# HSFI: accurate fault injection scalable to large code bases Regular paper

**Conference Paper** · June 2016

CITATIONS
0

READS
59

**2 authors:**

Erik Van der Kouwe
VU University Amsterdam
**7** PUBLICATIONS **10** CITATIONS

SEE PROFILE

Andrew S. Tanenbaum
VU University Amsterdam
**372** PUBLICATIONS **14,381** CITATIONS

SEE PROFILE

# HSFI: accurate fault injection scalable to large code bases

Regular paper

Erik van der Kouwe
Computer Systems Section
Faculty of Sciences, Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
Email: erik@minix3.org

Andrew S. Tanenbaum
Computer Systems Section
Faculty of Sciences, Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
Email: ast@cs.vu.nl

***Abstract*—When software fault injection is used, faults are typically inserted at the binary or source level. The former is fast but provides poor fault accuracy while the latter cannot scale to large code bases because the program must be rebuilt for each experiment. Alternatives that avoid rebuilding incur large run-time overheads by applying fault injection decisions at run-time. HSFI, our new design, injects faults with all context information from the source level and applies fault injection decisions efficiently on the binary. It places markers in the original code that can be recognized after code generation. We implemented a tool according to the new design and evaluated the time taken per fault injection experiment when using operating systems as targets. We can perform experiments more quickly than other source-based approaches, achieving performance that come close to that of binary-level fault injection while retaining the benefits of source-level fault injection.**

## I. Introduction

Despite significant advances in bug detection tools, software bugs continue to be a major source of system outages [1]. This means that we have no choice but to accept the presence of bugs as a given and devote our attention to mechanisms that allow systems to tolerate bugs. To build such systems, we need to be able to quantify fault tolerance to verify whether we are doing a good job. Fortunately, fault injection has been shown to be an effective way to measure the impact of real faults [2], [3].

Although there has been substantial research into software fault injection and several techniques are widely used, each approach has important limitations. Injection of hardware faults such as bit flips is not representative of software bugs [4] and neither are faults injected at the interfaces between components [5], [6]. The main alternative is to use software mutations, modifying software to introduce deliberate bugs similar to those that a real programmer might accidentally introduce. While the tested software is no longer identical to the original, this approach is useful if we need to verify fault tolerance mechanisms in the unmodified part of the code. For example, if an operating system (OS) has a trusted code base (TCB) we can use this approach to verify that the TCB can recover from bugs elsewhere in the OS.

Software mutations can be performed at several levels. At the binary level, the machine code is analyzed to recognize coding patterns and mutate them directly in the machine code. Unfortunately, Cotroneo et al. [7] have shown that in practice this approach does not accurately reflect the intended (source-level) faults. This can be explained by the fact that context information is lost when the compiler transforms a program's source code into binary code. The main alternative is to modify the source code itself. Daran et al. [2] and Andrews et al. [3] have shown this approach to be effective in mimicking real software faults. Source-level fault injection is increasingly popular [8] but unfortunately it scales poorly to large code bases due to relinking overhead [9]. Van der Kouwe et al. [10] have reduced this overhead by injecting multiple faults at a time and selecting the desired ones at run time but this incurs a severe run-time performance penalty. To the best of our knowledge, there is currently no approach that allows high-accuracy software mutations on large code bases without a substantial performance penalty.

In this paper, we introduce a new design called HSFI (Hybrid Software Fault Injection) that allows source-level context information to be used to inject faults while making it possible to enable and disable those faults in the binary without rebuilding. As a consequence we do not have to rebuild the code for each experiment, which allows the approach to scale to large code bases without substantial run-time overhead. This allows for substantial savings when testing fault tolerance of large software systems such as operating systems. These savings are becoming even more important as we move towards multiple-fault models [11], which greatly increase the number of potential fault scenarios. Our technique complements other approaches to reduce the cost of fault injection, such as pruning the search space to test the most important failure scenarios first [12]–[15] or performing multiple experiments in parallel [16]. Moreover, within the domain of software faults, our design is agnostic to the fault model and our implementation can easily be extended with new fault types or fault injection policies.

Our core idea is to inject many disabled faults at the same time before the code generation phase and selectively enable individual faults by directly modifying the machine code in the binary. For each injected fault, two versions of the code are generated: a pristine version that performs the original operations as specified in the source code and a faulty version where those operations are mutated according to the fault type. For each injected fault, we generate a marker that our binary pass recognizes to determine which code sections are pristine and which are faulty. It uses this information to

efficiently enable or disable individual faults without the need to rebuild the program.

Our evaluation shows that our approach performs well and is highly effective at marker detection. It provides performance close to that of binary-level fault injection while traditional source-level approaches result in substantial overhead.

### A. Contributions

This paper makes the following contributions:

- A new design point in software fault injection that combines availability of source-level information with low overhead even for large code bases.
- An implementation of this design that outperforms source-level fault injection in most cases and is close to binary-level fault injection performance in all cases.
- An empirical evaluation that shows in which cases it is beneficial to use our new approach.

## II. Background

To show the need for a new approach for fault injection, in this section we discuss why fault injection itself is important. Next, we briefly go through the various types of fault injection currently used in academic research and their purposes. Finally, we consider available techniques for software fault injection in more detail to demonstrate why we need a new trade-off between availability of source-level information and performance on large code bases.

Fault injection is used for a wide variety of purposes, usually related to quantifying and/or improving reliability. Although fault injection is used in many different domains and our solution is also widely applicable, we will focus particularly on fault injection into operating systems. Operating systems are typically very large pieces of software and they perform a critical role in system reliability as a failing operating system causes the entire system to be unavailable. Because operating systems run at a high privilege level, bugs can do a lot of damage. Fault injection has been used on operating systems in many different ways. Examples include evaluating operating system stability in case of faults in the operating system itself [10], [17], [18], testing isolation properties to protect against faulty device drivers [19], [20], determining to what extent faults can go unnoticed and cause corruption later [21], certification of safety properties [22], and improving fault tolerance by finding the most likely situations in which data can be lost [23]. Fault injection is a critical tool that is necessary to reach more acceptable levels of operating system reliability in the face of faults.

The first step of any fault injection campaign should be to decide on a *fault model*. The fault model specifies what types of faults will be injected and should be consistent with the types of faults the target system will be expected to tolerate. Fault models can be broadly categorized in three groups that typically require different fault emulation techniques to be used: faults external to the system that are observed at the system's interfaces, faults in the hardware the system is run on, and faults in the software of the system itself.

Fault models where interface-level faults are injected are appropriate to test system responses to external changes without changing the target system itself. An external fault could occur in a library (for example LFI [24]), another software component on the system or another system entirely (such as FATE [13] which emulates I/O failures). In all cases, such faults can be easily emulated by providing an incorrect response at the interfaces that delimit the target component, for example by replacing a system library or interposition in inter-component communications. Although these models are suitable to evaluate robustness in the face of unexpected external events, they are not appropriate for cases where we must deal with software bugs in the target component itself [5], [6].

We will refer to models where hardware faults are emulated as hardware fault injection, although it should be noted that this term is sometimes used to indicate that hardware is used as a tool to inject faults (such as for example MESSALINE [25]). However, many more recent systems emulate hardware faults in software; for example, FINE [26] and Xception [27]) inject faults in program binaries, LLFI [28] injects hardware faults in intermediate compiler representation, and GemFI [29] uses a cycle-accurate full-system emulator to inject hardware faults. Hardware fault models have been widely used for a long time and are well-researched. Typical example fault types include memory faults, CPU faults, bus faults and I/O faults, all of which can be either permanent or transient [26]. Although such fault models are suitable to research the impact of failing hardware, Madeira et al. [4] show that they are not suitable to emulate software faults.

We refer to the final group of fault models as *software fault injection* because software faults are being injected. Again, it should be noted that the term is used inconsistently in the literature: it is sometimes used to indicate that software is used as a tool to inject faults (as for example in FERRARI [30]). While the other types of fault model have been widely used for a long time, software fault injection has started to see widespread use mostly in recent years [8]. Software fault injection gives rise to a number of unique challenges. Software faults, also known as bugs, are introduced accidentally by human programmers. This means that fault models cannot be derived from system designs or API specifications, but should be empirically derived from mistakes humans make in practice. Several such models have been proposed, most notably G-SWFIT [31], which has seen considerable use in the literature. Unlike other fault models, software faults are always persistent and the target system itself must be modified. Using hardware to emulate faults is not practical and run-time approaches like virtualization are not needed. It should be noted that, although persistent, software faults need not be deterministic, as for example race conditions can be persistent yet nondeterministic. Since improving software fault injection is the focus of this paper, we will continue by discussing the implications of these unique properties of software fault injection.

After deciding on a specific fault model, the next step is to identify *fault candidates* in the target system. A fault candidate is the combination of a code location and a type of fault that can be introduced there according to the fault model. Because software faults are introduced by humans, we must consider the context in which they wrote the code to determine whether it is a viable fault candidate. After all, the introduced

faults must be representative of those that a real programmer would make and that may depend on the context. After fault candidate identification, one or more fault candidates are chosen for each experiment. The fault injector then mutates the program to introduce the selected fault types at the selected locations. Finally, the modified target system will be run with a workload to examine the impact of the injected fault(s).

One common way to implement software fault injection is to perform these steps on the binary after it has been generated by the compiler and the linker, either in the binary file or in the executable image after it has been loaded into memory. We will refer to this approach as *binary-level fault injection*. This approach is used, for example, by the widely used G-SWFIT [31] tool and has also been used to improve the RIO file cache [23]. Binary-level fault injection can be very efficient because recompilation and relinking is not necessary and no run-time overhead is introduced. Moreover, this approach applies even if no source code is available. To identify fault candidates in binary-level fault injection, the fault injector disassembles the machine code in the binary and identifies programming constructs in the resulting assembly code. It also introduces Mutations at this level. Unfortunately, however, context information is lost when binary code is generated. This means that binary-level fault injectors have limited knowledge about the context. For example, when functions are inlined or loops are unrolled, the binary fault injector cannot distinguish these cases from code that was manually written multiple times. Because information is lost, faults injected at the binary level do not accurately represent bugs at the source level, as shown by Cotroneo et al. [7]. Another drawback is that each processor architecture requires its own fault injector. It is likely that different fault candidates will be detected on different architectures, which limits comparability between systems. Such differences may arise even when just comparing different compilers or compiler settings on the same architecture.

The main alternative is *source-level fault injection*. Now, fault candidates are identified and mutations performed on the source code, before the compiler generates machine code. A widely used example is SAFE [32], which is based on the G-SWFIT fault model. As an alternative to modifying the source code, this approach can also be implemented as a compiler pass (like in SCEMIT [33]). In either case, no context information is lost; the code seen by the fault injector is exactly what the programmers wrote. This approach has been shown to be representative of real software faults [2], [3]. However, every time mutations are performed in the source code, the system must be rebuilt to obtain a modified binary. While a system such as `make` can usually reduce the compilation time by recompiling only files that have been changed, the linking step can take a lot of time for large programs such as operating systems and must be performed every time one or more source files are changed. Hence, the rebuild time can become a dominant factor in fault injection experiments on large code bases [10]. Although it is possible to bypass rebuilding by injecting multiple faults and selecting one or more at run time, this results in substantial run-time overhead [10], which takes away some of the gains. This overhead is especially important for long-running workloads and makes testing of timing-related faults such as race conditions less representative of real-world conditions. We conclude that, although there are many techniques available for software fault injection, none of them combines the good accuracy of source-level techniques with the good performance offered by binary-level techniques.

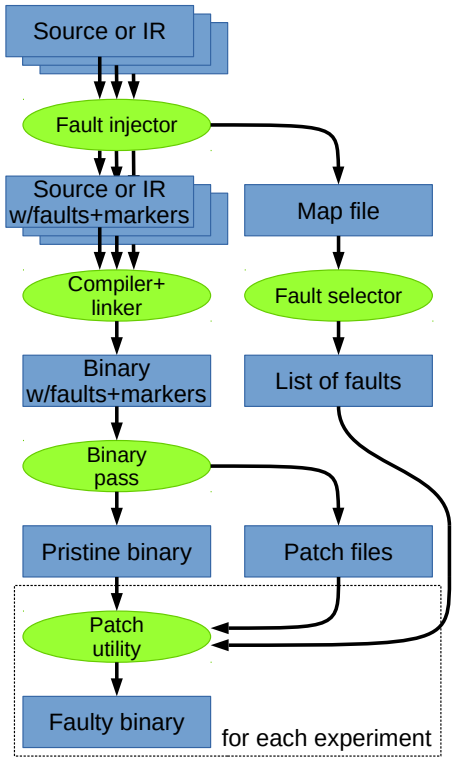


Fig. 1. Fault injection design; IR=intermediate representation

## III. Overview

In this section we describe the steps performed by a fault injector using the HSFI design. Fig. 1 provides an overview of this design. We only discuss elements that are fundamental to the design in this section, leaving elements that are specific to our implementation of the design for the next section.

When setting up a fault injection campaign, the first step is to select an appropriate fault model. Although our design is specific to the injection of software faults, it is agnostic to the exact fault types defined by the fault model and these can be chosen freely. For maximum representativeness it is recommended to select a fault model based on software bugs found in real-world software, for example using prior research such as that of Duraes et al. [31] or Kidwell et al. [34]. It is also important to ensure that the selected faults are representative of residual faults that are likely to elude testing and end up in production systems [32].

The second step is to run the fault injector on the program code. Depending on the implementation this can be done either on the source code directly (in a language such as C) or inside the compiler on an intermediate representation (IR) used by the compiler to represent the parsed code. The fault injector identifies fault candidates and injects as many faults as possible into the code. It does so by duplicating the target code and then applying the appropriate fault type to one of the copies. This yields one *pristine* copy that executes the original code and one *faulty* copy that the fault has been

injected in. It then inserts a *marker* in the code that allows the two copies to be identified even after the source code or IR is converted into machine code. In addition, it writes a *map file* that lists all the markers, including context information such as the fault type and the original code location that helps in fault selection later on. The addresses of the markers in the binary are not yet known at this point.

After running the fault injector pass, the program can be compiled and linked into an executable binary. Because many faults have been injected by the fault injector, this step does not need to be repeated for every experiment like in traditional source-level fault injection.

Now that we have the binary, we run a binary pass on it. This pass scans through the binary and identifies the markers. For each marker, it modifies the binary to ensure that the pristine code is executed. It also creates a *binary patch* for each fault that can be used to switch to the faulty version instead. Although this pass is architecture-specific, it only needs to recognize predefined markers and perform very simple patches. While some porting effort would be required to support multiple architectures, this does not affect the fault injection code itself. Moreover, our approach injects semantically equivalent faults regardless of the underlying architecture.

Before each experiment, we use a patch tool to enable the fault(s) that are to be tested. This is the only part that needs to be repeated for each experiment. Patching is near-instant because the patches usually change just a few bytes. Moreover, it can be done in-place, which avoids making a copy of the (potentially large) binary. In this case the fault patches must be reverted after the experiment. After injecting the appropriate faults, one can run a workload to exercise the code and observe the system's response to the faults.

## IV. Implementation

We have implemented a fault injector and a binary pass according to the design described in the previous section. We implemented our fault injector using the intermediate compiler (IR) representation. We use LLVM [35] as it provides an extension framework that allows the creation of passes to analyze and modify the IR during the compilation process. Using the LLVM IR has the benefit of greatly simplifying code manipulation by leveraging LLVM's parsing and rewriting mechanisms and it provides portability to multiple source languages. Simplifying code manipulation is important because it makes adding fault types easier, which simplifies adapting our tool to different fault models. This approach comes at a small accuracy cost compared to approaches that operate directly on the source code because the compiler expands preprocessor macros, but the IR is a good and platform-independent representation of the source code otherwise [36]. If macros are considered to be critically important, they can be replaced by function calls using demacrofication [37].

LLVM provides a plug-in for the GNU Gold linker to link IR code. Fig. 2 compares the traditional build process with source-level or binary-level fault injection (on the left) to our approach (on the right). The main difference is the fact that we swap linking and code generation. This provides the benefit of being able to run the pass over all the code at once, giving access to context information from the entire program.

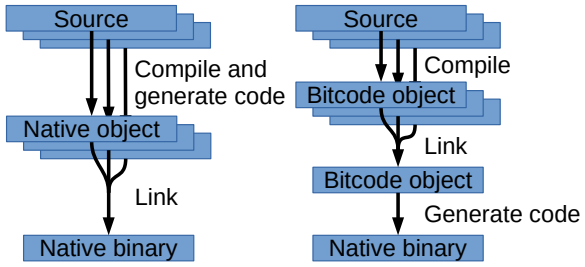


Fig. 2. Traditional compilation (left) and LLVM with IR linking (right)

In this section we will discuss implementation details of our fault injection solution based on the HSFI design. First, we explain how fault candidates are identified and the code is modified to apply them conditionally. Next we discuss how fault candidate markers are inserted in the IR and later recognized in the binary. Finally, we consider how binary patches are generated to enable and disable specific faults.

### A. Injecting faults

We first compile the program to IR using the Clang compiler. Afterwards, the Gold linker links the IR files together, which allows for easy integration in existing build systems. It also allows our tool to operate on all code at once, providing maximal context information. Code generation is postponed until after fault injection, causing source-level information to remain available to the fault injector.

Our fault injector is an LLVM pass which is invoked using the LLVM optimizer. The fault injector loops through each LLVM instruction in the code and determines for which available fault types the instruction is a suitable target (a fault candidate). For example, an assignment operation could be a target for injecting a "wrong value assigned to variable" fault type and the instruction performing the memory store is considered a fault candidate for that fault type. The pass writes a map file containing information that is potentially relevant to the fault selector later on. This file lists all modules, functions, basic blocks and instructions. Where known, it specifies which line of which file in the source code they derive from. Finally, for each instruction it lists applicable fault candidates and their types.

The actual fault injection is performed using basic block cloning, based on EDFI [36]. A *basic block* is a node in the control flow graph (CFG), defined as a code sequence with exactly one entry and one exit point. Once the start of a basic block is reached, it is executed in its entirety unless its execution is interrupted by a deviation from the CFG that is not visible to the compiler, such as an exception or a signal. At the end of the basic block, it selects a new block for execution. For example, consider the code in Fig. 3. The left-hand side of Fig. IV-A shows what the CFG of this code looks like, with the boxes representing basic blocks (the nodes of the CFG) and the lines representing the control flow between them (the edges of the CFG).

To insert faulty code while retaining the original code, we duplicate every basic block that is subject to fault injection. One of the clones remains pristine while a single fault is injected into the other one by applying a mutation operator

```
1  void setanswer(int *p) {
2    if (p != NULL) {
3      *p = 42;
4    }
5  }
```

Fig. 3. Code example for basic block cloning

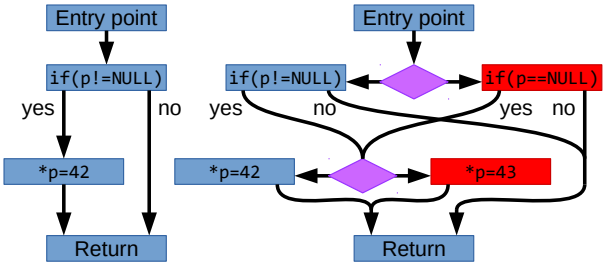


Fig. 4. Control flow graph of the code example before (left) and after (right) fault injection

to that clone. Our implementation only supports one fault per basic block, which means one of the fault candidates must be selected at this point. We currently use a uniform random distribution, but a specific distribution tailored to the fault model can be used instead. The injected fault is documented in the map file. We could create further clones to allow simultaneous injections in the same basic block, with the drawback that the code size increases exponentially with the number of faults per basic block. After cloning, the incoming edges are redirected to a new basic block which we refer to as the *fault decision point* (FDP). The FDP embeds the new basic block into the CFG by conditionally selecting either of the basic blocks. Outgoing edges are cloned, but may be changed by the injected fault. The result is a duplicated CFG with all original code as well as many injected faults. The right-hand side of Fig. IV-A shows our example code after injecting two faults, with the diamonds representing the FDPs.

### B. Fault candidate markers

To identify pristine and faulty basic blocks in the binary, we need markers in the IR (or, alternatively, the source code) that can be recognized after the code generation step. Unfortunately, debug symbols are not sufficiently accurate for this purpose and they are often even less accurate after optimization. Moreover, they have no concept of pristine or faulty basic blocks, nor of basic block identification. We use the FDPs introduced by the basic block cloning step as markers. We insert a global variable and generate FDPs that test whether this variable matches the number of the basic block the FDP was generated for. This test determines whether it jumps to the pristine basic block (when not equal) or the faulty one (when equal). These tests cannot be optimized away as the value of the global variable is only known at run-time and the compiler must assume it can change during execution.

To find the markers, we search for the address of the global variable (which can be looked up in the symbol table) in the disassembly of the program. For this purpose we use Dyninst. From our experiments on the x86 architecture, we have verified that every reference to this address is indeed an FDP. Moreover, we were able to find almost all inserted FDPs in the binary (we found a false negative rate of just 0.01 %, see section V-C). In all cases, the basic block number is directly compared against the value of the global variable in a single instruction, which means it is very simple to determine exactly which basic block ended up where. Any deviations from this pattern (for example due to optimizations) is reported by our tool, but we have found none. We conclude that this is a reliable way to pass information across the code generation step without the need to change the compiler itself. Moreover, although the binary code is architecture-dependent, recognizing an address is very simple to implement on other architectures if a suitable disassembler is available.

### C. Binary patching

As explained, the markers in the binary code are comparisons of a global variable with the basic block number. In all instances where the address of the global is used we found a compare opcode directly followed by either a jump-if-equal or jump-if-not-equal opcode. To enable the pristine code and make the faulty code unreachable, our binary pass simply replaces the comparison and jump-if-equal instructions with no-operation instructions and jump-if-not-equal instructions with unconditional jumps. This can all be done in place. Because the patching is so simple, it would be easy to port to other architectures if the following assumptions hold: there exists a no-operation instruction with a size suitable to overwrite a comparison or conditional branch and there exists an unconditional branch instruction no longer than a conditional branch. Our program issues a warning for any unrecognized code sequences, which means it is easy to debug and ensure every FDP is indeed patched.

While patching the binary to assure that only the pristine code is accessible, the binary pass also writes binary patch files that can be used to switch individual blocks from pristine to faulty and vice versa. These patches simply reverse the cases where conditional branches are replaced with no-operation instructions and the cases where they are replaced with unconditional branches. For most basic blocks, only up to six consecutive bytes in the binary need to be changed to switch between correct and faulty execution, which makes patching nearly instantaneous.

As a performance optimization, we assign branch weight metadata to the FDPs that indicate that the pristine block is very likely to be selected. As a consequence, LLVM places all faulty basic blocks at the ends of functions. This means that in practice no extra jumps are inserted in the common code path and cache pollution is limited. However, some slowdown is still to be expected because the compiler has to set up register usage in a way that works for both the pristine and the faulty basic block. The presence of no-operation instructions and the expansion of the total code size are also expected to have a small performance impact.

## V. Evaluation

In this section we will evaluate our system in terms of performance, both run-time performance and the total time it takes per experiment, and marker recognition accuracy. We

have chosen to evaluate HSFI with operating systems because they are large pieces of software (making scalability to large code bases important) for which reliability is of particular importance. This is demonstrated by the fact that there is much prior work evaluating operating system stability in the face of faults (for example [10], [17]–[23]). We have chosen to use the Linux and MINIX 3 operating systems to evaluate our system. Linux is widely used, particularly in situations where reliability is especially important such as servers. We use LLVMLinux [38], which allows Linux to be built with LLVM IR linking. We use MINIX 3 because it has a modular design, making it structurally different from Linux and demonstrating that our approach is applicable to a wide range of operating systems. Moreover, it is designed specifically for reliability and comes with support for LLVM IR linking out-of-the-box. As workloads, we use Unixbench [39] and the MINIX 3 POSIX test suite. Unixbench is widely used for performance measurements of Unix-like operating systems The MINIX 3 POSIX test set is an example of a regression test workload that aims for high coverage, which is important to perform representative fault injection experiments [40]. Unfortunately it contains some tests that are specific to MINIX 3. We have omitted those tests on both systems to keep the results comparable. This is not a limitation of HSFI, which is applied to the OS itself in this experiment and therefore workload-agnostic. We test all combinations of operating systems and workloads, with the operating system itself instrumented for fault injection.

We performed the compilation of the target systems as well as the tests themselves on Intel Xeon E5-2630 machines with 16 cores at 2.40 GHz and 128 GB of memory, running 64-bit CentOS Linux 7. We used the lto-3.11 branch of LLVMLinux and the llvm_squashed branch of MINIX 3. The operating systems were virtualized using QEMU 2.3.0 [41] with KVM enabled. We virtualized the operating systems because this is the most convenient way to perform fault injection experiments on operating systems. We modified QEMU to provide a hypercall interface that allows the guest to report its progress by writing to a fixed physical memory address. This allows times to be measured and logged on the host, with the benefit that the measurements are more accurate and the logs are retained even if the guest system crashes (which is likely to happen when faults are injected into the operating system).

We compare our new approach against binary-level fault injection, source-level fault injection and EDFI [36] with reduced linking overhead as proposed by Van der Kouwe et al. [10]. To achieve a fair comparison, we have taken an optimistic approach of estimating the overhead of competing techniques. To evaluate binary fault injection, we use an uninstrumented binary and consider the build time only once for all experiments. This means we assume run-time overhead and instrumentation time to be zero. We achieve the same for source-level fault injection by measuring run-time performance on the uninstrumented binary and measuring the rebuild time by changing the last modified date of a single small source file and calling *make*. This means we assume run-time overhead and instrumentation time to be zero and recompile time to be minimal. These decisions allow us to compare the performance of our solution against the fastest possible alternatives.

The fault model we use is taken from EDFI [36], which in turn is based on G-SWFIT [31]. Table I provides an overview

TABLE I. CODE METRICS FOR THE TARGET PROGRAMS

| | Linux | MINIX 3 |
|---|---|---|
| Modules loaded during execution | 1 | 28 |
| Functions | 34578 | 3389 |
| Basic blocks | 313028 | 41884 |
| LLVM instructions | 1817389 | 244775 |
| Fault candidates | 2281880 | 343176 |
| Lines of code | 6775204 | 1128797 |
| Binary code size uninstrumented (KB) | 646 | 995 |
| Binary code size EDFI (KB) | 2230 | 3158 |
| Binary code size HSFI (KB) | 1553 | 2525 |

for the code size and number of fault candidates found for both targets. Most numbers are reported by our compiler pass. Here we included only the code that is actually visible to the compiler; that is, we exclude source files not compiled and code disabled by preprocessor conditionals and we count static libraries used by multiple modules only once. The number of lines of code is counted by the Cloc utility [42] and includes all code files that are not disabled due to configuration or architecture. The binary code size is computed as the sum of executable sections in the ELF files, so in this case static libraries are counted twice to accurately represent the memory overhead introduced by our solution.

In this section, we first discuss the run-time performance of the various approaches. Next, we present the build times and use those to compare the average time taken for each experiment. Then, we show how well our marker detection approach does in terms of false positives and false negatives. Finally, we discuss some potential threats to validity that might influence our results presented here.

### A. *Run-time performance*

Our approach introduces a run-time overhead compared to binary-level and source-level approaches because basic block cloning is performed throughout the program. Although the branches are fixed to avoid any extra conditional jumps, this does affect code generation and the larger code size also affects caching. We have measured the time taken by the system to boot and the time taken to run the MINIX 3 test set. The results are presented in Table II and Table III respectively. The "system CPU usage" indicates how much of the time the CPU has been running system code as reported by the "time" utility (system time divides by real time). The numbers presented are medians over 32 runs and the numbers between parentheses are sample standard deviations. The high standard deviation on the binary/source MINIX 3 boot is due to an outlier where the boot time was excessively long; the boot times are consistent otherwise and the result is not affected because the median reduces the influence of outliers. The numbers show that our approach generally incurs a modest slowdown that is at most about 10% compared to binary and source approaches. EDFI show considerably larger slowdowns. It is important to note that our instrumentation only affects system code. The fact that MINIX 3 reports larger slowdowns seems to be due to the fact that more of the time is spent running system code, which may mean that MINIX 3 is less efficient at handling system calls due to its modular design. This is consistent with the fact that system CPU usage is higher for MINIX 3. [1] Based on the total run

[1] FiXme Note: why is CPU usage MINIX 3 with EDFI so low?

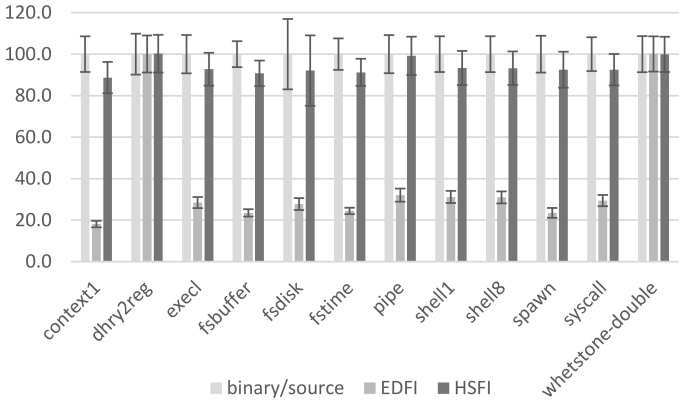


Fig. 5. Unixbench performance on Linux (higher is better)

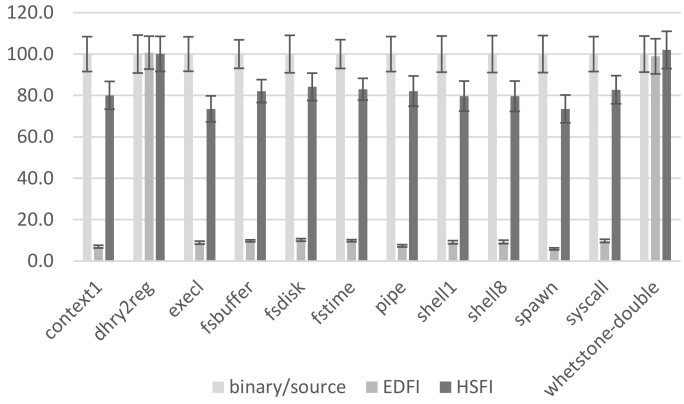


Fig. 6. Unixbench performance on MINIX 3 (higher is better)

times and the percentage of time spent by the CPU executing system code, we made an estimate of the overhead while running system code, which is included in Table III. As expected, it shows considerably larger overhead for system code than for the time spent executing the test set as a whole. Nevertheless, HSFI still has a reasonable overhead compared to source and binary approaches (below 20 % for both Linux and MINIX 3) while EDFI shows even larger overheads than before.

While the MINIX 3 test set gives an indication of the slowdown for a full high-coverage workload, it runs at low CPU usage and does not stress the system very much. To provide an indication of system performance under load, we have run Unixbench 32 times on both systems. The resulting performance numbers are shown in Fig. 5 and Fig. 6. These numbers are normalized to 100 for the uninstrumented system. They represent the number of operations that can be performed in a fixed time, so that higher numbers indicate better performance. Unixbench scores are based on averages where the 33% worst results are left out to reduce the influence of outliers. The error bars give the sample standard deviation based on all experiments. It should be noted that the *dhry2reg* and *whetstone-double* benchmarks do not use the operating system, so the lack of slowdown is to be expected. Although there appears to be a minor speedup, it is well within the margin of error and expected to be a random measurement error. The other subtests all give similar results. To compare the overall results, we have taken the geometric means of the scores for those other tests. EDFI averages 26.6 on Linux and 8.6 on MINIX 3, showing a very large slowdown of respectively about 3.8x and 11.6x. The fact that the slowdown is higher on MINIX 3 suggests that it handles system calls less efficiently (as was also seen with the test set), executing more (instrumented) operating system code for each call. These are substantial slowdowns that cause stress-test experiments to take considerably longer. HSFI averages at 92.6 (Linux) and 80.0 (MINIX 3), so the slowdown is just 1.1x and 1.3x respectively. This means run-time overhead is just a minor factor compared to EDFI.

The previously presented performance results all involve instrumentation of the operating system code. To give an indication of the impact of our instrumentation on a more diverse set of programs, we instrumented the C-based SPEC benchmarks [43] and measured their performance. These tests were performed on the same system as the operating system-

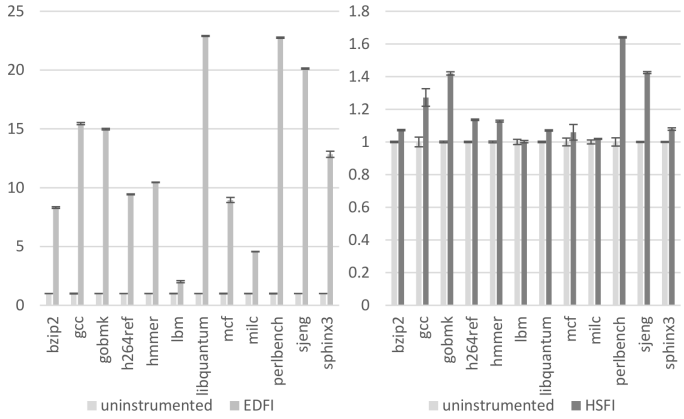


Fig. 7. Instrumentation slowdown on SPEC (lower is better)

based tests but without virtualization. The results are presented in Fig. 7. The graph shows instrumentation slowdowns, computed as the median instrumented run time divided by the median uninstrumented run time. The error bars show standard deviations. The geometric mean over all benchmarks of the slowdown due to EDFI instrumentation is 10.6x, with slowdowns ranging from 2.01x to 22.9x. For HSFI the geometric mean is 1.18x, with a range from 1.00x to 1.64x. Therefore, we conclude that our run-time performance is an order of magnitude better than EDFI and that our instrumentation causes an acceptable slowdown for a wide range of programs, not just operating systems.

### B. Time taken per experiment

To determine how long each experiment takes, we must measure not just the run time but also the time needed to prepare the experiment. Different experimental setups require different steps to be performed for each experiment. Table IV shows an overview. For binary-level approaches, the target program only needs to be compiled once because faults are injected directly into the binary. For source-level approaches, rebuilding the system introduces very large overhead. Every time a source file is changed, the build system must go through the source tree, recompile the appropriate file(s) and eventually relink the entire program. In the case of Linux, some postprocessing on the binary is also required to perform linker tricks such as support for kernel modules. Both EDFI and HSFI require the system to be fully built only a few times

TABLE II. BOOT TIME (LOWER IS BETTER, STD. DEV. IN PARENTHESES)

| | | Binary/source | | EDFI | | HSFI | |
|---|---|---|---|---|---|---|---|
| Linux | Boot time (s) | 7.9 | (1.1) | 11.5 | (1.1) | 8.7 | (1.1) |
| | Slowdown (%) | | | 45.6 | | 10.5 | |
| MINIX 3 | Boot time (s) | 9.5 | (9.5) | 24.6 | (6.7) | 9.7 | (0.8) |
| | Slowdown (%) | | | 159.8 | | 2.0 | |

TABLE III. RUN TIME AND OVERHEAD ON MINIX 3 TEST SET (LOWER IS BETTER, STD. DEV. IN PARENTHESES)

| | | Binary/source | | EDFI | | HSFI | |
|---|---|---|---|---|---|---|---|
| Linux | Run time (s) | 369.6 | (12.4) | 457.6 | (17.1) | 372.6 | (14.2) |
| | Slowdown (%) | | | 23.8 | | 0.8 | |
| | System CPU usage (%) | 7.8 | (0.4) | 25.1 | (0.7) | 8.6 | (0.4) |
| | System overhead (%) | | | 298.5 | | 11.5 | |
| MINIX 3 | Run time (s) | 392.9 | (11.0) | 1096.7 | (28.4) | 410.1 | (5.8) |
| | Slowdown (%) | | | 179.1 | | 4.4 | |
| | System CPU usage (%) | 21.7 | (0.5) | 23.0 | (2.9) | 24.6 | (0.6) |
| | System overhead (%) | | | 196.9 | | 18.5 | |

for many experiments, namely when all faults injected on the previous build have been tested. In addition, they require their passes to be applied to the system to perform fault injection and (in the case of HSFI) generate patch files from the binary. For HSFI, the binary must be patched for every experiment. In principle this is near-instant as it requires only modifying a few bytes at a known file offset. However, in case of Linux it takes longer because we also have to rebuild the boot image, which includes a compressed version of the kernel. It should be noted that we use link-time optimization (LTO), which is needed for our approach but for Linux is not a default configuration. Without LTO Linux takes 734.5s (std. dev 6.2s) to build. For fairness, we use this number for source-level injection in our comparison. We would also like to point out that we do not use parallel make because with source-level experiments one build is needed for each run, so if we use cores for multiple simultaneous runs it is most efficient to also perform the different compilations simultaneously. The numbers show that rebuilding introduces a large amount of overhead, especially for a large program such as Linux. Although the passes also take some time to run, this is not needed for every experiment.

To compare how efficient the approaches are overall, we need to compute the average time taken per experiment. Here, it is very important the know how often the "few times" from Table IV actually is. With our implementation, many faults can be injected with one compilation, but just one for each basic block. This means the number of largest number of selected faults in a single basic block determines how often we must recompile. It is not clear a priori how large this number will be and the probability distribution is fairly complicated. If we assume a fixed number of faults is selected for injection and equal probability of selection for each fault candidate, the number of faults selected in a basic block is binomially distributed but the number of faults selected per basic block is not independent between basic blocks; a large number selected in one leaves fewer for the others. For this reason, we took the distribution of fault candidates per basic block for both target systems and performed a Monte Carlo simulation [44] to determine the number of rebuilds needed for various numbers of faults to inject. For this simulation, we assume that each fault candidate is equally likely to be selected. The results are shown in Fig. 8 (Linux) and Fig. 9 (MINIX 3). There is clearly a linear relationship between the number of faults injected and the number of rebuilds needed, although the ratio between the two differs between the operating systems. This difference is

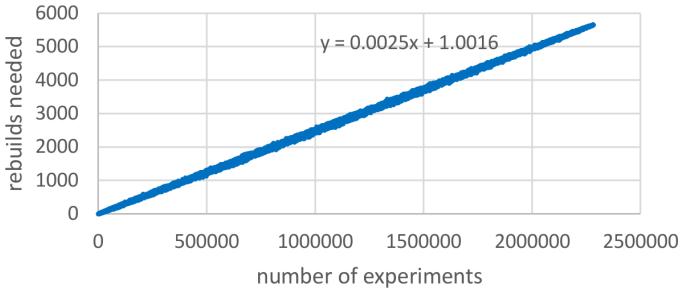


Fig. 8. Monte Carlo simulation of rebuilds needed for Linux with HSFI

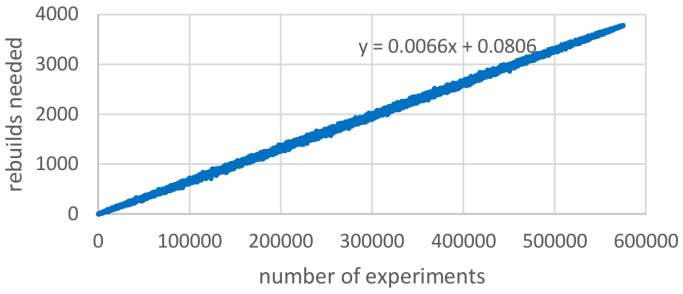


Fig. 9. Monte Carlo simulation of rebuilds needed for MINIX 3 with HSFI

caused by the distribution of fault candidates over basic blocks; if there are relatively more large basic blocks with many fault candidates, more recompiles should be needed on average.

Now that we know the boot time, the slowdown of the workload, the time it takes to rebuild and instrument the system and how often a rebuild is needed, we can compute the average time taken per experiment by adding up the expected values of the numbers. The results are shown in Fig. 10 (Linux) and Fig. 11 (MINIX 3). We show both the slowdowns from the MINIX 3 test set to represent a high-coverage regression test workload and Unixbench to represent a stress test. Source-level fault injection is suitable on smaller code bases with large workload durations, but introduces substantial rebuild overhead otherwise. EDFI, on the other hand, works well for large code bases but only in case the workload duration is short and the workload is not too intense. With stress tests or long-running workloads the run-time overhead becomes a problem. HSFI, on the other hand, always achieves performance close to binary-level fault injection, regardless of the target system and workload.

TABLE IV. MEDIAN TIME TO PREPARE EXPERIMENTS IN SECONDS (STD. DEV. IN PARENTHESES)

| | Needed for | | | | Time taken (s) | | | |
|---|---|---|---|---|---|---|---|---|
| | Binary | Source | EDFI | HSFI | Linux | | MINIX 3 | |
| Rebuild | once | every time | few times | few times | 3798.5 | (216.3) | 232.5 | (13.8) |
| EDFI pass | never | never | few times | never | 670.0 | (15.3) | 174.0 | (4.5) |
| HSFI passes | never | never | never | few times | 580.6 | (12.7) | 214.7 | (4.9) |
| Patch application | never | never | never | every time | 17.3 | (0.1) | 0.0 | (0.0) |

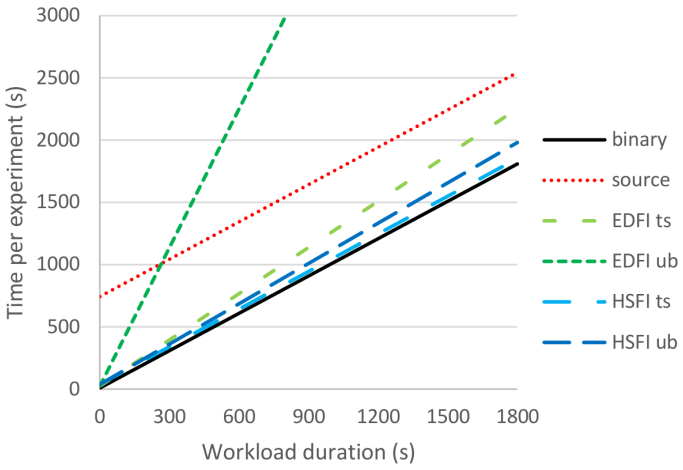


Fig. 10. Time taken per experiment depending on the workload duration for Linux; ts=test set, ub=Unixbench

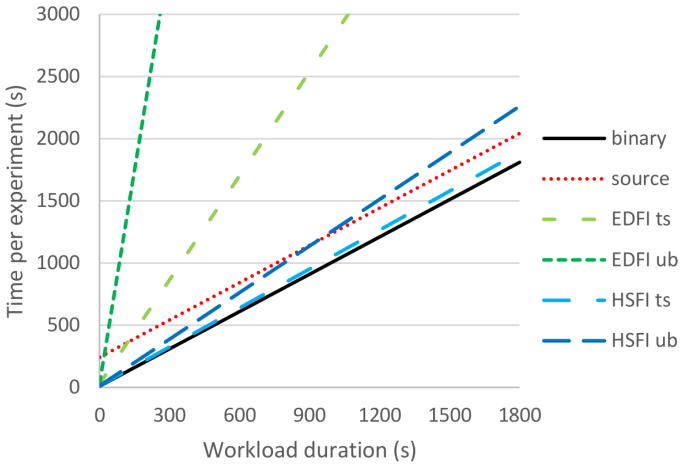


Fig. 11. Time taken per experiment depending on the workload duration for MINIX 3; ts=test set, ub=Unixbench

### C. Marker recognition

To determine whether our approach yields false positives and false negatives, our tool reports any unexpected code sequences involving any references to the marker variable. We have also cross-referenced the patches, the map files and the output from the *objdump* utility on the binary. We have not found any instances of false positives; every reference to the marker variable was indeed a valid marker. In case of Linux we found that out of the 313028 basic blocks for which markers were inserted, 125 were not found by our binary pass. Out of these, 93 do not seem to be present in the binary at all and were presumably eliminated by the optimizer. The remaining 32 were present but not found, so those should be considered false negatives. In all those cases, this seems to be because Dyninst (used for the binary pass) did not recognize some instructions included in inline assembly, causing it to produce an incomplete control-flow graph. The result is a false negative rate of just 0.01 % for Linux. In case of MINIX 3, all compiled modules together contain 219261 basic blocks with markers (with some occurring multiple times due to static libraries). In 6602 cases, the marker was not found. We found that just 25 of these are actually false negatives, the remainder not ending up in the binaries. All 25 false negatives were in the kernel module, which contains 5706 basic blocks. Like for Linux, the false negatives were found near assembly instructions not recognized by Dyninst. The false negative rate is 0.01 % for all of MINIX 3 and 0.38 % for the kernel. Given that we have found no false positives and very few false negatives, we conclude that our marker mechanism is very effective in mapping binary code to the original basic blocks.

### D. Threats to validity

There are several factors that might influence the results presented here. First, we measured performance when running on a virtual machine, namely QEMU with KVM. This is a deliberate choice because fault injection experiments on operating systems are likely to be performed in virtual machines in practice because this makes the experiments much easier to set up. Times measured in inside the virtual machine may not be as accurate as on the host machine. For this reason we implemented a hypercall that causes the current status including timestamp to be logged on the host. We have used these timestamps to get our performance results. The only exception is CPU usage, which has been measured on the guest. Any inaccuracies here would not affect the comparison. A second threat is the fact that we only measured performance without enabling any faults. If a fault is hit, it could cause the system to crash, cutting the test short, or hang, causing the test to last until some set timeout. We decided not to consider this as the behavior depends strongly on the exact fault model, something which is out of scope for this paper. A third threat is the fact that we have not measured any actual binary-level or source-level fault injection tools. However, we made an optimistic estimate which means that they should not be able to perform any faster than what we measured. As a result, the conclusions would either remain unchanged or become even more favorable to our proposed approach. Finally, it is possible that we missed false negatives in cases where basic blocks were duplicated by the optimizer, for example due to inlining or loop unrolling, and at least one instance was detected. Given the very low number of false negatives overall, it is unlikely this has happened on a substantial scale.

## VI. LIMITATIONS

The evaluation confirms that our new approach is able to perform fault injection with the benefits of source-level fault injection at a performance close to that of binary-level fault

injection. That said, some limitations of this approach still need to be considered. We first consider limitations fundamental to the design and then limitations of our implementation.

One fundamental limitation that we share with other source-based approaches is that the source code of the target program must be available. If no source code is available, a binary-level approach must be used. A second design limitation is the fact that the system being tested is not identical to the system used in production. This limitation is shared with other software mutation approaches, although we make more changes per binary. However, there are important use cases where the modified program is not the system whose reliability is being tested, such as changing one part of the OS outside the trusted code base (TCB) to determine whether the TCB itself can deal with the results properly. One final design limitation is the risk of false positives and negatives while detecting markers in the binary pass. We have not found any false positives in practice, nor does it seem likely that the exact same code sequence (including global variable address and valid basic block number) could appear from other sources. False negatives do occur and could result in fault candidates not being injected as the default without patch is to execute the pristine version. This could cause a deviation from the fault model, but the very low number of false negatives we found in practice suggests the effect should not be statistically relevant.

The remaining limitations are due to implementation choices and can be solved without changing the design. Our implementation requires the target program to be ported to LLVM with IR-based linking. Fortunately, Clang mostly provides a drop-in replacement for the more widely used GCC and GNU Gold allows linking to be performed on IR files in the same way machine code object files are normally linked. Often the configure script is versatile enough to allow LLVM with IR linking to be used, but in other cases small build system changes may be required. If the design were implemented working directly on the source code, this step would not be needed. Another limitation is the fact that, due to the use of the intermediate representation, some information from the source code does get lost. However this mostly affects preprocessor macros and demacrofication [37] allows those to be replaced by function calls, which are accurately represented in the IR. One more implementation limitation is that only a single fault can be selected for each basic block. This can be solved by creating more clones of the basic blocks to be able to select one of several combinations of faults. The drawback would be that the code size of the basic block would grow exponentially in the number of faults injected. The number of rebuilds needed has been shown to be very low even when selecting just one fault candidate per basic block at a time, so such a change is unlikely to be worthwhile unless the fault model requires it. Another limitation is the fact that we do not consider which mutations are equivalent. This could be addressed by applying a Trivial Compiler Equivalence compiler pass [45]. Finally, it is important to mention that we deliberately do not consider whether the injected faults are representative of real-world faults. Our implementation has been written to be easily adaptable to different fault models, which allows a representative fault model from the literature to be selected based on the requirements for the experiment.

## VII. Related work

In this section we will consider the use of various software fault injection implementations in research, papers assessing the representativeness of fault injection, and finally other research about improving the performance of fault injection. We do not repeat the classification of the various types of fault injection (which has been discussed in section II) and will instead focus specifically on software fault injection, which is the approach used in this paper.

### A. Use of software fault injection

Software fault injection is widely used in the research literature for many different purposes. For binary-level fault injection, G-SWFIT [31] stands out for being reused for many purposes. It has been used, among others, to compare binary-level injection against source-level fault injection [7] and interface-based faults models [5], for failure prediction [46], and to evaluate suitable fault loads for large systems [15]. Another tool has been produced to quantify and improve fault tolerance for the RIO file cache [23], which has been reused to study impact of faults on Linux kernel [17] and the isolation properties of MINIX 3 device drivers [20]. Duraes et al. [19] have implemented this approach to evaluate the impact of faults in drivers on system. A different approach to binary-level fault injection is found in XEMU [47], which uses QEMU to perform binary rewriting to inject faults.

As for source-level software fault injection, Jia and Harman [8] provide a literature review which shows how the use of this technique has greatly increased over time. Recently SAFE [32] has gained popularity and has been reused for certification of software [22], to evaluate the relationship between fault types and software metrics [48], to compare monitoring techniques [49], for evaluating whether Linux crashes in response to faults [18], and for evaluating the impact of running multiple fault injection experiments simultaneously [16]. SCEMIT [33] is notable for being implemented as a compiler pass rather than modifying the source code directly. EDFI [36] is implemented deeper in the compiler, performing fault injection on the intermediate representation after linking as we do. This still provides most benefits of source-level injection while allowing for easy build system integration and access to the entire program. This tool has been reused to determine how common silent failures are [21], to measure the impact fault load distortion [40], and to compare operating system stability [10].

Both source-level and binary-level approaches are widely used to improve software reliability. Our design provides a hybrid of the two, offering access to the source code or intermediate representation when injecting the faults, while achieving performance close to that of binary-level approaches through a binary-level pass selecting which of the injected faults to enable.

### B. Fault representativeness and accuracy

There are several papers providing an overview of representative fault types based on bugs found in real software [31], [34], [50]. Natella et al. [32] consider which fault locations are most representative of residual faults that are likely to remain in the software in production systems. These works

are orthogonal to our approach, which applies equally to any fault model based on software mutations. Our implementation allows the desired fault model to be implemented quickly, using the code rewriting abilities of the LLVM framework.

To apply a representative fault model, it is necessary to accurately perform the mutations as intended. Several papers compare the accuracy of different fault models or injection techniques. Of particular importance is Cotroneo et al. [7], who show that source-level fault injection is more representative of real software faults than binary-level injection. Lanzaro et al. [6] and Moraes et al. [5] show that interface-level faults do not accurately mimic software faults. [4] shows that hardware faults poorly represent software faults. Daran et al. [2] and Andrews et al. [3] perform source-level fault injection and show that their approach is representative of real faults. These results supports the case for preferring source-level fault injection if real software faults are to be emulated.

The influence of other factors on fault representativeness has also been considered. Winter et al. [11] argue that injection of multiple simultaneous faults is a more realistic fault model than the traditional approach of injecting just one fault at a time. Our design allows for arbitrary combinations of faults, while our implementation allows almost all combinations. Kikuchi et al. [18] and Van der Kouwe et al. [40] show that using an improper workload leads to poor fault representativeness. As our evaluation shows, our approach is suitable for both high-coverage and stress-testing workloads and beneficial for a wide range of workload durations, allowing flexibility to select the most suitable workload.

### C. Fault injection performance

Many previous papers have worked on improving the performance of fault injection. One common theme is speeding up fault space exploration to try the most relevant failures first. Banabic et al. [12] use fitness-guided exploration to search for high-impact faults that impact system recovery code. FATE [13] searches the most different failure scenarios first. Prefail [14] prunes the search space by finding which (combination of) faults is most promising to uncover bugs. Li et al. [51] use static analysis to determine which faults are most likely to cause long latency crashes. Although these papers are based on different fault models than ours (interface-level and hardware faults), variations that deal with software faults would be complementary to our approach and could help to speed up fault injection experiments even further.

There has also been some work that, like ours, proposes ways to speed up the experiments themselves. Winter et al. [16] investigate parallel fault injection, performing multiple experiments simultaneously on the same machine. This approach is complementary to ours. Van der Kouwe et al. [10] use EDFI to inject multiple faults at compile time and select a single one at run-time. This approach is closest to ours, but it introduces considerable run-time overhead because the decision to inject must be made at run-time. Our proposed solution performs better in all cases.

## VIII. Conclusion

Although source-level software fault injection is a good technique to inject faults that accurately conform to the fault model, it suffers from poor scalability to large code bases. In this paper, we presented a design that solves the scalability issue by reducing the number of times the target program needs to be rebuilt. We also built an implementation of this design, which can easily be adjusted for different fault models and different programming languages, which has most source-level information available to the fault injection logic, and which can inject software faults fully automatically. Although porting the binary pass to different architectures would require some effort, this effort is greatly reduced due to its simplicity and the fact that the binary pass is not involved in the actual fault injection. Although our approach incurs run-time overhead compared to source-level fault injection, our evaluation shows that this overhead is reasonable even for stress-testing workloads. As a consequence, the total time needed per experiment is considerably lower for large code bases, which means that statistically sound fault injection results can be reached at lower cost. Our approach also outperforms another solution for source-level injection with reduced rebuild overhead by a wide margin due to a much lower run-time overhead. We conclude that our solution retains the benefits of source-level fault injection while achieving performance close to that of binary-level approaches, providing the best of both worlds.

We have made the source code for our prototype available at https://github.com/vusec/dsn-2016-hsfi.

## IX. Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This work was supported by the Netherlands Organisation for Scientific Research through the NWO 639.023.309 VICI “Dowsing” project.

## References

[1] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, “Have things changed now?: an empirical study of bug characteristics in modern open source software,” in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 2006, pp. 25–33.

[2] M. Daran and P. Thévenod-Fosse, “Software error analysis: a real case study involving real faults and mutations,” in *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3. ACM, 1996, pp. 158–171.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 402–411.

[4] H. Madeira, D. Costa, and M. Vieira, “On the emulation of software faults by software fault injection,” in *Proc. of the Int’l Conf. on Dependable Systems and Networks*, 2000, pp. 417–426.

[5] R. Moraes, R. Barbosa, J. Duraes, N. Mendes, E. Martins, and H. Madeira, “Injection of faults at component interfaces and inside the component code: are they equivalent?” in *Proc. of the 6th European Dependable Computing Conf.*, 2006, pp. 53–64.

[6] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, “An empirical study of injected versus actual interface errors,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 397–408.

[7] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, “Experimental analysis of binary-level software fault injection in complex software,” in *Proc. of the 9th European Dependable Computing Conf.*, May 2012, pp. 162–172.

[8] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649–678, 2011.

[9] C. Byoungju and A. P. Mathur, “High-performance mutation testing,” *Journal of Systems and Software*, vol. 20, no. 2, pp. 135–152, 1993.

[10] E. Van der Kouwe, C. Giuffrida, R. Ghitulete, and A. S. Tanenbaum, "A methodology to efficiently compare operating system stability," in *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*. IEEE, 2015, pp. 93–100.

[11] S. Winter, M. Tretter, B. Sattler, and N. Suri, "simfi: From single to simultaneous software fault injections," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE, 2013, pp. 1–12.

[12] R. Banabic and G. Candea, "Fast black-box testing of system recovery code," in *Proc. of the 7th ACM European Conf. on Computer Systems*, 2012, pp. 281–294.

[13] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A framework for cloud recovery testing," in *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation*, 2011, pp. 18–18.

[14] P. Joshi, H. S. Gunawi, and K. Sen, "PREFAIL: A programmable tool for multiple-failure injection," in *Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications*, vol. 46, 2011, pp. 171–188.

[15] P. Costa, J. G. Silva, and H. Madeira, "Practical and representative faultloads for large-scale software systems," *Journal of Systems and Software*, vol. 103, pp. 182–197, 2015.

[16] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, "No pain, no gain? the utility of parallel fault injections," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015, pp. 494–505.

[17] T. Yoshimura, H. Yamada, and K. Kono, "Is linux kernel oops useful or not?" in *Proc. of the Eighth Workshop on Hot Topics in System Dependability*, 2012, p. 2.

[18] N. Kikuchi, T. Yoshimura, R. Sakuma, and K. Kono, "Do injected faults cause real failures? a case study of linux," in *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 174–179.

[19] J. Duraes and H. Madeira, "Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation," in *Proc. of the Pacific Rim Int'l Symp. on Dependable Computing*, 2002, p. 201.

[20] J. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum, "Fault isolation for device drivers," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pp. 33–42.

[21] E. Van der Kouwe, C. Giuffrida, and A. S. Tanenbaum, "On the soundness of silence: Investigating silent failures using fault injection experiments," in *Dependable Computing Conference (EDCC), 2014 Tenth European*. IEEE, 2014, pp. 118–129.

[22] D. Cotroneo and R. Natella, "Fault injection for software certification," *Security & Privacy, IEEE*, vol. 11, no. 4, pp. 38–45, 2013.

[23] W. T. Ng and P. M. Chen, "The systematic improvement of fault tolerance in the rio file cache," in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*. IEEE, 1999, pp. 76–83.

[24] P. Marinescu and G. Candea, "LFI: A practical and general library-level fault injector," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, Jul. 2009, pp. 379–388.

[25] J. Arlat, Y. Crouzet, and J.-C. Laprie, "Fault injection for dependability validation of fault-tolerant computing systems," in *Proc. of the 19th Int'l Symp. on Fault-Tolerant Computing*, 1989, pp. 348–355.

[26] W.-l. Kao, R. K. Iyer, and D. Tang, "FINE: a fault injection and monitoring environment for tracing the UNIX system behavior under faults," *IEEE Trans. Softw. Eng.*, vol. 19, no. 11, pp. 1105–1118, 1993.

[27] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 125–136, Feb. 1998.

[28] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "Llfi: An intermediate code-level fault injection tool for hardware faults," in *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 11–16.

[29] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 622–629.

[30] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A flexible software-based fault and error injection system," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 248–260, 1995.

[31] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, 2006.

[32] R. Natella, D. Cotroneo, J. Duraes, H. S. Madeira *et al.*, "On fault representativeness of software fault injection," *Software Engineering, IEEE Transactions on*, vol. 39, no. 1, pp. 80–96, 2013.

[33] P. Lisherness and K.-T. Cheng, "Scemit: a systemc error and mutation injection tool," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*. IEEE, 2010, pp. 228–233.

[34] B. Kidwell, J. H. Hayes, and A. P. Nikora, "Toward extended change types for analyzing software faults," in *Quality Software (QSIC), 2014 14th International Conference on*. IEEE, 2014, pp. 202–211.

[35] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of the Int'l Symp. on Code Generation and Optimization*, 2004, p. 75.

[36] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "EDFI: A dependable fault injection tool for dependability benchmarking experiments," in *Proc. of the Pacific Rim Int'l Symp. on Dependable Computing*, 2013.

[37] A. Kumar, A. Sutton, and B. Stroustrup, "Rejuvenating C++ programs through demacrofication," in *Proc. of the 28th IEEE Int'l Conf. on Software Maintenance*, 2012.

[38] "LLVM linux," http://llvm.linuxfoundation.org/index.php/Main_Page.

[39] "byte-unixbench," https://github.com/kdlucas/byte-unixbench.

[40] E. Van der Kouwe, C. Giuffrida, and A. S. Tanenbaum, "Finding fault with fault injection: an empirical exploration of distortion in fault injection experiments," *Software Quality Journal*, pp. 1–30, 2014.

[41] "Qemu - open source processor emulator," http://wiki.qemu.org/Main_Page.

[42] "cloc - count lines of code," https://github.com/AlDanial/cloc.

[43] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[44] S. Asmussen and P. W. Glynn, *Stochastic simulation: Algorithms and analysis*. Springer Science & Business Media, 2007, vol. 57.

[45] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 936–946.

[46] I. Irrera, M. Vieira, and J. Duraes, "Adaptive failure prediction for computer systems: A framework and a case study," in *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*. IEEE, 2015, pp. 142–149.

[47] M. Becker, D. Baldin, C. Kuznik, M. M. Joy, T. Xie, and W. Mueller, "Xemu: an efficient qemu based binary mutation testing framework for embedded software," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2012, pp. 33–42.

[48] D. Cotroneo, R. Pietrantuono, and S. Russo, "Testing techniques selection based on odc fault types and software metrics," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1613–1637, 2013.

[49] M. Cinque, D. Cotroneo, R. Della Corte, and A. Pecchia, "Assessing direct monitoring techniques to analyze failures of critical industrial systems," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE, 2014, pp. 212–222.

[50] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults based on field data," in *Proc. of the 26th Int'l Symp. on Fault-Tolerant Computing*, 1996, p. 304.

[51] G. Li, Q. Lu, and K. Pattabiraman, "Fine-grained characterization of faults causing long latency crashes in programs," 2015.