XEMU: An Efficient QEMU Based Binary Mutation Testing Framework for Embedded Software

Markus Becker, Mabel Mary Joy, Tao Xie, Christoph Kuznik, Daniel Baldin, Wolfgang Mueller

1C-LAB, University of Paderborn, Fürstenallee 11, 33102 Paderborn, Germany {beckerm, kuznik, mabeljoy, tao, wolfgang}@c-lab.de
2Heinz Nixdorf Institute, University of Paderborn, Fürstenallee 11, 33102 Paderborn, Germany dbaldin@hni.upb.de

ABSTRACT
This paper presents the XEMU framework for mutation based testing of embedded software binaries. We apply an extension of the QEMU software emulator, which injects mutations at run-time by dynamic code translation without affecting the binary software under test. The injection is based on a mutation table, which is generated by control flow graph (CFG) analysis of the disassembled code prior to its execution without presuming access to source code. We introduce our approach by the example of the ARM instruction set architecture for which a mutation taxonomy is presented. In addition to extending the testing scope to target specific low level faults, XEMU addresses the reduction of the mutants creation, execution, and detection overheads. Moreover, we reduce testing efforts by applying binary CFG analysis and constraint-based test generation for improved test quality. The experimental results of a car motor management software show significant improvements over conventional source code based approaches while providing 100% accuracy in terms of the computed test quality metrics.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging—Testing tools (e.g., data generators, coverage testing); D.2.8 [Software Engineering]: Metrics—performance measures; D.2.4 [Software Engineering]: Software/Program Verification—Formal methods, Reliability; D.3.4 [Programming Languages]: Processors—Compilers, Optimization, Run-time environments

Keywords
Embedded systems, Software emulation, Just-in-Time compilation

1. INTRODUCTION
Embedded software development requires profound testing and validation of software artifacts before the final shipping of the product. This especially applies for areas like automotive, avionics, and health care with hard requirements for dependability, robustness and faultlessness are omnipresent in the software development cycle. Therefore, various verification techniques and methodologies, each targeting different aspects of the embedded software, have been introduced. In this context, mutation based testing methods are well established for the functional qualification of complex test benches to enhance embedded software quality. They measure the quality of test cases by means of identifying faults in the hardware model or software, respectively. As such, mutants in form of faulty software modifications are injected into the code of the system under test.

The fault injections are modeled by a set of mutation operators. Each mutation operator represents a type of a syntactic modification reflecting a coding error in the program such as:

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

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 compared to the output of the original program, it is considered to be killed by the test data. One of the biggest challenges with mutation testing is its high computation cost in generation, execution, and recognition of individual mutants, which is addressed by our approach.

In general, mutation based testing for software is applied to high-level languages (e.g., Fortran, Java, C/C++) by instrumenting the source code coming with the following drawbacks for practical application: (i) it requires the availability of source code which is sometimes not accessible; (ii) the instrumented source code is different from the final code and thus may give different results. Additionally, mutants are derived either by compilation of the instrumented source code or through a special compiler. Therefore, it either results in additional compilation overhead as each mutant has to be compiled individually or it requires the modification of the compiler.

We introduce a novel approach for mutation testing of binary software in conjunction with formal methods to enhance test set quality. The testing is seamlessly integrated at run-time into the binary translation cycle of the software emulation framework. Mutants are derived from the original software binary under test by control flow analysis prior to its execution. Though we introduce mutation operators by the example of the ARM instruction set [1] [20], the basic principles apply to other embedded processors as well.

As such, our approach does neither presume the availability of
the source code nor does it require modifications of the applied target compiler. By creating mutations at binary code level, we can also capture faults specific to different target instruction set architectures (ISA) and vendor-specific tool chain and cover compiler-specific effects like code optimization.

Though we introduce our approach by the example of QEMU [6][2], our approach applies to the general concepts of dynamic binary translation and Just-In-Time (JIT) compilation which are used by several software emulators for efficient run-time conversion of different ISAs, i.e., from guest to host ISA. In contrast to instruction interpreting instruction set simulators (ISS), dynamic code translation is typically performed on basic block level, i.e., linear code segments closing with a final branch instruction. Unlike static code translation only those blocks encountered at run-time are considered thereby avoiding unnecessary translation overhead. Moreover, basic blocks are translated into translated blocks (TB), which are stored in a translation buffer to avoid redundant translations at run-time and keeping the execution speed close to native execution. In the case of QEMU, the effort of porting to new target and host platforms is reduced by an intermediate code level, i.e., a canonical set of micro operations, which is then translated to native code by the so-called tiny code generator (TCG). For our evaluations, we applied and extended user mode QEMU. The user mode provides user space emulation for a single program on top of the Linux operating system (OS). However, the basic principles also apply to QEMU full system mode, which provides emulation of an entire target system including physical memory and I/O in order to run a complete software stack, i.e., boot firmware, operating system, and kernel space device drivers.

Our results are evaluated by a case study from the automotive industry, a fault tolerant fuel injection control system. Our experiments showed that binary mutation testing can generate and cover identical set of mutants compared to source code instrumentation. Therefore, focusing on control flow mutation we reached 100% accuracy compared to source code instrumentation with a GDB/ARMulator tool chain with a speed up of up to 100-1000x at the same time. Though our framework is based on the execution of non native binary code, we can even outperform the native (i.e., host-compiled) source code based approaches as we avoid significant compilation overhead. Moreover, by efficiently utilizing multicore hosts we further reduce testing efforts proportional to the number of available cores.

The remainder of this paper is organized as follows. Section 2 describes our binary mutation testing approach, its application to the ARM instruction set architecture, and the binary translation based testing framework. Section 3 presents the experimental results. Thereafter, related work is discussed before the final chapter closes with a conclusion.

2. BINARY MUTATION TESTING

Our approach is based on the XEMU binary mutation testing flow. We start with a description of the general flow and its application to the ARM instruction set format followed by a detailed description of our mutation testing framework based on the QEMU user mode emulator.

Our test flow (see Fig. 1) is composed of four major steps: binary analysis followed by automatic test pattern generation (ATPG), binary mutation testing and evaluation. In a first step, a table of mutations is derived from the original binary by static code analysis presuming the provision of relevant symbol information but no availability of source code. For this, the considered symbols of the input binary or object code are disassembled in order to construct an annotated control flow graph (CFG). Based on a further analysis of the annotated CFG, a mutation table is generated describing binary mutations for creating a set of mutants from the original binary. In the next step, advanced ATPG techniques are applied to the annotated CFG in order to provide pertinent test cases for reaching sufficient test quality.

In the mutation testing step binary mutants are created and tested by injecting mutations from the table separately and executing each mutant with all test cases. For each combination of mutant and test case, its output is compared with the output of a golden run, i.e., a first run of the original binary that is carried out in advance. Section 2.3 introduces our QEMU based testing framework that applies mutation testing efficiently at run-time. Finally, the evaluation step extends the annotated CFG with mutation testing report data, i.e., instruction address coverage, mutation coverage, and mutation detection (killed mutants). This is done in order to extract quality metrics for the applied set of test cases, e.g., the number of killed mutants w.r.t. the total number of mutants. In case the computed metrics do not meet the targeted level of test quality, steps two to four have to be repeated until a sufficient level is reached or the computed metrics converge to an upper bound.

2.1 Binary Analysis

For the derivation of a mutation table and the generation of test cases from the binary under test, we apply static analysis techniques. In order to construct an annotated CFG, we first disassemble the binary code to identify the static basic blocks and the control flow between these blocks of the program. In general, every program can be uniquely partitioned into a set of non-overlapping static basic blocks, i.e., blocks with a single entry and exit point. The analysis of binary code is a non-trivial task. Disassembling and interpreting binary files can be challenging due to the Code Discovery Problem as many ISAs allow binary data to be mixed up with executable instructions. Not distinguishing between instructions and data may invalidate the analysis process since control flows may not be discovered and data may be misinterpreted as instructions or vice versa. However, on ARM platforms this issue is addressed by the embedded applications binary interface (EABI) that specifies the provision of position information for data and instruction blocks by special mapping symbols inside the symbol table (see Section 4.6.5 in [5]).

In the next step the constructed CFG is annotated by static data flow analysis. For this, we use the common approach of forward substitution as described by Cifuentes et al. [8, 9] to derive complex
expressions from low level expressions, which are the assembler instructions of the binary in our case. For assembly code, one can express the contents of a register \( r \) in terms of a set \( \{ a_k \} \) at instruction \( i \) as \( r = f_1 (\{ a_k \}, i) \). If the definition at instruction \( i \) is the unique definition of a register \( r \) that reaches an instruction \( j \) along all paths in the program without any of the registers \( a_k \) being redefined, one can forward substitute the register definition at instruction \( j \) with \( s = f_2 (\{ r \}, j) \), resulting in:

\[
s = f_2 (\{ f_1 (\{ a_k \}, i) \}, j)
\]

Fig. 2 depicts an annotated CFG generated from the example C code function check_bounds (see Listing 1). We use expressions from forward substitution analysis to annotate the edges of the CFG with constraints that need to be fulfilled in order to be taken. With the binary analysis framework [23] we are able to map the high level information from header files such as the function interfaces to the binary CFG. Applying global data flow analysis techniques to the extracted information we can annotate parts of the CFG with high level constraints based on the input parameters of the binary objects. However, detecting access to high level data structures is not trivial as there can be an unlimited amount of access possibilities generated by the compiler. Thus, an expression normalization step is applied to allow a meaningful and usable annotation of the binary CFG.

In order to extract a table of mutants from the CFG, the applicability of instruction set specific mutation operators (mutators) to the individual basic blocks has to be investigated. ARM instructions commonly take two, three, or four operands, e.g., source and destination registers Rs and Rd with optional operand register 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., it can be specified that the instruction only executes if the condition code flags pass a given condition or test. By using conditional execution performance and code density can be increased. The conditional code flags of the CPSR register. 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 according instruction flag mnemonic (S).

In general, the pattern of an ARM instruction word is as follows:

```assembly
<operator> <condition> <flags> <operands>
```

where the latter two fields are optional/mandatory due to the individual instruction. According to that pattern we define a set of atomic mutator classes. Table 1 shows the main atomic mutator classes for ARM with their mnemonics and a concrete mutator example for each. The Operator mutator class (OPTR) covers 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. The Condition mutator class (COND) covers all possible mutations of an instruction’s condition, i.e., it applies to almost all ARM instructions. A typical mutator of this class is for instance COND(AL, NV), i.e., changing the condition from always to never in order to prevent instructions from being executed. The Flag mutator class (FLAG) covers all possible mutations to operation 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. The Operand mutator class (OPRD) covers all possible mutations of operands. Useful mutators of this class are for instance OPRD(Rd, Rs) for toggling source and destination register.

Additionally, we introduce the general DATA mutator in order to change constant or variable data at a given address and ADDW in order to insert a new instruction word. We chose this set of atomic mutator classes as they are orthogonal in changing different aspects of the instruction word. Moreover, by combining multiple atomic mutations we can efficiently cover any complex mutations such as source level faults or target specific faults, e.g., related to binary interfaces (see Table 2).

Table 3 shows a portion of the corresponding mutation table which is generated from the check_bounds CFG. The first column of that table gives the mutation type. Here, A stands for an atomic mutation and C stands for a complex mutation, i.e., the composition of multiple atomic mutations spanning over multiple lines. Columns two to four contain the atomic mutator class, the concrete mutator and the affected instruction word address (given by bold characters in the CFG of Fig. 2). Column five shows the equivalence in the affected source code line according to the binary mutation table entry. Row eight shows that the mapping of a typ-
2.2 Automatic Test Pattern Generation

We apply automatic test pattern generation (ATPG) by extracting new test cases from the annotated binary CFG using constraint satisfaction problem (CSP) solving. Our CSP/ATPG approach can be applied to both the generation of test cases from scratch and/or to improve existing test case sets with insufficient test quality such as a low percentage of killed mutants (mutant detection rate). As bad mutation detection is likely to be related to a lack of code and path coverage increasing, the coverage is applied as a heuristic for improving also the mutant detection. For this, we derive CSPs from the annotated CFG according to paths leading to a mutated basic block that has not been reached by the existing test cases. Let us consider basic block four of Fig. 2 to be unreached by the current set of test cases as an example. Thus, the mutant defined at address 0x48 cannot be killed as it does not impact the output. By path backtracking we find all paths and the constraints for each path by combining the edge constraints as logical conjunctions. For basic block four in Fig. 2, this leads to the following set of path constraints:

\[
\begin{align*}
&\quad a > b \\
&\quad (a \leq b) \land (b > c)
\end{align*}
\]

The basic block will be reached if any of the two constraints is fulfilled. Thus, for a reachability analysis the existence of a test case (i.e., a tuple of input values) that fulfills the following expression must be computed:

\[
\exists (a, b, c) : ((a > b) \lor (a \leq b) \land (b > c))
\]

By generating and solving the expressions for all unreached basic blocks, we then automatically create test cases, which fulfill all constraints to ensure that the path will be taken during test execution in order to increase the chance on killing the corresponding mutant. Our developed framework uses the STP Constraint Solver [11] to automatically derive the values of the variables for the test cases. In general, the constraints may contain subexpressions that may not be solvable or variables we may not be able to calibrate by the test environment. In this case we try to solve as many test case relevant subexpressions and use the solutions as the input parameters for the test cases. Although it may not be guaranteed that the path will be taken at run-time, we can show that the chance of it will be significantly increased.

As mutant killing does not just depend on the coverage of the block with the undetected mutant but also on the path to reach it, we need to compute test cases that cover as many paths as possible. As the number of paths to a single block can become very high, we apply a random approach that tries to solve the path constraints for n randomly chosen paths.

2.3 Run-Time Binary Mutation by QEMU

For efficient generation and testing of mutants, we induce mutations online during the execution of the original binary under investigation. For that we modified the dynamic binary translator of QEMU. Thus, there is no need for instrumenting the original binary itself. The induced mutation during the translation also allows the application of more complex mutations, which cannot be applied through simple patching of the binary file.

For the mutation of the translated code, we follow an instrumentation approach similar to [13] to make it easily portable to other target platforms supported by QEMU. They introduced a generic instrumentation interface for QEMU that is 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. A callback function assigned to a translation related event may access the translator’s code generator API in order to affect the emulation. Thus, it can suppress, extend, or modify the generation of translated code. Callback functions assigned to execution related events have access to the emulator’s run-time environment. Thus, they can trace or modify the state of the emulated CPU and memory. Plug-in code can be compiled into shared objects in order to be linked to the generic interface at run-time.

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

This process can be interrupted by our mutation extensions. For this, the encounter of a mutation affected address triggers the callback of a mutation plug-in. The remainder outlines our approach by the example of an instrumentation plug-in for the emulation of the COND mutant for ARM binaries. However, as it can be easily seen, it can be similarly applied to any of the proposed ARM mutator classes from Table 1.

In order to inject a condition code mutation into the translated code, the translation of the affected instruction address through the original translator function disas_arm_insn() is replaced by the slightly different function disas_arm_insn_cond() executed by

<table>
<thead>
<tr>
<th>Atomic mutator class</th>
<th>Mnemonic</th>
<th>Example mutator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change operator</td>
<td>OPTR</td>
<td>OPTR(add-&gt;sub)</td>
</tr>
<tr>
<td>Change condition</td>
<td>COND</td>
<td>COND(AL-&gt;NV)</td>
</tr>
<tr>
<td>Change flag</td>
<td>FLAG</td>
<td>FLAG(S-&gt;~S)</td>
</tr>
<tr>
<td>Change operand</td>
<td>OPRD</td>
<td>OPRD(Rd-&gt; Rs)</td>
</tr>
</tbody>
</table>

Table 1: Atomic binary mutator classes for the ARM instruction set.

<table>
<thead>
<tr>
<th>Symbol table/header</th>
<th>Branch, load and store addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subroutine arguments passing</td>
<td>Register/stack access</td>
</tr>
<tr>
<td>Subroutine return value passing</td>
<td>Register/stack access</td>
</tr>
</tbody>
</table>

Table 2: Coverage of binary interface related errors.

<table>
<thead>
<tr>
<th>Type</th>
<th>Class</th>
<th>Mutator</th>
<th>Addr.</th>
<th>Source code equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>OPRD</td>
<td>Rm→Rn</td>
<td>0x24</td>
<td>2: (f(f &lt; a)&amp;&amp;b &lt; c)</td>
</tr>
<tr>
<td>A</td>
<td>OPTR</td>
<td>sub→add</td>
<td>0x24</td>
<td>2: (f(a ≤ b)&amp;&amp;c &lt; e)</td>
</tr>
<tr>
<td>A</td>
<td>OPRD</td>
<td>Rm→Rn</td>
<td>0x34</td>
<td>2: (f(a ≤ b)&amp;&amp;c &lt; e)</td>
</tr>
<tr>
<td>A</td>
<td>OPTR</td>
<td>sub→add</td>
<td>0x34</td>
<td>2: (f(a ≤ b)&amp;&amp;c &lt; e)</td>
</tr>
<tr>
<td>A</td>
<td>FLAG</td>
<td>S→~S</td>
<td>0x34</td>
<td>2: (if(a &lt; b)</td>
</tr>
<tr>
<td>A</td>
<td>COND</td>
<td>MI→AL</td>
<td>0x28</td>
<td>2: (f(b &lt; e)</td>
</tr>
<tr>
<td>A</td>
<td>COND</td>
<td>MI→NV</td>
<td>0x28</td>
<td>2: (if(false)</td>
</tr>
<tr>
<td>C</td>
<td>COND</td>
<td>MI→AL</td>
<td>0x28</td>
<td>2: (true)</td>
</tr>
<tr>
<td>A</td>
<td>OPRD</td>
<td>Op2→0x1</td>
<td>0x48</td>
<td>3: return 1;</td>
</tr>
<tr>
<td>A</td>
<td>OPRD</td>
<td>Op2→0x0</td>
<td>0x5c</td>
<td>5: return 0;</td>
</tr>
</tbody>
</table>

Table 3: Binary mutation table.
the COND mutator plug-in. In contrast to the original function, it additionally accepts an argument specifying the condition code to be used for translation. In the QEMU ARM translator, conditional execution is supported by instrumenting the translated instruction with a preamble code performing the condition test and – for that case the condition test fails – a conditional branch to a label that is inserted just behind the translated instruction. In order to generate the condition test, the condition code is usually extracted from the four most significant bits of the instruction word. In contrast, the COND mutator plug-in uses the condition code argument provided through the currently selected mutation table entry. The suppression of the original code generation is then indicated by the mutator plug-in through a specific return code. 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.

#### 2.4 Efficient Mutant Execution and Detection

Mutant sets can become very large when applying the full set of mutators to complex software. Therefore, we introduce several extensions to the QEMU user mode emulator in order to speed up binary mutation testing. For this, three major improvements were made: (i) reduction of initialization and binary translation efforts, (ii) reduction of mutant execution and detection efforts, and the (iii) utilization of multicore hosts for parallelization.

For this, we combine the golden run and all the subsequent mutant runs in a single emulator invocation. As such, we avoid restarting the emulator for each mutant, so that we save the translator initialization and avoid redundant code translation as mutants do not largely differ. By performing a mutation coverage analysis already at the golden run we can also reduce the number of runs by skipping mutants that cannot be killed anyway due to a lack of coverage.

Several extensions to the QEMU user mode emulator are required in order to extend the lifetime, which usually ends with the executed program’s 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 we just need to allocate the amount of memory that is big enough to hold a copy of the initialized memory regions. In order to minimize backup efforts, we copy those memory areas that are affected during a test, i.e., the CPU context and the program’s data section. After a mutation run, the QEMU translation buffer contains mutated code. In order to avoid flushing the buffer after each mutation run, 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. For that, we trap the exit syscall in order to perform the reinit. Fig. 4 depicts the extended QEMU lifetime for executing multiple program runs in a loop with fast 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 the 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. The QEMU user mode emulator, for instance, treats system calls by raising an exception for returning QEMU to its main loop after the execution of the current TB. 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. During the golden run, we copy the data of all write() system calls to an output buffer storing the reference 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.

Then, the same mechanism is used during mutation run to compare a mutant’s output with the previously stored golden run data in an online fashion, i.e., instantly when a write() system call occurs. In case of the first deviating 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 actual syscall to be forwarded to the OS, we can also save costly context switching and kernel time.

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 detec-
tion is hard when there is no output generated in that loop. In that case, we can only set a timeout w.r.t. the golden run. If the timeout expired, the host thread executing the current mutant is killed and the mutant itself is considered as being killed. Fig. 5 depicts the extended QEMU lifetime with mutation testing loop and online detection.

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. The QEMU translation buffer is a global data structure that is shared among multiple virtual CPUs. Since the translation buffer contains mutated code, we need to be 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 buffer for each of the worker threads. For this, we make use of the fork() system call to create copies of the original QEMU process, which becomes the master process and acts like a watchdog process that kills and restarts worker threads being timed out due to infinite loops. By forking the master process directly before executing the mutant loop all data structures, e.g., CPU state, reference output buffer and the translation buffer, are in a ready-to-use state. This avoids redundant QEMU initialization and redundant golden run execution. By repeating the fork() system call \( n \) times, we create \( 2^n \) worker threads. Now, as processing of mutants has no interdependencies the synchronization overhead is negligible. The assignment of mutants to worker threads is achieved by a semaphore initialized to the total number of mutants. The worker threads update the global testing report via shared memory. After all worker threads have completed, the master process finalizes the metrics report. Fig. 6 depicts the forking of the QEMU master process for efficient multicore host utilization.

3. EXPERIMENTAL RESULTS

Our case study is based on the embedded software of a fault-tolerant fuel injection controller, which is a part of the car motor management system. The software is internally composed of two components: Sensor Correction and Fuel Rate Computation. The software requires four signed 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 sensor data the fuel rate computation component computes the fuel injection rate for the actuator.

The controller was originally modeled in MATLAB/Simulink where the software was automatically generated by the dSPACE TargetLink production code generator [3]. The generated C code consists of 10 functions with a total complexity of 3397 lines of code. The target binary was compiled with arm-elf-gcc version 4.1.1 using -O0, i.e., no code optimization. The case study comes with 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 predefined delta step. The delta can be any integer divisor of the signal’s range. Thus, for our four 16 bit input signals (each having a range from 0..65535) and a delta of 4096 (resulting in 16 steps per signal) a total of 16^4 = 65536 test cases is generated. The engine model test case generator 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 (see Fig. 7). Moreover, certain error situations are stimulated by injecting sensor faults, e.g., one or more sensor faults at a time. The engine model test case generator is set up by a virtual execution time. As the controller software is designed to run with a 10ms period 15000 test cases correspond to the execution of 150s of virtual time.

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

The second tool chain is based on the GDB/ARMulator ISS that comes as a part of the GDB debugger provided with the ARM GCC tool chain. GDB/ARMulator is a pure functional, i.e., no cycle accurate, simulator/emulator of a single ARM CPU running in user mode. In contrast to QEMU, ARMulator relies on a simple instruction interpreter loop. Here, binary mutations are directly applied to the ARM executable prior to its execution. For mutant detection standard outputs are piped to a dump file in order to be compared to a golden run output using diff.

3.1 Test Quality

We consider three different metrics in order to assess the test quality of a used set of test cases: instruction coverage, mutation coverage, and mutant detection (killed mutants). Instruction coverage measures the percentage coverage of instruction words reached by the test set’s control flow. Mutation coverage measures the rate of mutants reached by control flow. Mutant detection (killed mutants) measures the percentage of mutants that were killed in terms of propagating a program deviation to the outputs. For the comparison of metrics accuracy, we consider two typical C mutation operators that were easily applied to all of the three tool chains: if(\texttt{cond}) \rightarrow \texttt{if(true)} and if(\texttt{cond}) \rightarrow \texttt{if(false)}. For proving the reasonability of our metrics, we matched mutations using the addr2line tool provided with the GCC binutils though our approach does not rely on exact mapping of source level to binary level. Therefore, we used the --O0 flag as the relationship of source to optimized binary code cannot be easily followed. However, our approach also applies to optimization.

As the case study source code contains 115 if-statements, this leads to a total number of 230 mutants by applying two mutations to each. Fig. 8 shows the testing metrics generated according to the test cases from the two generators. The x-axis denotes the num-

![Figure 6: Multicore host utilization by process forking.](image)

![Figure 7: Closed-loop engine model test case generator for the fault tolerant fuel injection controller software.](image)
number of applied test cases per mutant. The y-axis shows the corresponding metric in percent. Since the generated metrics are identical for all approaches, we proved that binary mutation testing can reach 100% accuracy w.r.t. the considered control flow mutators. It turned out that the significant increase of metrics between test cases #5000 and #10000 with the engine model test case generator corresponds to the stimulation of two sensor faults at a time leading to an increased code coverage. As expected, the engine model performs better in terms of providing sufficient test quality with few test cases as it is more aware of the functionality of the controller software. Table 4 shows a detailed evaluation of the test quality metrics w.r.t. different test case generator approaches.

![Table 4: Comparison of test quality metrics reached by different test case generator approaches.](image)

### 3.2 Test Performance

Fig. 10(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 and the x-axis denotes the number of applied test cases per mutant. As each test case was applied to all 230 mutants, this leads to a total number of tests to be investigated (including the golden run). Basically, we can see that all approaches scale linearly w.r.t. the number of test cases (and mutants).
(a) 65% covered mutants and 48% killed mutants applying 1048576 tests by delta test case generator.

(b) 100% covered mutants and 83% killed mutants applying 15000+1054 tests by engine model test case generator with CSP-based ATPG.

Figure 9: Annotated binary level control flow graphs of the FuelingMode_du function demonstrating efficient test quality improvement by constraint solving based automatic test pattern generation.
Typically, with source code mutation testing there is a higher base effort related to the number of mutants as each mutant has to be compiled from sources. Here, the native approach is dominated by compilation efforts, i.e., testing time increases very slightly with the number of test cases. Fig. 10(a) shows the break even for the GDB/ARMulator is only below 10-100 test cases per mutant.

The figures show that the break-even point can be extended to below 100,000-1,000,000 test cases and XEMU performs in average 100-1000x faster than GDB/ARMulator. Fig. 10(b) demonstrates the speed up achieved by online detection and mutant skipping extensions. Finally, Fig. 10(c) depicts the additional speed up that can be achieved by utilizing multicores. The gradient of the curve is nearly halved by doubling the available cores. We utilized four full cores with hyper threading.

4. RELATED WORK

Mutation testing has inherent higher execution costs, hence various mutant reduction and execution cost reduction techniques have been proposed [14]. 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 [19], has to be available for the generation of mutants. Moreover, most frameworks focus on high-level software programming languages such as C# and Java [7]. For example, a large set of C language mutation operators were introduced in [17]. Later it was shown in [16] that a reduced number of operators still achieves a high mutation score. For hardware design, CERTITUDE by SpringSoft supports functional qualification for C and VHDL/Verilog [21]. In [22] mutation operators for IP-XACT electronic component descriptions were introduced. In contrast the XEMU framework aims to leverage mutation testing in the embedded software domain, which is mainly C and SystemC based. By doing so, it also targets mutation faults at the application binary interface (ABI) for COTS libraries.

DeMillo et al. [10] modified a GNU C compiler chain to generate patches in order to enable compiler-integrated mutant generation. In [18] they also introduced Godzilla an automatic constraint-based test data generation framework which was integrated with the Mothra mutation testing framework.

The authors of [15] propose a SystemC error and mutation injection tool based on compiler injection via a plugin for the GCC compiler based on four mutant operators. Another approach for SystemC and TLM mutation testing [12] allows to selectively activate a mutant at a time through the use of a configuration 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 call-backs mechanism during translation. Moreover, targeting COTS libraries (with no source code available) our approach is language and compiler independent. In [4] the authors propose a software fault injection technique for the IA32 platform by means of machine code level patterns. Mutations are induced directly in the target executable. In contrast to traditional mutation testing, the targeted application is to emulate residual software.

Though we are considering embedded code such as ARM binaries, our approach can be even faster than native approaches by applying an extended QEMU dynamic translation. Additionally, this enables more complex and efficient mutations as it is based on a modified code translation at run-time and not on binary pattern search expression. Moreover, our framework also provides feedback to the verification engineer via graphically rendered CFGs annotated with testing results such as a lack of coverage and non detected faults with the corresponding address to line information.

5. CONCLUSION

In this paper we introduced the XEMU framework for efficient mutation testing of binary software. The testing is seamlessly integrated into binary translation cycle of QEMU software emulation cycle at run-time. Mutants and test patterns are derived from the original software binary under test by a CFG analysis prior to testing. Though we introduce our approach by mutation operators for the ARM instruction set, the basic principles are applicable to other embedded processors. Our approach comes with several major 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 specific faults of different target ISAs and tool chains, e.g., compiler bugs and anomalies in the code optimization or binary interface issues; (iii) we can considerably reduce the mutant generation, execution, and detection efforts. Our results are evaluated by a case study from the automotive industry, a fault tolerant fuel injection control system. Our experiments reached a 100% accuracy w.r.t. source code mutation testing at the same time providing a speed up of up to 100-1000x compared to the execution with GDB/ARMulator ISS. We can even outperform native execution as we avoid individual mutant compilation. The utilization of multicore hosts through efficient multi-threading further improves testing speed. By employing advanced ATPG techniques based on binary analysis and constraint solving, we improved the test quality significantly at the same time reducing the number of required test cases.

6. ACKNOWLEDGEMENTS

This work was partly funded by the German Ministry of Education and Research (BMBF) through the DFG SFB 614, the project SANITAS (01M3088), the ITEA2 projects VERDE (01S09012) and TIMMO-2-USE (01IS10034).

7. REFERENCES


(a) Speed comparison w.r.t. GDB/ARMulator and native execution.

(b) Speed up through online detection and mutant skipping.

(c) Speed up through multicore utilization.

Figure 10: Evaluation of mutation testing performance.

Analysis for Systemc Designs at TLM. In Test Workshop (LATW), 2011 12th Latin American, pages 1–6, march 2011.


