Binary Mutation Testing Through Dynamic Translation

Markus Becker, Christoph Kuznik, Mabel Mary Joy, Tao Xie and Wolfgang Mueller‡

Faculty of Electrical Engineering, Computer Science and Mathematics
University of Paderborn/C-LAB, D-33102 Paderborn, Germany
{beckerm, kuznik, mabeljoy, tao, wolfgang}@c-lab.de

Abstract—This paper presents a novel mutation based testing method through binary mutation. For this, a table of mutants is derived by control flow analysis of a disassembled binary under test. Mutations are injected at runtime by dynamic translation. Thus, our approach neither relies on source code nor a certain compiler. As instrumentation is avoided, testing results correspond to the original binary. In addition to high-level language faults, the proposed approach captures target specific faults related to compiling and linking. We investigated the software of an automotive case study. For this, a taxonomy of mutation operators for the ARM instruction set is proposed. Our experimental results prove 100% accuracy w.r.t. confidence metrics provided by conventional testing methods while avoiding significant mutant compilation overhead. Further speed up is achieved by an efficient binary mutation testing framework that relies on extending the open source software emulator QEMU.

Keywords—Embedded software verification; software emulation; fault-based testing; mutation analysis; test confidence;

I. INTRODUCTION

Today’s embedded systems software has become increasingly complex. Thus, advanced verification methods are mandatory in order to maintain a high level of dependability, while managing costs at the same time. Though, considerable progress has been made in the field of formal verification throughout the recent years, testing is still a crucial building block in order to achieve a high degree of reliability as it scales well with increasing complexity. However, as exhaustive testing (i.e., testing of all possible test cases) is commonly not applicable, the significance of testing results depends on the pertinence of the used set of test cases. Here, mutation based testing methods have proven to be valuable in order to answer the testing inherent philosophical question „Who guards the guardians?” by means of providing an objective measure for the test confidence.

Mutation based testing measures the quality of a set of test cases in terms of assessing the ability to disclose software faults. For this, mutations are injected into the software under test. The faulty software (referred to as mutant) is executed with the provided test cases. A mutant is considered as being killed when the mutation is getting propagated to the outputs. The quality metric for the given test cases is defined by the number of killed mutants w.r.t. the total number of mutants.

‡We declare that the material presented in this paper has been cleared through the affiliations of the author(s).

Mutation testing is commonly applied to high-level languages (e.g., Java, C/C++) by instrumenting the source code. This leads to a set of drawbacks which decreases the acceptance of mutation testing in practical design environments. First, it requires the availability of source code which is sometimes not accessible due to intellectual property issues. For instance, when commercial off-the-shelf software (e.g., software modules or libraries) provided by a supplier is integrated as object/binary code. Second, as the source code needs to be instrumented testing results do not rely on the original software shipped with the product. Moreover, mutants are typically derived either by compilation of the instrumented source code or through a special compiler. The former approach results in additional compilation overhead as each mutant has to be compiled individually. The latter approach requires a modification to the compiler.

Thus, we propose the novel approach of binary mutation testing. Here, mutants are derived from the original binary under test by applying binary analysis prior to its execution. Our approach does neither suppose the availability of source code nor the use of a certain compiler. By applying mutations to binary code we are able to capture faults specific to a certain target instruction set architecture and tool chain (e.g., induced through compiling or linking). We also introduce an efficient framework for binary mutation testing of ARM code by means of extending QEMU: an open source software emulator based on dynamic translation. Here, we propose a set of speed up extensions in order to further reduce mutant testing overhead.

For the evaluation of the proposed approach an industrial case study from the automotive domain has been investigated. We considered the accuracy and performance of deriving test confidence measures in terms of comparing binary mutation testing to conventional source code mutation testing. Evaluations were made on different sets of test cases created by two test case generators provided with the case study: a fault tolerant fuel injection control system. By employing two typical mutation operators we were able to prove that binary mutation testing reaches 100% accuracy w.r.t. the metrics derived by the source code mutation approach.

Concerning performance we were able to reach significant speed up, though our framework relies on the execution of non-native code. Here, the avoidance of individual mutant compilation and the use of binary translation result in
remarkable acceleration, which is close to native speed. Further speed up is achieved by online mutant detection, mutant skipping through coverage pre-analysis, and multicore utilization. For the given case study, the break even for the QEMU approach w.r.t. native execution is below 100,000-1,000,000 test cases per mutant. Moreover, we compared the QEMU framework to a different approach based on the conventional instruction set simulator GDB/ARMulator that comes with the gcc tool chain for ARM. Here, the QEMU testing speed up is approximately 100x-1000x in average.

The remaining paper is organized as follows. Section II gives a brief introduction on fundamentals. Section III describes the binary mutation testing approach, its application to the ARM instruction set architecture, and the binary translation based testing framework. Section IV provides the accuracy and performance evaluation based on the fuel injection controller case study. In Section V we describe related research work on the fields of mutation testing. Finally, in Section VI we conclude before giving a short outlook to our future work in Section VII.

II. FUNDAMENTALS

In this section a brief introduction is given to the general mutation testing approach, the ARM instruction set architecture, and the binary translation approach implemented by QEMU.

A. Mutation Based Testing

Mutation testing measures the quality of test data by their ability to reveal faults which are deliberately seeded into the software program under test. The possible fault injections are modeled by a set of mutation operators. Each mutation operator represents a type of syntactic change on programs such as replacing a plus with a minus or replacing a condition expression with true:

\[ c = a + b; \rightarrow c = a - b; \]
\[ if(a < b)... \rightarrow if(true)... \]

A mutant is generated by applying one mutation operator, i.e., fault injection, to a single place of a program copy. A large data base of mutants can be obtained when we apply each mutation operator to different possible locations of the program. To assess the test data, each mutant from the data base is separately executed with the tests and its outputs are compared with the executions of the original program. If under any test case the mutant produces a different output from that of the original program, it is said to be killed by the test data. Basically, the overall percentage of killed mutants is then used as an objective quality metric for the testing process.

Originally, mutation operators are intended to represent typical errors that a programmer can make. These errors are modeled as simple syntactic errors in mutation testing because of the Competent Programmer Hypothesis [1], which assumes that programmers are competent and write programs that are nearly to be correct. Further, another hypothesis called Coupling Effect [1] tries to establish the effectiveness of mutation testing. It states that, if a set of test data is able to kill most of the mutants that are generated with simple fault seeding, they will also be capable of revealing the real, more complex bugs of the program. In other words, simple faults are coupled with complex bugs with respect to the testing. This should give us the confidence, when a testing process reaches a high score under the mutation testing metric.

One of the biggest challenges with mutation testing is its high computational cost, which comprises both the generation of mutants and the execution to measure whether they can be killed under tests. Schema-based mutation [2] is a practical technique developed to reduce the effort of mutant generation. Instead of creating each mutant as an individual program, mutation schema codes all mutants into one copy of the program. Fault injections by mutation operators are included to this single object at its various locations, as a meta-mutant. Still, only one mutation will be made active each time by selecting a unique mutant ID from this meta-mutant program. The compilation also becomes a one-time job on the meta-mutant. Weak mutation is defined in [3] to alleviate the problem that we need to execute each mutant completely to see whether it can be killed under a test and we may have a large number of mutants and tests. The decision of killing is no longer only defined as comparison at the program output, but can also be allowed as comparing the program states immediately after the execution of the fault-injected expression with the original. Thus, complete mutant execution can be avoided, although this is a trade-off that indeed lowers the standard for test qualification.

B. ARM Instruction Set Architecture

The Advanced Risc Machine (ARM) architecture [4][5] can be found in many embedded 32-bit microcontrollers since late 1980s. It relies on the Reduced Instruction Set Computer (RISC) architecture which is optimized for high instruction throughputs at low power consumption. Since the first release of ARM instruction set (ARMv1) several extensions have been made. The current release (ARMv7) is used for instance in ARM Cortex A/M/R processor family.

Since ARM architecture is based on the load and store principles, instructions can only process data held in registers. There are 16 general purpose registers (R0-R15). The Program Counter (PC) is held in R15. R14 is used as the Link Register (LR) for efficient returning from subroutines. By convention the Stack Pointer (SP) is held in R13.

ARM instructions commonly take two, three, or four operands (e.g., source and destination registers Rs and Rd with optional operand registers Rm and Rn). They are broadly classified into five classes: data processing instructions, branch instructions, load-store instructions, software interrupt instructions and program status register instructions. Some instruction classes additionally make use of instruction flags.

Almost all ARM instructions can be executed conditionally, i.e., you can specify that the instruction is executed when its condition passes a given test according to the status
Table I

<table>
<thead>
<tr>
<th>CC</th>
<th>Condition</th>
<th>Flags test</th>
<th>CC</th>
<th>Condition</th>
<th>Flags test</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>=</td>
<td>Z set</td>
<td>NE</td>
<td>≠</td>
<td>Z clear</td>
</tr>
<tr>
<td>HS</td>
<td>≥*</td>
<td>C set</td>
<td>LO</td>
<td>&lt;*</td>
<td>C clear</td>
</tr>
<tr>
<td>MI</td>
<td>&lt; 0</td>
<td>N set</td>
<td>PL</td>
<td>≥ 0</td>
<td>N clear</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow</td>
<td>V set</td>
<td>VC</td>
<td>Overflow</td>
<td>V clear</td>
</tr>
<tr>
<td>HI</td>
<td>&gt;*</td>
<td>C set, Z clear</td>
<td>LS</td>
<td>≤*</td>
<td>C clear, Z set</td>
</tr>
<tr>
<td>GE</td>
<td>≥</td>
<td>N=V</td>
<td>LT</td>
<td>&lt;</td>
<td>N≠V</td>
</tr>
<tr>
<td>GI</td>
<td>&gt;</td>
<td>Z clear, N=V</td>
<td>LE</td>
<td>≤</td>
<td>Z set, N≠V</td>
</tr>
<tr>
<td>AL</td>
<td>Always</td>
<td>Flags ignored</td>
<td>NV</td>
<td>Never</td>
<td>Flags ignored</td>
</tr>
</tbody>
</table>

*unsigned

Table I

ARM CONDITION CODES.

register. By using conditional execution performance and code density can be increased. The conditional code is a two letter mnemonic appended to the instruction mnemonic. The default mnemonic is AL, or always execute. Conditional execution reduces the number of branches, which also reduces the number of pipeline flushes and thus improves the performance of the executed code. Conditional execution depends upon two components: conditional code and condition flags. The condition code is located in the instruction word, and the conditional flags negative (N), zero (Z), carry (C), and overflow (V) are held in the Current Program Status Register (CPSR). Condition flags can be updated through instructions by appending the respective instruction flag mnemonic (S). Table I lists the complete set of condition codes with the according flag testing being performed.

C. The QEMU Dynamic Binary Translator

Dynamic binary translation is used for efficient conversion of an Instruction Set Architecture (ISA) into a different ISA, e.g., when the ISA of an executable differs from its execution environment. Unlike static binary translation only code encountered at runtime is considered, thereby avoiding unnecessary translation overhead. This is sometimes also referred to as Just-In-Time (JIT) compilation technique. In contrast to instruction interpreting Instruction Set Simulators (ISS) code translation is performed on basic block level, i.e., linear code segments until a final branch instruction. Moreover, translated blocks (TB) are stored in a translation cache in order to provide execution speed close to native execution by avoiding redundant translation. Dynamic binary translation is often used for fast CPU emulation especially when there is no need for a detailed model of the CPU’s micro architecture.

QEMU [6][7] is an open source software emulator based on dynamic binary translation with support for different target and host platforms. Besides x86 many other architectures are supported, e.g., ARM, PowerPC, SPARC, MIPS or Microblaze. In general, QEMU can operate in two emulation modes: user mode and system mode. The user mode provides user space emulation for a single program on top of the Linux operating system. QEMU full system mode provides emulation of an entire target system including I/O in order to run a complete software stack, i.e., boot firmware, operating system, and kernel space device drivers.

The effort of porting QEMU to new target and host platforms is reduced through indirect translation by mapping front-end and back-end to an intermediate representation, i.e., a canonical set of micro operations. In QEMU this is achieved by a so called Tiny Code Generator (TCG).

The TCG accepts blocks of intermediate micro code (also referred to as TCG micro code) generated from the decoded target binary in order to rewrite blocks of native host code.

Table II shows the indirect translation of a simple basic block of ARM code (performing a subroutine call) into functionally equivalent x86 code. Each of the decoded ARM instruction invokes the generation of a set of simpler intermediate micro operations working on a virtual register set. The resulting block of micro code is subject to on-the-fly optimizations before it is rewritten as translated block of x86 code. The translation cache of QEMU maintains a hash table mapping an entry address of a target code block to the respective address of the translated block. Hence, in case a block is executed multiple times (e.g., with functions or loops) the major emulation effort of the nth execution is just jumping to the translated block’s entry address being look-up from the hash table. Additional speed up is achieved by chaining the execution of translated blocks without returning to the emulator’s main loop.

III. BINARY MUTATION TESTING

This section describes our binary mutation testing approach. Initially, a description of the general flow and its application to the ARM instruction set format is given. This is followed by a detailed description of our mutation testing framework based on the QEMU dynamic binary translator.

A. Binary Mutation Testing Flow

Fig. 1 depicts the testing flow which is composed of three major steps: binary analysis followed by mutation testing and evaluation. In contrast to source code mutation testing, a table of mutations is derived from the original binary by means of binary analysis (as we do not suppose the availability of source code). For this, the input binary is first disassembled and transformed into a control flow graph (CFG). Here, the disassembled basic blocks of the binary are represented by nodes connected through directed edges. An edge corresponds to the control flow between two basic blocks, e.g., through branches. Based on a further analysis of the CFG, the mutation table is generated describing binary...
mutations for creating a set of mutants from the original binary.

![Binary mutation testing flow](image)

In the mutation testing step binary mutants are created and tested by applying the mutation table entries separately. For each mutant, its output according to the given set of test cases is compared with the output of a golden run, i.e., a run of the original binary being carried out in advance. Finally, the evaluation step annotates the CFG with mutation testing results, i.e., instruction address coverage, mutation coverage, and mutation detection. This is done in order to extract metrics for the given set of test cases, e.g., the number of killed mutants w.r.t. the total number of mutants. Moreover, the annotated CFG may serve as input format for further investigations, e.g., for automatic test pattern generation (ATPG).

**B. Mutation Taxonomy for ARM Instruction Set**

According to Section II-B the general format of an ARM instruction word disassembly can be described as follows:

<operator> <condition> <flags> <operands>

The latter two fields are optional/mandatory depending on the class and type of instruction. Accordingly, we propose a set of atomic binary mutation operator classes affecting a single instruction word at a given address. Table III lists the mutant classes with their operator mnemonics that will be used throughout the remaining text.

1) **Operator**: The Operator mutator class (OPTR) summarizes all possible mutations of operators sharing the same format, i.e., number and type of operands and flags. This can be for instance OPTR(add→sub) in order to turn an arithmetic addition into a subtraction or it can be OPTR(and→or) in order to turn a logic and into a logic or.

2) **Condition**: The Condition mutator class (COND) summarizes all possible mutations of an instruction’s condition. As in ARM instruction almost all instructions can be executed conditionally this applies to nearly any instruction word. A typical mutator of this class is for instance COND(AL→NV) in order prevent instructions from being executed. Moreover, COND(*→AL) can be used in order to switch off the evaluation of any condition listed in Table I.

3) **Flag**: The Flag mutator class (FLAG) summarizes all possible mutations to operation flags. Obviously, this only applies to instruction classes taking flags. A useful mutator is for instance FLAG(S→¬S) in order to switch on/off an update of the condition flags in the CPSR register.

4) **Operand**: The Operand mutator class (OPRD) summarizes all possible mutations of operands. The applicability of a certain operand mutator to an instruction word depends on the number and type of operands used. Useful mutators of this class are for instance OPRD(Rd→Rs) in order to toggle source and destination register or OPRD(Opn→C) in order to change the immediate operand n to a constant value C.

In addition to the proposed ARM specific mutator classes, we need to introduce the general mutants DATA in order to change constant or variable data at a given address and ADDW in order to insert a new instruction word. Moreover, as some typical source code faults can only be covered at the binary level by performing mutations spanning over several addresses atomic mutations need to be extendable to complex mutations by means of applying a composition of atomic mutators to form a single mutant. This is commonly referred to as higher order mutation.

We chose this particular set of mutator classes for two reasons. First, they have an orthogonal characteristic in changing the meaning of an instruction word as they affect different aspects. Second, they are canonic in a way of being sufficient enough to perform any required mutation for testing the coverage of both high-level programming language faults (e.g., by the programmer) as well as low level target specific faults (e.g., compilation or binary linking).

Commonly, commercial off-the-shelf binary code is provided with symbol tables and header files in order to be linkable to some other code. As linkers cannot guarantee binary compatibility successful linking relies on the agreement on Application Binary Interfaces (ABI) specifying for instance the calling conventions for subroutines such as argument/return passing. ABIs are sometimes subject to changes (e.g., with gcc3/gcc4). Moreover, the usage of certain compiler flags can affect the binary compatibility.
int check_bounds(int a, int b, int c) {
    if (a<=b && b<=c) { /* Is b inside [a,c]? */
        return 1; /* True */
    }
    return 0; /* False */
}

Listing 1. Example function check_bounds in C language.

The ability to detect interface incompatibilities through a given set of test cases (referred to as interface mutation) can be investigated by our method at the binary level. Table IV summarizes binary interface issues related to linking of object/binary code and their coverage through binary mutation. Such kind of issues cannot be covered efficiently through source code mutation.

C. Binary Analysis

In order to identify appropriate locations for the application of the proposed mutation operators an analysis of the binary code is carried out prior to its execution. Let us consider a simple source code example (though source code is not required for the approach). Listing 1 shows the C function check_bounds that takes three integer arguments: a, b, and c. The purpose of that function is to check whether the value of b is inside an interval bounded by a and b in such a way that following condition is true: a ≤ b ≤ c (which is equivalent to a, b, and c being sorted in ascending order). If true the function returns 1. Otherwise it returns 0 (false).

The respective executable binary code is created using arm-elf-gcc cross compiler with -O0 flag (i.e., no code optimization). For the creation of a CFG, the binary code is disassembled starting with the function addresses extracted from the binary’s symbol table. Here only functions of user interest are taken into account. For each of the following instruction words the operation’s assembler mnemonic along with the operands are decoded from the instruction word to form linear segments of disassembled code until a branch occurs or a return statement is recognized. The resulting linear segments are stored in the CFG nodes.

In case of branch instructions the corresponding destination address is decoded in order to being disassembled in a recursive fashion. By means of coloring visited addresses redundant processing of instruction addresses is avoided, e.g., when branches join together. Unconditional branches are represented by a single directed edge pointing from the branch instruction’s source node to the corresponding branch target’s node. Conditional branches are represented by an additional directed edge according to control flow taken for the negative evaluation of the condition. In that case the edge points to a new node beginning with the instruction address directly after the respective branch instruction.

In case of a branch destination pointing to an internal address (i.e., not a linear segment’s start address) the node and its corresponding code segment is split at the branch target address. Thus, the resulting linear segments of the two new nodes are equal to the common definition of a compiler basic block, i.e., they are non-overlapping and there is only a single entry point and a single exit point. CFG construction is complete once all recursive disassembling paths have terminated due to recognition of a return statement or the encounter of a colored instruction address.

Fig. 2 depicts the corresponding CFG derived from the check_bounds function compiled into ARM code. The CFG is composed of five basic blocks. The first block (considered from up to down) contains the function prologue, i.e., saving the calling function’s register context to stack. Moreover, it evaluates the left part of the if statement’s condition in source line 2 (see Listing 1). In case the evaluation returns...
true the second block evaluates the right part of the condition (i.e., \( b \leq c \)) being conjuncted with a logic \textit{and}. Otherwise it is skipped. The 3rd and 4th blocks contain the assignment of the functions return value according to the evaluation of the \textit{if} statement. Finally, block five passes the return value and restores the calling function’s register context. This also includes restoring of the PC register from the saved LR thereby performing the return to the calling function.

For the derivation of a mutation table from the binary under test, the CFG is iterated in order to investigate the applicability of binary mutators based on the disassembled code segments stored in the nodes. Table V shows the portion of a mutation table derived from the \textit{check_bounds} CFG. The mutation table format is as follows. The first column denotes the mutation type. Here, \( A \) stands for an atomic mutation. Respectively, \( C \) stands for a complex mutation, i.e., the composition of multiple atomic mutations (spanning over multiple lines) in order to form a single mutant. Columns two to four contain the atomic mutator class, the concrete mutator, and the affected instruction word address (denoted by bold font style in the corresponding CFG in Fig. 2). Column five shows the equivalence in the affected source code line according to the binary mutation table entry.

Here, one can see that the mapping of a typical simple source code mutator (e.g., switching off the evaluation of an \textit{if} statement condition) might require the application of a complex mutation at the according binary code such as the mutation of the \textit{and conjuncted expression to }\textit{if }\textit{(true)}. It has to be mentioned that the previous introduced table is incomplete. Although the given example is very simple, an exhaustive application of the proposed mutator classes results in a much longer mutation table.

D. Binary Translation Induced Mutation

Dynamic binary translation provides a flexible and efficient execution environment for target specific binary code. It is widely used for the translation of non-native code into functionally equivalent native code at runtime. This way, the execution of an original binary on its target hardware can be emulated on hosts with a different ISA. By means of adapting the translator, the output, i.e., the resulting native code, can be instrumented in order to change the emulated behavior. We make use of this technique in order to emulate binary mutations through the translated code. Thus, there is no need for instrumenting the original binary itself. Moreover, translation induced mutation allows the application of more complex mutations. For instance, injecting code into binaries cannot be applied by a simple patch as this would need to shift instruction words which would invalidate the whole addressing. In such a case, it would be necessary to rewrite the complete binary.

For the mutation of the translated code we follow an instrumentation approach similar to the one introduced in [9]. In their work they describe a generic instrumentation interface for QEMU based on event-triggered plug-ins. The plug-in interface consists of a set of callback functions invoked at the occurrence of an event. Such events can be translation related or execution related. Callback functions assigned to translation related events can access the translator’s code generator API. Thus, they can suppress, extend, or modify the generation of translated code. Callback functions assigned to execution related events have access to the emulator’s runtime environment. Thus, they can trace or modify the state of the emulated CPU and memory.

Fig. 3 depicts the QEMU emulation flow extended by mutation injection. In QEMU the fetch-decode-execute cycle is realized by alternating translation and execution phases. A translation phase is entered when the emulated PC register encounters an unknown target address, i.e., when the look up of the corresponding Translated Block (TB) from translation buffer failed. The translation loop consists of fetching and decoding single instruction words from memory until the encounter of a branch instruction (denoting the end of a translation phase). Then, the content of the intermediate buffer is rewritten as native TB into the translation buffer. The TB’s entry address is stored with the target code PC entry address in a hash table.

We employ three generic translation events according to the following block translation phases: \textit{post-prolog}, \textit{pre-instruction}, and \textit{pre-epilog}. Such events can interrupt the translation process as depicted by Fig. 3 in order to modify the content of the intermediate code buffer. In case of a mutation affected address the encountered event is forwarded to the corresponding mutation plug-in through an event manager, which is aware of event sensitivities. Listing 2 shows the corresponding code of the extended QEMU function that
decodes ARM target code in order to generate intermediate code. Here, in lines 6, 9, and 11 translation events are fired. In order to replace valid code by mutated code the generation of valid code can be suppressed by plug-ins through a corresponding return code. The remainder outlines our approach by the example of an instrumentation plug-in for the COND mutator class for ARM binaries. However, as it can be easily seen, it can be similarly applied to any of the proposed ARM mutator classes.

A plug-in is compiled into a shared object that can be loaded at runtime by adding -mtplug-in my plug-in.so to the QEMU command line. Listing 3 is an excerpt of the COND mutator plug-in showing the pre-instruction event callback. In order to inject a condition code mutation into the intermediate code the original translation of the affected instruction address through the disas_arm_insn() function is replaced by a slightly different function called disas_arm_insn_cond(). This function additionally accepts an argument specifying the condition code to be used for mutation.

In QEMU ARM translator, conditional execution is supported by instrumenting the translated instruction with a preamble code performing the condition test and – if the condition test fails – a conditional branch to a label inserted just behind the translated instruction code. In order to generate the condition test the condition code is extracted from the four most significant bits of the instruction word. In contrast to that, the COND mutator plug-in uses the condition code provided through the currently selected mutation table entry (see line 12 in Listing 3).

Obviously, condition code mutations could be achieved more easily by patching the four most significant bits of the affected instruction word directly in the emulator’s memory. However, our approach is more powerful as it is not limited to mutations relying on patching of instruction words.

E. Efficient Mutation Testing

Since mutant sets can become very large when applying the full set of available mutators to a complex software we made several extensions to the QEMU user mode emulator in order to speed up binary mutation testing. For this, three major improvements were made concerning:

- Reduction of initialization and binary translation efforts
- Reduction of mutant execution and detection efforts
- Utilization of multicore hosts for parallelization

We comprise the golden run and all the subsequent mutant runs in a single QEMU invocation. By avoiding to restart the QEMU for each mutant we save the translator’s initialization phase and avoid redundant code translation. Moreover, by performing a coverage analysis prior to mutant testing we reduce the number of runs by skipping mutants that cannot be killed anyway due to lack of mutation coverage.

For this, several extensions to QEMU user mode emulator are required in order to extend the lifetime which usually ends with the program termination. First, we need to make a backup of the initialized CPU and memory state in order to reset QEMU efficiently. Since the emulator and the binary under test share a single host process, this just means to allocate a chunk of memory that is big enough hold a copy of the initialized memory regions. In order to minimize backup efforts, we just copy those memory areas that are affected during a test, i.e., the CPU context and the program’s data section. After a mutation test the QEMU translation cache contains mutated code. In order to avoid flushing the cache after each mutation test a list of affected translated blocks is maintained for deletion. Finally, we need to prevent QEMU from termination which is usually done by forwarding of the final exit syscall to the host OS which then kills the QEMU process. Thus, we trap the exit syscall in order to perform the reinit. Fig. 4 depicts the extended QEMU user mode emulator lifetime using efficient reinitialization.

The definition of strong mutation analysis states that a mutant is being killed when it is propagated to the design interfaces, i.e., resulting in a deviation of the mutant’s output and golden run’s output. Typically, relevant program output
is written directly or indirectly (i.e., via standard output) to
a dump file using printf() and fprintf() or it is written
to a device file using fwrite(). Under POSIX based OS
like Linux all output related standard library functions end
up with a write() syscall to a device handle. QEMU user
mode emulator, for instance, treats system calls by raising an
exception that lets QEMU return to the main loop after the
execution of the current translated block. In the main loop
the system call is trapped by forwarding it to the host’s OS
system call API.

We adopt this mechanism in two ways. First, during
the golden run we copy the data of all write() system
calls to an output buffer. This way the buffer is filled
with reference output data. As the amount of output data
can be really huge and is not known a priori the size
of the allocated buffer grows dynamically. Second, during
subsequent mutation tests the same mechanism is used to
compare a mutant’s output with the buffered reference data
in an online fashion, i.e., instantly when a write() system
call occurs. In case of the first deviated output character
the current mutant is marked as being killed and execution
stops immediately in order to reset QEMU and proceed with
the next mutant. Online mutant detection saves unnecessary
execution overhead. By suppressing the forwarding of the
actual syscall to the OS, we can also save costly context
switching and kernel time. Fig. 5 depicts the mutation testing
loop with online detection mechanism.

Besides output deviation a mutation can also lead to
program abortion when the emulator or executed program
enters a critical state, e.g., a segmentation fault or an illegal
instruction. In that case, we also trap exceptions in order to
avoid QEMU abortion and consider the current mutant as
being killed. Under certain circumstances, a mutation may
lead to an infinite loop. Infinite loop detection is hard when
there is no output generated in that loop. In that case, we
can only set a timeout w.r.t. the execution time of the golden
run. If the timeout is exceeded by a user defined factor the
host thread executing the current mutant is killed and the
mutant itself is considered as being killed.

As mutation testing is inherently parallel our testing
framework supports multicore hosts by means of distributing
the mutants execution on top of a set of worker threads.
In QEMU the translation cache is a global data structure
that is shared among multiple virtual CPUs. Now, since the
translation cache contains mutated code we need to keep sure
that mutants do not get corrupted by executing mutated code
from different mutants. In order to avoid additional thread
synchronization overhead we introduce a private translation
cache for each of the worker threads. For this, we make
use of the fork() system call in order to create copies of
the original QEMU process which is now referred to as
the master process. On success, the system proceeds the
execution of two identical processes which are both in state
of the master process (except their unique process IDs).

IV. EVALUATION

For the evaluation of our approach we consider the
accuracy of the generated test metrics and the testing per-
formance by investigating two test case generators provided
with an automotive case study. For this, we implemented two
reference tool chains for mutation testing based on native
compilation of instrumented source code and binary patching
for Instruction Set Simulation (ISS).

A. Case Study

Our case study is an embedded software of a fault-tolerant
fuel injection controller which is a part of a car motor man-
agement system. The software is internally composed of two
components: Sensor Correction and Fuel Rate Computation. The software requires four 16-bit integer sensor signals such as throttle angle or engine speed. The sensor correction component is able to compensate one signal fault at a time by use of approximation functions. Based on the corrected data the fuel rate computation component computes a 16-bit integer value for the fuel injection actuator.

The controller was modelled in MATLAB/Simulink. The software was generated by dSPACE TargetLink production code generator [10]. The C code consists of 10 functions with a total complexity of 3397 lines of code (LoC): 2387 LoC for the sensors component and 1010 LoC for fuel rate component. The target binary was compiled with arm-elf-gcc version 4.1.1 using -O0 flag (i.e., no code optimization). We use our method to assess the confidence of two test case generators: a generic delta generator and an engine model. The delta generator is a combinatorial approach that produces test cases by iterating integer input values with a pre-defined delta step. The delta can be any integer divisor of the input signal’s range. Thus, for our 16-bit input signals (each having a range from 0...65535) and a delta of 4096 (resulting in 16 steps per signal) the combination leads to 16^4 = 65536 different test cases. The engine model test case generator comes with the case study. Thus, it is more specific to the software as it provides a physical model of the engine. Test cases are generated in a closed-loop with the feedback of the controller’s output as depicted by Fig. 7. The engine model test case generator is set up by a virtual execution time. As the controller software is designed to run with a 10 ms period 15000 test cases correspond to the execution of 150s of virtual time. Moreover, certain error situations are stimulated by injecting sensor faults at predefined points in the simulated virtual time.

![sensor model](image)

**Figure 7.** Closed-loop testbed for the fault tolerant fuel injection controller case study.

### B. Reference Tool Chains

We compare our QEMU based framework against two different mutation testing tool chains: a native source code mutation tool chain based on instrumentation and compilation and a binary tool chain executing patched ARM code on a conventional ISS. The former tool chain is realized by a `sed` based source code instrumentation script. The script wraps preprocessor macros around C statements. This is done in order to switch on mutations separately through providing an according flag to the host compiler. The resulting executable runs natively on the host just like any other program.

The second tool chain is based on the GDB/ARMulator ISS that comes as part of GDB debugger provided with the gcc tool chain for ARM. GDB/ARMulator is a pure functional (i.e., not cycle accurate) simulator/emulator of a single ARM CPU running in user mode. In contrast to QEMU, ARMulator relies on a simple interpreter loop. Here, binary mutations are directly applied to the ARM executable prior to its execution (i.e., we did not apply any modifications to the GDB/ARMulator). For mutant detection standard outputs are piped to a dump file in order to be compared to a golden run output using the `diff` command.

### C. Metrics Accuracy

In order to assess the test confidence of the different test sets we consider three different metrics: instruction coverage, mutation coverage, and mutant detection (also referred to as mutant killing rate). Instruction coverage measures the percentage coverage of target instruction words reached by the test cases. Thus, it can only be provided by binary testing. Mutation coverage measures the amount of mutants that were reached through the control flow of the applied test cases. Mutant detection measures the percentage of mutants that were detected (or killed) in terms of propagating a mutation to the program outputs such as a deviation of the computed fuel rate. This is commonly referred to as strong mutation analysis. For accuracy comparison we consider two typical C mutation operators that were easily applied to all of the three tool chains: `if(false)` and `if(<cond>)->if(true)` and `if(<cond>)->if(false)`. For comparability of source code and binary approaches we identified matching mutations using the `addr2line` tool provided with the gcc binutils.

As the case study source code contains 115 `if`-statements this leads to a total number of 230 mutants by applying two mutators to each. Table VI shows the detailed metrics generated for selected sets of test cases. For each investigated function the different metrics are given by absolute and relative numbers. The metrics computed by the different approaches are identical. Thus, for the investigated set of mutants we reach 100% accuracy.

Fig. IV-B compares the metrics the two test case generators by increasing the number of test cases per mutant. The x-axis denotes the number of applied test cases per mutant. The y-axis shows the corresponding metric in percent. It turned out that the significant increase of metrics between test cases #5000 and #10000 for the engine model corresponds to the stimulation of a special situation (two signal faults at a time). The engine model performs slightly better in terms of providing sufficient confidence metrics with few test cases. With a small step size (e.g., 2048) the delta generator provides slightly better metrics as it generates 1048576 different test cases. Table VI shows that combining both test case generators pushes mutation coverage to 100% and mutant killing rate reaches 81%. Examining the residual mutants by hand turned out that the majority of the undetected mutants can be actually considered as equivalent mutants as they have no impact on the considered output.
D. Performance

Fig. 9(a) shows the performance numbers comparing the different testing approaches. The experiments were carried out on an Intel Xeon Quadcore HT processor running at 3.4 GHz. Here, the y-axis denotes the measured testing time in seconds. Again, the x-axis denotes the number of applied test cases per mutant. As each test case is applied to all mutants for the delta generator, 230 \* 1048576 = 241172480 test cases have to be investigated in total (assuming a step size of 2048). Basically, we can see that all mutation testing approaches scale well, i.e., almost linearly according to the number of test cases per mutant. As expected, source code mutation testing comes with a quite high base effort related to the number of mutants as each mutant is generated separately by compilation. Thus, the native approach is dominated by compilation efforts, i.e., testing time increases only slightly with the number of test cases per mutant. Fig. 9(a) shows that the break even for the GDB/ARMulator is only below 10-100 test cases per mutant. With QEMU the break even is below 100,000-1,000,000 test cases per mutant. In average, the QEMU approach performs 100-1000x faster than GDB/ARMulator. Fig. 9(c) demonstrates the speed up achieved by online detection and mutant skipping extensions applied to QEMU. Finally, Fig. 9(b) depicts the additional speed up that can be achieved by utilizing multicore hosts. With the proposed parallelization approach the testing effort can be almost divided by the number of available cores imposing no extra synchronization overhead. Here, four full cores with hyper threading are utilized.

V. RELATED WORK

Mutation testing has inherent higher execution costs, hence various mutant reduction and execution cost reduction techniques have been proposed [11]. Most of the existing approaches focus on white-box testing and source code instrumentation, so the source code or intermediate object code of the design-under-verification, such as Java bytecode in [12], has to be available for the generation of mutants. Moreover, most frameworks focus on high-level software programming languages such as C# and Java [13]. For example, a large set of C language mutation operators were introduced in [14]. Later it was showed in [15] that a reduced number of operators still achieves a high mutation score. For hardware design, SpringSoft
CERTITUDE supports functional qualification for C and VHDL/Verilog [16]. In [17] mutation operators for IP-XACT electronic component descriptions were introduced. In contrast to prior mentioned related work we aim to leverage mutation testing in the embedded software domain, which is mainly C and SystemC based, by doing so, also targeting mutation faults at the application binary interface (ABI) for COTS libraries [18]. In [19] the authors modified a GNU C compiler chain to generate patches in order to enable compiler-integrated mutant generation. In [20] the authors propose a SystemC error and mutation injection tool based on compiler injection via a plugin for the gcc compiler. In total, four mutant operators are introduced. Another approach for SystemC and TLM mutation testing [21] allows to selectively activate one mutant at a time through the use of a fault_number variable, properly driven by the testbench during the simulation phase. In contrast to the presented compiler-induced and super mutant techniques our proposed binary translation based approach allows to perform mutation testing for different ISAs and offers much greater mutation flexibility by means of the event-triggered callbacks mechanism during translation. Moreover, targeting COTS libraries (with no source code available) our approach is language and compiler independent. In [22] the authors propose a software fault injection technique for the IA32 platform by means of machine-code level patterns. Mutations are applied directly to the target code. As their focus is on the emulation of residual software they do not consider efficient fault injection/detection for mutation testing such as our binary translation based technique. Moreover, their proposed system does not provide cross platform support for typical embedded systems instruction set architectures.
such as ARM. Besides that, our framework provides the verification engineer with feedback on the achieved test confidence by means of a binary CFG annotated with results such as undetected mutations and their corresponding addr2line information.

VI. CONCLUSION

We introduced a novel approach for mutation testing of binary software. The testing is seamlessly integrated into binary translation cycle of software emulation runtime. Mutants are derived from the original software binary under test by a control flow analysis prior to its execution. Though we introduce our approach by mutation operators for the ARM instruction set, the basic principles are well applicable to other embedded processors. Our approach comes with several significant advantages: (i) it does not presume the availability of the source code nor does it require modifications of the applied target compiler; (ii) we can capture faults specific to a target instruction set architecture and tool chain, e.g., compiler bugs and anomalies in the code optimization.

Experiments were conducted with a case study from the automotive industry, a fault tolerant fuel injection control system. Our experiments reached a 100% accuracy compared to source code mutation testing at the same time providing a speed up of up to 100-1000x compared to the execution of GDB/ARMulator ISS. We can even outperform native execution as our approach avoids individual mutant compilation and testing relies on an efficient framework extending the QEMU binary translator. The utilization of multicore hosts through efficient multi-threading further improves testing speed with the number of available cores.

VII. FUTURE WORK

Our future work will focus on a more complete evaluation of the proposed mutator classes for ARM and their efficient application to testing low level faults such as ABI mismatches. Further work will consider automatic test pattern generation (ATPG) from binary CDFG analysis and porting the framework to QEMU system mode in order to cover also system software such as operating systems, device drivers, and boot firmware.

ACKNOWLEDGMENTS

This work was partly funded by the DFG SFB 614 and the German Ministry of Education and Research (BMBF) through the project SANITAS (01M3088I) and the ITEA2 projects VERDE (01S09012H) and TIMMO-2-USE (01IS10034A).

REFERENCES