime, which limits them to the executed paths and resulting thread-interleavings. While these methods require exhaustive tests to be useful in practice, they result in few or even no false positives. These approaches typically rely on instrumentation. A further differentiation can be made between approaches that instrument statically before the execution and approaches that do the instrumentation dynamically at runtime. The first group of approaches, such as TSAN [46, 47], require source code to be available. Dynamic tools like Helgrind [11], a part of the Valgrind [35] project, usually incur steep performance overheads. BINTSAN bridges this gap as it combines the benefits of both subcategories. Furthermore, BINTSAN differs from Helgrind regarding the data race detection functionality: Helgrind applies a happens-before analysis, while BINTSAN leverages a combined lockset and happens-before analysis. Also, Helgrind does not support spinlocks and low-level concepts. Similarly, Helgrind fully lacks the atomic operations and memory order support that is offered by BINTSAN. Altogether, this makes Helgrind more prone to false positives in practice, compared to TSAN's and BINTSAN's data race detection approach.

7 Conclusion

In this work, we have analyzed the challenges of implementing sanitizers on the binary level. Our analysis suggests that a binary-only implementation of UBSAN is likely infeasible and that applying the concept behind MSAN to binaries is very challenging. We have presented the design and implementation of BINTSAN, a binary-level thread sanitizer. In a comprehensive evaluation, we have shown how effective and efficient it is in uncovering data races for binary executables.

Acknowledgement

The authors would like to thank our anonymous shepherd and reviewers for their valuable feedback. We also thank the following colleagues and researchers for their help: Franziska Mäckel, Matti Schulze, Keno Hassler, Florian Schweins and Avian Krämer. This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and the German Federal Ministry of Education and Research (BMBF) under the grants KMU-Fuzz (16KIS1898) and CPSec (16KIS1899).

References

[1] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, et al. BinRec: Dynamic Binary Lifting and Recompilation. In *European Conference on Computer Systems (EuroSys)*, 2020.

[2] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[4] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and Reflections. *IEEE Softw.*, 38(3):79–86, 2021.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2002.

[6] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 213–223. IEEE, 2011.

[7] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-generation Compute Benchmark. In *ACM/SPEC International Conference on Performance Engineering*, 2018.

[8] Andrew Chambers. *C-testsuite. Repository*, 2018. URL `https://github.com/c-testsuite/c-testsuite`.

[9] Xingman Chen, Yinghao Shi, Zheyu Jiang, Yuan Li, Ruoyu Wang, Haixin Duan, Haoyu Wang, and Chao Zhang. MTSan: A Feasible and Practical Memory Sanitizer for Fuzzing COTS Binaries. In *USENIX Security Symposium*, 2023.

[10] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[11] Valgrind Developers. Helgrind: A Thread Error Detector. `https://valgrind.org/docs/manual/hg-manual.html`, 2007.

[12] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[13] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary Rewriting without Control Flow Recovery. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[14] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks.

ACM SIGOPS Operating Systems Review, 37(5):237–252, 2003.

[15] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *International Symposium on Computer Architecture*, 2011.

[16] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing Binaries for Memory Safety Errors with QASan. In *IEEE Secure Development (SecDev)*, 2020.

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[18] Cormac Flanagan and Stephen N Freund. Type-based Race Detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[19] Xiang Gao, Gregory J. Duck, and Abhik Roychoudhury. Scalable Fuzzing of Program Binaries with E9AFL. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2021.

[20] Google. *Sanitizers. Repository and Documentation*. Google, 2011. URL https://github.com/google/sanitizers.

[21] William H Hawkins, Jason D Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W Davidson. Zipr: Efficient Static Binary Rewriting for Security. In *Conference on Dependable Systems and Networks (DSN)*, 2017.

[22] Marc Heuse. *Binary-only Fuzzing with Dynamorio and Afl*, 2018. URL https://github.com/vanhauser-thc/afl-dynamorio.

[23] Marc Heuse. *Afl with pintool*, 2018. URL https://github.com/vanhauser-thc/afl-pin.

[24] Jason Hiser, Anh Nguyen-Tuong, William Hawkins, Matthew McGill, Michele Co, and Jack Davidson. Zipr++ Exceptional Binary Rewriting. In *Workshop on Forming an Ecosystem Around Software Transformation*, 2017.

[25] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. TIFF: Using Input Type Inference to Improve Fuzzing. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[26] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[27] Andrey Konovalov. Kernel Thread Sanitizer (KTSAN). https://github.com/google/kernel-sanitizers/blob/master/KTSAN.md, 2021.

[28] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21:558–565, 1978.

[29] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In *USENIX Security Symposium*, 2021.

[30] William Mahoney and J Todd McDonald. Enumerating x86-64 - It's not as easy as counting. Technical report, University of Nebraska Omaha, 2021.

[31] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-race Detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[32] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99(2013):57, 2013.

[33] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[34] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through Binaries: Compiler-quality Instrumentation for better Binary-only Fuzzing. In *USENIX Security Symposium*, 2021.

[35] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007.

[36] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2020.

[37] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 Memory Model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, University of Cambridge, Computer Laboratory, 2009. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-745.pdf.

[38] Eli Pozniansky and Assaf Schuster. MultiRace: Efficient on-the-fly Data Race Detection in Multithreaded C++ Programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[39] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. Locksmith: Context-sensitive Correlation Analysis for Race Detection. *ACM SIGPLAN Notices*, 41(6):320–331, 2006.

[40] The GNU Project. *Memory model synchronization modes. GCC Documentation.*, 2012. URL https://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicSync.

[41] The LLVM Project. *UndefinedBehaviorSanitizer Documentation.*, 2023. URL https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[42] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[43] Paul Sack, Brian E Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas. Accurate and Efficient Filtering for the Intel Thread Checker Race Detector. In *Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[44] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[45] John Scott. Obsolete Check in aligned_alloc(). https://github.com/google/sanitizers/issues/1495, 2022.

[46] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Workshop on Binary Instrumentation and Applications*, 2009.

[47] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic Race Detection with LLVM Compiler. In *International Conference on Runtime Verification*, 2011.

[48] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference (ATC)*, 2012.

[49] Kostya Serebryany. Sanitize, Fuzz, and Harden your C++ Code. In *USENIX Enigma*, 2016.

[50] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference (ATC)*, 2005.

[51] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[52] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for Security. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[53] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[54] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[55] Yuchao Wang and Baojiang Cui. The Study and Realization of a Binary-Based Address Sanitizer Based on Code Injection. In *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2020.

[56] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. Egalito: Layout-agnostic Binary Recompilation. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[57] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. SLF: Fuzzing Without Valid Seed Inputs. In *International Conference on Software Engineering (ICSE)*, 2019.

[58] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative Taint Tracking (STT) a Comprehensive Protection for Speculatively Accessed Data. In *IEEE/ACM International Symposium on Microarchitecture*, 2019.

[59] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Symposium on Operating Systems Principles (SOSP)*, 2005.

[60] Google Project Zero. *WinAFL*, 2016. URL https://github.com/googleprojectzero/winafl.

Table 5: Type of all applications in the evaluation data set. For benchmark programs, the version/commmit of the benchmark is stated.

	Target	Type	Version
SPEC CPU 2017	505.mcf_r	Route Planning	
	523.cpuxalan_r	XML Conversion	
	525.x264_r	Video	
	531.deepsjeng_r	Tree Search	1.1.0
	541.leela_r	Tree Search	
	548.exchange2_r	Sudoku	
	557.xz_r	Compression	
PARSEC	blackscholes	Simulation	
	bodytrack	Body Tracking	
	dedup	Compression	
	ferret	Similarity Search	
	fluidanimate	Simulation	
	freqmine	Data Mining	
	raytrace	Simulation	3.0
	streamcluster	Data Mining	
	swaptions	Simulation	
	vips	Image	
Real-World Programs	QtNotepad	Text Editor	1
	axel	Download Account	2.17.11
	bifrost	Graph Software	v1.3.1
	bsdtar	Compression	60f086b
	csv_parser	Data Parser	de17df6
	dns-discovery	Network	60f086b
	ffmpeg	Multimedia	N-110171-g2244722f1f
	hdiffz	File Comparison	60f086b
	libQt5Core	GUI Lib	5.15.2
	libSwell	Emulation Lib	1
	libbifrost	Graph Software	v1.3.1
	lrzip	Compression	0.651
	lz4	Compression	a3042b9
	pbz2	Compression	1.1.13
	pigz	Compression	2.7
	pixz	Compression	294875b
	plzip	Compression	1.9
	rar	Compression	6.22 beta 1
	reaper	Audio	6.8
	sqlite3	Database	3.42.0
	unrar	Compression	6.22 beta 1
	x265	Video	3.4.1
	xz	Compression	5.4.3
	zstd	Compression	06b5b37

Table 6: Size comparison of binaries in the evaluation data set. The original file size, the instrumented file size, the total binary growth and the share of growth (SOG) caused by BINTSAN is given.

	Target	Size	Instr. Size	Growth	SOG
SPEC CPU 2017	505.mcf_r	34.94 KB	112.47 KB	221.93%	60.07%
	523.cpuxalan_r	7.25 MB	29.27 MB	303.81%	58.56%
	525.x264_r	674.98 KB	2.43 MB	259.52%	80.04%
	531.deepsjeng_r	104.61 KB	339.03 KB	224.09%	67.06%
	541.leela_r	267.31 KB	868.98 KB	225.08%	63.92%
	548.exchange2_r	174.3 KB	708.57 KB	306.51%	86.85%
	557.xz_r	215.34 KB	531.93 KB	147.02%	56.61%
PARSEC	blackscholes	18.81 KB	54.89 KB	191.85%	60.07%
	bodytrack	588.92 KB	1.73 MB	194.42%	58.97%
	dedup	158.42 KB	484.77 KB	206.01%	73.66%
	ferret	644.47 KB	2.13 MB	230.33%	74.1%
	fluidanimate	47.81 KB	152.17 KB	218.29%	67.95%
	freqmine	88.6 KB	365.0 KB	311.96%	77.4%
	raytrace	5.53 MB	18.76 MB	239.04%	72.09%
	streamcluster	51.82 KB	147.25 KB	184.18%	65.96%
	swaptions	80.3 KB	171.38 KB	113.43%	73.12%
	vips	6.3 MB	20.7 MB	228.38%	67.4%
Real-World Programs	axel	80.66 KB	166.6 KB	106.54%	70.29%
	bifrost	937.17 KB	3.54 MB	277.52%	73.51%
	bsdtar	792.8 KB	2.09 MB	163.2%	61.54%
	csv_parser	43.81 KB	181.94 KB	315.3%	73.04%
	dns-discovery	18.57 KB	45.61 KB	145.63%	64.26%
	ffmpeg	20.76 MB	59.82 MB	188.1%	57.31%
	hdiffz	679.28 KB	2.55 MB	274.67%	73.91%
	libQt5Core.so	5.52 MB	17.59 MB	218.64%	73.52%
	libSwell.so	753.13 KB	3.0 MB	298.28%	75.9%
	libbifrost.so	1.43 MB	5.29 MB	270.05%	74.11%
	lrzip	283.03 KB	1.12 MB	295.06%	74.19%
	lz4	321.77 KB	668.57 KB	107.78%	70.87%
	pbz2	80.76 KB	274.72 KB	240.17%	69.61%
	pigz	146.24 KB	488.12 KB	233.78%	60.88%
	pixz	55.45 KB	126.62 KB	128.37%	41.5%
	plzip	167.08 KB	543.25 KB	225.14%	80.92%
	rar	541.49 KB	2.47 MB	356.15%	75.03%
	reaper	10.58 MB	45.14 MB	326.46%	72.17%
	sqlite3	1.98 MB	4.7 MB	137.39%	86.81%
	unrar	319.28 KB	1.29 MB	305.25%	65.23%
	x265	162.01 KB	268.13 KB	65.5%	51.72%
	xz	81.18 KB	203.19 KB	150.31%	37.83%
	zstd	1.19 MB	3.32 MB	179.82%	76.25%

A Appendix

Table 5 lists the type and version for all applications we have evaluated, showing the broad range of our tested programs. For benchmark programs, we include the version number of the benchmark.

Table 6 shows the file size growth for all applications we have evaluated. Furthermore, the proportion of size growth that can be attributed to BINTSAN is given. The rest of the growth is caused by the rewriting process of ZIPR.

Table 7 contains a list of sanitization features included in UBSAN as defined in the clang documentation [41]. The table also contains a short explanation of why certain features are impossible at the binary level. Note that partially possible features are still marked as possible and that this assessment assumes the use of a heuristic to distinguish between signed and unsigned types. Other type inference heuristics have to be very precise for the sanitization checks to work as expected, and they appeared in our preliminary experiments as not sufficient. However, this might change in the future with new advancements in type inference research.

Table 7: Overview of UBSAN sanitization features [41], with a short assessment of whether these are portable to the binary level or not.

Sanitization	Description	Binary	Explanation
alignment	Use of a misaligned pointer or creation of a misaligned reference. Also sanitizes assume_aligned-like attributes.	✘	Type information is not accurate enough.
bool	Load of a bool value that is neither true nor false.	✘	Type information is not accurate enough.
builtin	Passing invalid values to compiler builtins.	✘	Information on compiler builtins is not available.
bounds	Out of bounds array indexing, in cases where the array bound can be statically determined.	✘	Static bound information is not available.
enum	Load of a value of an enumerated type that is not in the range of representable values for the enumerated type.	✘	Information on what values are representable is not available.
float-cast-overflow	Conversion to, from, or between floating-point types that would overflow the destination. Because the range of representable values for all floating-point types supported by Clang is [-inf, +inf], the only cases detected are conversions from floating point to integer types.	✓	
float-divide-by-zero	Floating point division by zero.	✓	
function	Indirect call of a function through a function pointer of the wrong type.	✘	Function type information is not accurate enough.
implicit-(un)signed-integer-truncation	Implicit conversion from integer of larger bit width to smaller bit width, if that results in data loss. That is, if the demoted value, after casting back to the original width, is not equal to the original value before the downcast.	✓	
implicit-integer-sign-change	Implicit conversion between integer types, if that changes the sign of the value. That is, if the original value was negative and the new value is positive (or zero), or the original value was positive and the new value is negative. Issues caught by this sanitizer are not undefined behavior but are often unintentional.	✓	
integer-divide-by-zero	Integer division by zero.	✓	
nonnull-attribute	Passing null pointer as a function parameter that is declared to never be null.	✘	Non-null type information is not accurate enough.
null	Use of a null pointer or creation of a null reference.	✘	Pointer type information is not accurate enough.
nullability-arg	Passing null as a function parameter that is annotated with _Nonnull.	✘	Non-null type information is not accurate enough.
nullability-assign	Assigning null to an lvalue that is annotated with _Nonnull.	✘	Non-null type information is not accurate enough.
nullability-return	Returning null from a function with a return type annotated with _Nonnull.	✘	Non-null type information is not accurate enough.
objc-cast	Invalid implicit cast of an ObjC object pointer to an incompatible type. This is often unintentional, but it is not undefined behavior; therefore, the check is not a part of the undefined group.	✘	Type information is not accurate enough.
object-size	An attempt to potentially use bytes for which the optimizer can determine that they are not part of the object being accessed.	✘	Struct type information is not accurate enough.
pointer-overflow	Performing pointer arithmetic that overflows or where either the old or new pointer value is a null pointer.	✓	
return	In C++, reaching the end of a value-returning function without returning a value.	✘	Function type information is not accurate enough.
return-nonnull-attribute	Returning null pointer from a function that is declared to never return null.	✘	Non-null type information is not accurate enough.
shift	Shift operators where the amount shifted is greater or equal to the promoted bit-width of the left-hand side or less than zero, or where the left-hand side is negative. For a signed left shift, also checks for signed overflow in C and for unsigned overflow in C++.	✓	
unsigned-shift-base	Check that an unsigned left-hand side of a left shift operation doesn't overflow.	✓	
signed-integer-overflow	Signed integer overflow, where the result of a signed integer computation cannot be represented in its type.	✓	
unreachable	If control flow reaches an unreachable program point.	✘	Compiler based reachability analysis is not available.
unsigned-integer-overflow	Unsigned integer overflow, where the result of an unsigned integer computation cannot be represented in its type.	✓	
vla-bound	A variable-length array whose bound does not evaluate to a positive value.	✘	Array bound information is not available.
vptr	Use of an object whose vptr indicates that it is of the wrong dynamic type or that its lifetime has not begun or has ended.	✘	Type information is not accurate enough.