Emulation of Software Faults: a Field Data Study and a Practical Approach

Appendix A - G-SWFIT fault emulation operators

In this Annex we describe the fault emulation operators for our current implementation of G-SWFIT. We started with this particular set because it contains the most representative faults, according to our field-data study. We present here the operators for the IA32 architecture using the Intel notation.

Each operator is described according to the rules that define the search pattern and the code change to apply to the locations identified by the search pattern. The search is bound by constraints that help to avoid locations where the code change would not emulate a realistic fault. Note that the same constraint may appear in different operators. These constraints were defined based on our field data study presented in section 3 of the paper.

The process of target code analysis is based on a per-module strategy: each module of the target is analyzed separately from the others. This makes sense as modules are also self contained units at the source-code level (or at least they should, according to the programming best practices). A module is typically a sub-routine (a *function* in C, *function/procedure* in Pascal, etc.).

Modules starting and ending points relate to very specific instruction patterns and offer useful information to the code-analysis process (Table A.1). More specifically it is possible to determine the stack frame size (the value *"immed"* in the instruction "sub exp, *immed*"). The size of the stack frame offers hints on how many local variables are used in a particular module (some of the operators require this information).

Table A.1 – Module entry and exit points.

Module entry point		Module exit point	
Instruction sequence	Explanation	Instruction sequence	Explanation
push ebp mov ebp, esp sub esp, *immed*	stack frame setup	mov esp, ebp pop ebp ret	stack frame cleanup
Note: variations may include instructions *enter* / *leave*			

A.1 Operator for missing function call – OMFC

Operator OMFC (detailed in Table A.2) locates function calls in a context where the value returned by the function call (if any) is not used. The removal a function call from a context where its return value was being used would not represent real software faults. This operator also avoids removing calls in locations where that call is the single statement within its containing block (a comparative example is given in Table A.3).

Table A.2 – Operator OMFC.

Operator	Example	Example with fault	Search pattern	Code change
OMFC	function(…..);	~~function(…..);~~	CALL *target-address*	CALL instruction removed

Contraints	
Return value of the function must not being used	Constraint C01
Call must not be the only statement in the block	Constraint C02

Constraint C01 is implemented by checking that the CALL instruction is not followed by instructions that represent the usage of the returned value. The standard method of returning values from functions is via the EAX register. Thus, it is only needed to check if the value of that register is used after the call instruction. Any instruction that sets a new value for that register stops this check. If the instruction related to the call are followed by the signature of exit point of the module we assume that the return value of that function is being used in a return statement (as in the following: *return function();*).

Constraint C02 is implemented as follows: in a first step the full instruction sequence related to the function call is identified (as precisely as possible). This sequence includes the parameter passing before the actual call code and the "add esp, *immed"* instruction to reclaim the stack space used by the those parameters and the use of the return value (if any). It is worth noting that the "*immed"* value in the instruction "add esp, *immed*" enables the detection of how many parameters were passed to the function being called. In a second step, the boundaries of the code block where the call is located are identified. The following instructions mark block boundaries: module entry/exit points, unconditional jumps, and conditional jumps to backward locations. Table A.3 clarifies this issue with example of source code and compiled code for a call which is the single statement in its code block and call which is not the single statement within its code block. It is worth noting that the distinction of single statement / non-single statement can be adapted to any type of fault location.

Table A.3 – Single statement vs. multiple statements.

Single statement		Not single statement	
Source code	**Compiled code**	**Source code**	**Compiled code**
if (a==123) { b = function(c); }	mov offA[ebp], 123 cmp offA[ebp], 0 je *loc-01* mov eax, off-C[ebp] push eax call *function-address* add esp, 4 mov off-B[ebp], eax *loc-01:*	if (a==123) { b = function(c); **c++;** }	mov off-A[ebp], 123 cmp off-A[ebp], 0 je *loc-01* mov eax, off-C[ebp] push eax call *function-address* add esp, 4 mov off-B[ebp], eax **mov ecx, off-C[ebp]** **add ecx, 1** **mov off-C[ebp], ecx** *loc-01:*

A.2 Operator for missing variable initialization with a value – OMVIV

This operator reproduces the omission of the initialization of a given local variable with a constant value (Table A.4).

Table A.4 – Operator OMVIV.

Operator	Example	Example with fault	Search pattern	Code change
OMVIV	var = *value*	~~var = *value*~~	mov *offset* [ebp], *value*	MOV instruction removed

Contraints	
Assignment must not be the only statement in the block	Constraint C02
Variable must be inside stack frame	Constraint C03
Must be the first assignment for that variable in the module	Constraint C04
Assignment must not be inside a loop	Constraint C05
Assignment must not be part of a *for* construct	Constraint C06

Constraint C02 is addressed in the same manner as in the previous operator. The only difference is that the sequence of low-level instructions is related to an assignment instead of a function call.

Constraint C03 addresses some situations where the compiler uses temporary locations within the stack to hold temporary values. This can happen in the evaluation complex expressions. We also observed that the use of macros also tend to cause this situation. This constraint filters out these cases because the stack locations involved do not relate to real variables in the source-code. Valid fault locations for this operator are those that involve parameters passed to the function (if any) or local variables. Function parameters are memory locations within the stack with a positive offset relative to *ebp*; local variables are locations

within the stack with a negative offset relative to *ebp* and within the stack frame (recall that this information is retrieved from the module entry point instruction sequence).

The fact that this operator only searches for variable initialization implies that only the first occurrence of an assignment to that particular variable are eligible for fault locations. Constraint C04 checks this requirement analyzing the module code from the beginning to detect any instruction that modifies the contents of the memory location of the variable under consideration.

Any assignment that occurs within a loop can never be considered as variable initialization because the assignment will occur several times. Thus, any assignment occurring within a loop will be considered as a simple variable assignment and not an initialization (variable assignments are addressed through different operators). Code inside a loop is relatively easy to identify though the existence of a *jump* instruction later in the module that jumps to an address lower than that of the code being considered. This verification is addressed by constraint C05.

Variables that constitute loop counters in *for* constructs are not interesting to emulate a missing variable initialization fault. This observation is based on our field-data study as none of the missing variable initialization where located within a *for* construct[1]. Constraint C06 addresses this issue by detecting the instruction patterns that relates to *for* constructs (some compiler optimization settings may interfere with the ability to detect *for* constructs). Table A.5 presents an example of this type of construct.

Table A.5 – Example of code relative to a *for* construct.

***for* construct**		
Source code	**Compiled code**	**Pattern notes**
for (i=0; i<10; i++) { /* ... */ }	mov *off-i*[ebp], 0 jmp loc-02 loc-01: mov eax, *off-i*[ebp] add eax, 1 mov *off-i*[ebp], eax loc-02: cmp *off-i*[ebp], 10 jge loc-03 ... jmp loc-01 loc-03:	initialization + jump variable modificatin followed by a test test stop condition code inside the loop jump to next iteration code past the loop

[1] From a programmer viewpoint, we also believe that it is much more difficult to forget to initialize a loop counter than to initialize a regular variable. However, we based our decision on the field-data study.

A.3 Operator for Missing Variable Assignment with a Value – OMVAV

This operator reproduces the omission of the assignment of a given local variable with a constant value (Table A.6). The first assignment to a given variable within a module is considered an initialization and it is if filtered out by constraint C07. This constraint is implemented in a similar way to constraint C04. The difference is that the decision mandated by the constraint is reversed: the fault location is considered eligible only is there is at least one previous instruction that stores a value or register in the stack location related to the variable being considered. All other constrains are as described before.

Table A.6 – Operator OMVAV.

Operator	Example	Example with fault	Search pattern	Code change
OMVAV	var= … … var = *value*	var= … … ~~var = *value*~~	mov *offset* [ebp], *value*	MOV instruction removed

Contraints	
Assignment must not be the only statement in the block	Constraint C02
Variable must be inside stack frame	Constraint C03
Must not be the first assignment for that variable in the module	Constraint C07
Assignment must not be part of a *for* construct	Constraint C06

A.4 Operator for Missing Variable Assignment with an Expression – OMVAE

Operator OMVAE is similar to operator OMVAV with the difference that it addresses variables being assigned the result of an expression instead of a constant value (see Table A.7). The applicable constrains are also the same as those of operator OMVAV.

Table A.7 – Operator OMVAE.

Operator	Example	Example with fault	Search pattern	Code change
OMVAE	var= … … var = *expression*	var= … … ~~var = *expression*~~	mov *offset* [ebp], *reg*	MOV instruction removed

Contraints	
Assignment must not be the only statement in the block	Constraint C02
Variable must be inside stack frame	Constraint C03
Must not be the first assignment for that variable in the module	Constraint C07
Assignment must not be part of a *for* construct	Constraint C06

Operators OMVAE and OMVAV can be merged into a new one (operator OMVA). This new operator would share the constrains and code changes and would have the following search pattern "mov *offset*[ebp],...".

A.5 Operator for Missing If Around statements – OMIA

Operator OMIA reproduces a missing *if* condition surrounding a set of statements (see Table A.8). The effect of this fault type is that the statements surrounded by the *if* construct are always executed instead of being executed only if a given condition is true. The effect of the operator is the removal of the instructions that cause the execution flow to jump over the statements surrounded by the *if* construct.

Table A.8 – Operator OMIA.

Operator	Example	Example with fault	Search pattern	Code change
OMIA	if (expression) { *statements* }	~~if (expression)~~ ~~{~~ *statements* ~~}~~	cmp *reg*, … *jcond* after ... cmp *reg*, … *jcond after* *statements* after:	The conditional jumps to the address *after* are removed It may be a single jump if the expression is simple

Contraints	
The *if* construct must not be associated to an *else* construct	Constraint C08
statements must not include more than five statemens and not include loops	Constraint C09

Notes
Currently, the side effects of the fisrt sub-expression of the complete expression is not ommitted
There may be several cond. jumps to *after* if expression is composed of several sub-expressions

The code change includes the removal of the all instructions located between the removed jumps and that change the contents of the memory. This removal is intended to emulate the omission of the side effects of expressions such as "*i*++ > *val*". In this case the side is the modification of the variable *i*. The operator does not emulate the removal of the side effects of the first sub-expression (if the expression of composed of several sub-expressions). The non removal of the possible side effects of the first sub-expression may cause some loss in the operator accuracy.

Our field data study suggests that the statements related to a missing if statement were always function calls and assignments. All assignments to variables ultimately refer to the modification of a memory location (except register variables). Constraint C09 addresses this aspect: the missing statements are contiguous and belong to the same code block. Additionally, there can be no *jump* instructions inside the considered location because if-constructs and loops are not allowed in these locations. The use of

optimized compilation may cause this constraint loose some accuracy if some variables are entirely kept in processor registers. The use of macros in the source code may also affect this constraint if the macro expansion contains a large portion of code.

Cases in which the *if* construct being removed is associated to an *else* construct are considered as a different fault type and addressed though another operator (described below). Constraint C08 filters out situations of *if-else* by detecting the particular interleaved jump instructions that is specific to this construct (see Table A.9).

Table A.9 – Example of code related to an *if-else* construct.

if-else construct		
Source code	**Compiled code**	**Pattern notes**
if (a == 123) { /* ... */ } else { /* ... */ } /* remaining code */	cmp *off-a[ebp], 123* jne *loc-01* ... jmp *loc-02* loc-01: ... loc-02: remaining code ...	condition test jump to "else" part "if true" code uncond. Jmp first jmp dest. } Adjacent "else" code code inside the loop jump to next iteration code past the loop

A.6 Operator for Missing IF construct and surrounded Statements – OMIFS

Operator OMIFS emulates a missing *if* construct and the statements surrounded by it (see Table A.10). This effect is achieved changing the conditional jumps into unconditional jumps. Thus, the surrounded statements are always avoided. This operator is similar to OMIA with the difference that the statements surrounded by the *if* construct are removed from the execution flow. The same constrains also apply.

Table A.10 – Operator OMIFS.

Operator	Example	Example with fault	Search pattern	Code change
OMIFS	if (expression) { *statements* }	~~if (expression)~~ ~~{~~ ~~*statements*~~ ~~}~~	cmp *reg*, … *jcond* after … cmp *reg*, … *jcond after* *statements* after:	All the conditional jumps to the address *after* are made into unconditional jumps

Contraints	
The *if* construct must not be the only statement in the block	Constraint C02
The if construct must not be associated to an else construct	Constraint C08
statements must not include more than five statemens and not include loops	Constraint C09

Notes
Currently, the side effects of the fisrt sub-expression(e.g., "var++ > 0") s not ommitted
There may be several cond. jumps to *after* if expression is composed of several sub-expressions

This operator removes the possible side effects of the sub-expressions composing the conditional expression of the *if* construct. The method is the same as in the case of operator OMIA.

A.7 Operator for Missing IF construct plus statements plus else before statements – OMIEB

Operator OMIEB emulates a missing *if* construct and the statements surrounded by it plus an else statement before a set of statements (see Table A.11). This operator is in fact similar to operator OMIA with the difference that the *if* construct must be associated to an *else* construct. The effect of this operator is the modification of the conditional jumps related to the expression evaluation into unconditional jumps to the start of the statements inside the else construct. Thus, the statements surrounded by the *if* construct are always avoided and execution flows directly to the statements surrounded by the *else* construct.

It is possible that the expression used as the condition in the *if* construct has side effects (e.g., “i-->0”). To emulate the omission of side effects in the expression of the *if* construct being removed, any *call* instructions or memory-storing instruction existing between the conditional jumps are removed.

Table A.11 – Operator OMIEB.

Operator	Example	Example with fault	Search pattern	Code change
OMIEB	if (*expression*) { *statements-IF* } else { *statements-ELSE* } *… remaining code*	~~if (expression)~~ ~~{~~ ~~*statements-IF*~~ }~~ ~~else~~ ~~{~~ *statements-ELSE* ~~}~~ *… remaining code*	*flag-affecting instr.* *jcond* elsecode *… instrs (IF)* jmp *after* elsecode: *… instrs (ELSE)* after: *… remaining code*	- All the conditional jumps to the address loc01 are changed into unconditional jumps - Call instructions and stores to memory existing between the cond jumps are removed

Notes
There may be several cond. jumps to *elsecode* if expressions is composed of several sub-expressions The side-effects (if any) of the first sub-expression are not ommited

A.8 Operator for Missing "and sub-expression" in logical expression used in branch condition – OMLAC

Operator OMLAC emulates the omission of part of a logical expression used in a branch condition (Table A.12). The logical expression is composed of a sequence of at least two sub-expressions linked together with the logical operator AND. If at least one of the sub-expressions evaluates to "false" the entire expression will also evaluate to "false" and the condition fails. Thus, these expressions usually originate a series of conditional jumps instructions to the same address (one for each sub-expression). The target address of the conditional jumps is the position in the code just after the instructions that are to be executed if the expression is "true". The effect of the omission of one of the sub-expressions can be emulated by removing its related jump instruction.

Table A.12 – Operator OMLAC.

Operator	Example	Example with fault	Search pattern	Code change
OMLAC	if (*expr1* && *expr2* *… && exprN*) { *statements* }	if (~~*expr1*~~ && *expr2* *… && exprN*) { *statements* } **Note**: each *expr* removed is a fault	*flag-affecting instr.* *jcond* after ... *flag-affecting instr.* *jcond* after *statements* after:	One of the conditional jumps is removed (each one is a different fault)

Notes
The complete logical expression must contain at least two sub-expresssions

A.9 Operator for Missing "or sub-expression" in logical expression used in branch condition – OMLOC

Operator OMLOC emulates the omission of part of a logical expression used in a branch condition (Table A.13). The logical expression is composed of a sequence of at least two sub-expressions linked together with the logical operator OR. Each sub-expression is enough to make the entire expression have the result "true". Therefore, the typical instruction sequence related to this type of construct contains a sequence of conditional jumps each one pointing to the same address. This address is the location of the code to be executed when the logical expression is true. That portion of code is preceded by a jump to an address just after its last instruction. This jump is executed only if all sub-expressions fail. To emulate the omission of one of the sub-expression the conditional jump related to that sub-expression is removed.

Table A.13 – Operator OMLOC.

Operator	Example	Example with fault	Search pattern	Code change
OMLOC	if (*expr1* || *expr2* … || *exprN*) { *statements* }	if (~~*expr1*~~ || *expr2* … || *exprN*) { *statements* } **Note**: each *expr* removed is a fault	flag-affecting instr. *jcond* before … flag-affecting instr. jcond before ... flag-affecting instr. *jcond after* before: *statements* after:	One of the conditional jumps is removed (each one is a different fault)

Notes
The complete logical expression must contain at least two sub-expresssions

A.10 Operator for Missing Localized Part of the Algorithm – OMLPA

Operator OMLPA attempts to reproduce the omission of a small localized part of the algorithm (example in Table A.14). We concluded from our field-data study that this type of faults never involved the omission of *if / if-else* and loop constructs (the missing statements were always function calls and assignments). This is addressed by constraint C10. This constraint is similar to C09 as described before with one additional restriction: the missing statements are contiguous and belong to the same code block. Thus, eligible location for this fault can not contain target addresses of jumps elsewhere in the modules (this constraint is verified analyzing the entire module).

Table A.14 – Operator OMLPA.

Operator	Example	Example with fault	Search pattern	Code change
OMLPA	… *statement* *statement* … *statement*	… ~~*statement*~~ ~~*statement*~~ ~~…~~ ~~*statement*~~	Sequence of instructions not containg more that five *mov [address], …* and not containing cycles or jump destinations	All instructions are removed

Contraints	
The statements to remove must not be the only code within its block	Constraint C02
statements are in the same block, do not include more than 5 stats. nor loops	Constraint C10

If the code removed were the complete block (e.g., all the instructions inside a loop) this would not correspond to a realistic fault. Constrains C02 assures that such situations are avoided.

A.11 Operator for Wrong Value Assigned to a Variable – OWVAV

This operator emulates the assignment of a wrong value to a variable (Table A.15). To avoid random factors in fault injection the wrong value to use in the fault emulation is obtained in a deterministic manner: the bits of the least significant byte of the value are inverted. We choose the least significant byte in order to affect all data sizes. The initialization of a variable with a wrong value is considered a different fault type (less common, according the data-filed study results). Constraint C07 filters out variable initializations. Constraints C03 and C06 have the same reasoning as in the operators OMVAE and OMVAV.

Table A.15 – Operator OWVAV.

Operator	Example	Example with fault	Search pattern	Code change
OWVAV	var= … … var = *value*	var= … … var = *value & 0xFF*	mov *offset* [ebp], *value*	All bits of the low order byte are reversed

Contraints	
Variable must be inside stack frame	Constraint C03
Must not be the first assingment to that variable in the module	Constraint C04
The assignment must not be part of a *for* construct	Constraint C06

A.12 Operator for Wrong variable in parameter of function Call – OWPFV

Operator OWPFV emulates the use of a wrong variable as a parameter in a function call (Table A.16). This operator locates CALL instructions is a similar way as OMFC does. Variables being used as parameters are identified by the detecting *push reg* instructions where *reg* was previously fetched from

the stack (these actions mean the fetching and pushing of a stack-resident variable). The emulation consists of changing the variable that is being fetched. To that effect the offset into the stack is changed by four bytes (assuming a 32 bits architecture). Recall that a local variable is identified as a reference to a stack location with a negative offset relative to *ebp*, and a parameter passed to the module is identified as a reference to a stack location with a positive offset relative to *ebp*. The algorithm to decide how the offset to the variable is modified is a follows:

- If the variable originally being used is a local variable (offset negative) and there are at least two local variables in that module, the offset is changed to point to the neighbor variable: the offset is increased by 4 unless the original variable is the last one, in which case he offset is decreased by 4. Recall that the number of local variables can be hinted by the stack frame size which is available in the module entry point.
- If the variable originally being used is a parameter of the module (offset positive) and there are at least two parameters passed to that module, the offset is changed to point to the neighbor parameter: the offset is decreased by 4, unless the original variable is the last one, in which case the offset is increased by 4. The number of parameters passed to the module cannot be directly assessed as in the case of the local variables. To determine the number of parameters, the code of the entire module is analyzed and the number of references to distinct positive offsets in the stack is counted.
- If the original variable is a local variable and there is only one local variable, but there is at least one parameter passed to the module, then the offset is changed to point to the first parameter.
- If the original variable is a parameter passed to the module and there is only one parameter, but there is at least one local variable, then the offset is changed to point to the first local variable.

Constraint C09 assures the verification that there must be at least two variable in the module (either local variable or parameters).

This operator has the limitation that it does not consider the data type of the variables. As such, it is possible that a given mutation may not relate to a syntactically correct change in the source-code.

Table A.16 – Operator OWPFV.

Operator	Example	Example with fault	Search pattern	Code change
OWPFV	function(…, *var1*, ...)	function(…, *var2*, ...)	mov *reg*, *offset* [EBP] push *reg* call *address*	*offset* is changed to refer another variable

Contraints	
Variable must be inside stack frame	Constraint C03
There must be at leat two variables in this module	Constraint C11

Notes
There may be more than one push instruction depending on the number of parameters
For a given call, each offset that is changed is a different fault

A.13 Operator for Wrong Arithmetic Expression in a function Parameter – OWAEP

This operator emulates faults that consist on having a wrong arithmetic expression used as parameter of a function call. To emulate this fault operator OWAEP omits the last arithmetic operation prior to the expression result being pushed into the stack (see Table A.17).

Table A.17 – Operator OWAEP.

Operator	Example	Example with fault	Search pattern	Code change
OWAEP	function(*expr*);	function(*incomplete expr*);	arith. inst. affecting *reg* push *reg* call *address*	The arithmetic instruction is ommited

Notes
The effect of this operator is as if the last operation of the expression is ommited