Mutation Testing of Memory-Related Operators

Jay Nanavati, Fan Wu, Mark Harman, Yue Jia and Jens Krinke
Dept. of Computer Science
University College London

***Abstract*—Though mutation operators have been designed for a wide range of programming languages in the last three decades, only a few operators are able to simulate memory faults. This paper introduces 9 Memory Mutation Operators targeting common memory faults. We report the results of an empirical study using 16 open source programs, which come with well designed unit test suites. We find only 44% of the new memory mutants introduced are captured by the traditional strong mutation killing criterion. We thus further introduce two new killing criteria, the Memory Fault Detection and the Control Flow Deviation killing criteria to augment the traditional strong mutation testing criterion. Our results show that the two new killing criteria are more effective at detecting memory mutants, killing between 10% and 75% of those mutants left unkilled by the traditional criterion.**

I. Introduction

Mutation testing is a fault-based testing methodology that aims to identify whether a codebase is vulnerable to specific classes of faults [1]. By simulating simple bugs in a program, this technique can identify vulnerabilities that are not detected by traditional testing techniques such as unit testing. A recent survey [2] has also shown that this mutation methodology is gaining traction and it is being used in a growing number of large scale commercial and experimental projects.

Although many (alternate) forms of testing exist, work on memory vulnerabilities detection [3]–[5] in C applications has shown the existence of vulnerabilities such as uninitialized memory access, buffer overruns, invalid pointer access, beyond stack access, free memory access and memory leaks in published code. Such vulnerabilities have been ranked highly in the CWE SANS top 25 most dangerous programming errors [6]. Moreover, such vulnerabilities are highly prone to exploitation. For example, vulnerabilities such as a buffer overflows when using *malloc()* facilitate exploits that overwrite heap metadata, gain access to unavailable function/data pointers, overwrite arbitrary memory locations, and create fake chunks of memory that may contain modified pointers.

Traditional Mutation Operators only simulate some simple syntactic errors based on the Competent Programmer Hypothesis [7]. Mutation Testing using these operators may drive testers to generate test suites mainly targeting such errors. This may lead to a lack of testcases revealing memory faults, thus a weakness in traditional Mutation Testing. To overcome this problem, we propose 9 Memory Mutation Operators simulating three classes of common memory faults. However, memory faults do not necessarily propagate to the output, making strong killing criterion, which is widely adopted in traditional Mutation Testing, not adequate to detect such faults. To augment traditional Mutation Testing for detecting memory faults, we introduce two additional weak killing criteria, i.e. Memory Fault Detection and Control Flow Deviation. A single Mutation Testing tool was developed with Memory Mutation Operators, with the traditional strong killing criterion and the proposed weak killing criteria incorporated. Using our tool, we applied traditional Mutation Testing and both of the proposed criteria to 16 subject programs. The results show that, among 359 generated mutants, traditional strong killing criterion killed only 44% of the mutants, leaving 201 unkilled. After Memory Fault Detection and Control Flow Deviation killing criteria are introduced, 10% to 75% of those survived mutants are killed across all subject programs.

The primary contributions of this paper are as follows:

1) We build a Mutation Testing tool that applies both traditional and Memory Mutation Operators and takes Memory Fault Detection and Control Flow Deviation as additional killing criteria into account.
2) 9 Memory Mutation Operators are introduced to mimic several categories of memory faults, revealing weaknesses in traditional Mutation Testing. On 16 subject programs, Memory Mutation Operators successfully insert memory faults and generate 359 mutants.
3) We propose Memory Fault Detection (using Valgrind for precise assessment of memory fault detection) and Control Flow Deviation as additional killing criteria, and show that up to 75% of surviving mutants are killed by these additional criteria.

The rest of the paper is organized as follows: background theory and the problem statement are presented in Section II, while the methodology including Memory Mutation Operators and proposed new killing criteria are presented in Section III together with a list of research questions. Section IV introduces the Mutation Testing framework and experimental setting, the results and analysis of which are shown in Section V. We summarise the threats to validity and related work in Section VI and Section VII respectively, followed by conclusions in Section VIII.

II. Background

This chapter provides an overview of background information relating to mutation testing.

A. Mutation Testing

Mutation Testing [1] is a white box testing technique that measures the quality/adequacy of tests by examining whether the test set (test input data) used in testing can reveal certain types of faults. A mutation system defines a set of rules (mutation operators) that generate simple syntactic alterations (mutants) of the program under test (PUT), representing errors that a “competent programmer” would make, known as the Competent Programmer Hypothesis (CPH) [7].

To assess the quality of a given test suite, the set of generated mutants are executed against the input test suite to determine whether the injected faults can be detected. If a test suite can identify a mutant from the PUT (i.e. produce different execution results), the mutant is said to be *killed*. Otherwise, the mutant is said to have *survived* (or live). A mutant may remain live because either it is equivalent to the original program (i.e. it is functionally identical to the original program although syntactically different) or the test suite is inadequate to kill the mutant. The Mutation Score (MS) is used to quantify how adequate a test suite is in detecting the artificial faults. It is calculated as the following formula:

$$MS(P,T) = \frac{\text{number of mutants killed}}{\text{total number of mutants generated}}.$$

P is the program under test and T is the set of tests. This metric is traditionally used as an estimation of test suite effectiveness. However it is biased by the number of equivalent mutants among all generated mutants.

The equivalent mutant problem is a major impediment to large scale wide spread use and whether a mutant is equivalent has been proven to be undecidable [8], [9]. Although it has been shown that the problem of detecting equivalent mutants cannot be completely automated, approaches to partially solve this problem have been introduced. They consist of applying compiler optimization techniques [9] and detecting infeasible paths using static analysis [10]. Other work combines mutants to generate HOMs (Higher Order Mutants) followed by using the number of unit tests that killed FOMs (First Order Mutants) that make up a HOM to identify equivalent mutants [11]. Co-evolution has also been proposed to achieve tailored selective mutation to partially evaluate mutants [12]. Due to the undecidable nature of this problem and the requirement of a human in order to solve it, it can be considered as a Human Intelligence Task (HIT) [13] such that the cost of the human oracle associated increases with the scale of the program under test.

B. Problem Statement

Though there are many Mutation Testing engines [14]–[23] for different programming languages, they do not address the issue of evaluating test suite effectiveness with regards to memory based faults adequately. Using this as a basis for our investigation, we propose a set of mutation operators that aim to model such faults and developed a Mutation Testing tool for C programs that integrates these Memory Mutation Operators.

In traditional Mutation Testing, only the mutant’s outputs are compared against the original program’s outputs to kill the mutants, while much more information are generated during the test, such as implicit memory faults and control flow graphs. The problem of equivalent mutants may be alleviated by using a richer source of other information to eliminate some seem-to-be equivalent mutants. In this paper, we propose two more killing criteria for memory-related mutation testing, Memory Fault Detection and Control Flow Deviation. The number of detected memory faults in a mutant is used to suggest whether the mutant is potentially equivalent to the original program, while Control Flow Deviation is to tell whether a mutant executes a different path than the original. More details about Memory Fault Detection and Control Flow Deviation criteria can be found in Section III-B and Section III-C respectively.

III. Methodology

In this section, we first present 9 Memory Mutation Operators we designed to simulate the memory faults, as well as Memory Fault Detection and Control Flow Deviation killing criteria. Then we list the research questions that we aim to answer in this paper.

A. Memory Mutation Operators

A set of 9 **Me**mory **M**utation **O**perators (**MeMO**s) are proposed in this project. Each mutation operator mutates calls to memory related function calls (e.g. *malloc()*, *calloc()*, or *free()*), their arguments, or assignments of *NULL*. We divide these mutation operators into three categories based on the faults they inject into the code base: *uninitialized memory access*, *faulty memory allocation* and *faulty heap management*.

Table I lists the proposed operators together with a brief descriptions. All of the mutation categories evaluate the inherent vulnerabilities of memory related functions in C programs which concerned previous work [24]–[26].

Each MeMO is detailed in one of the three categories below as well as a rationale behind its choice.

1) Uninitialized Memory Access: The proposed operators in this category generate mutants that can cause uninitialized memory to be accessed in the program. As defined in the C specification, memory allocated by *malloc()* is not guaranteed to be initialized in comparison to *calloc()*, which initialises the memory to 0. **REC2M** replaces instances of *calloc()* with *malloc()* in order to inject uninitialized memory usage faults into the program (Table II).

TABLE II. Example of **REC2M**

Original Program (P)	`int *array;`
	`array = calloc(15, sizeof(int));`

Mutated Program (P')	`int *array;`
	`array = malloc(15 * sizeof(int));`

RMNA removes *NULL* assignment statements. Depending on the usage of the mutated pointer, faults such as dangling pointers or dereferencing uninitialized pointer are injected. Table III shows an example application of the **RMNA** operator, where the mutated program P' is an example of how this mutation operator can inject a dangling pointer fault and introduce undefined behaviour into the application.

TABLE III. Example of **RMNA**

Original Program (P)	`char *s = calloc(15,sizeof(char));`
	`...`
	`free(str);`
	`str = NULL;`

Mutated Program (P')	`char *s = calloc(15,sizeof(char));`
	`...`
	`free(str);`
	`str;`

TABLE I. MEMORY MUTATION OPERATORS

Category	Operator	Brief Description
Uninitialized Memory Access	REC2M	Replace *calloc()* with *malloc()*
	RMNA	Remove *NULL* character assignment statement
Faulty Memory Allocation	REDAWN	Replace dynamic memory allocation calls *malloc()*, *calloc()*, *alloca()* and *realloc()* with *NULL*
	REDAWZ	Replace size of the requested block with 0 for dynamic memory allocation functions
	RESOTPE	Replace the arguments of the *sizeof()* unary operator with the pointer type equivalent if a non-pointer type is specified
	REMSOTP	Replace the arguments of the *sizeof()* unary operator with the non-pointer type equivalent if a pointer type is specified
Faulty Heap Management	RMFS	Remove *free()* statement
	REM2A	Replace *malloc()* with *alloca()*
	REC2A	Replace *calloc()* with *alloca()*

2) Faulty Memory Allocation: The proposed operators in this category generate mutants that mutate the way memory is allocated in order to measure the effectiveness of test suites at detecting faults such as buffer overflow, underflow and undefined behaviour. Variable length arrays (VLAs) are a class of C arrays that can be declared with a size that is not a constant integer expression, where the size expression is evaluated at runtime. According to the C specification, if size arguments of VLAs are not in a valid ranges, this could result in undefined behaviour. Moreover, a violation of this constraint does not stop code from being compiled and no compiler warning will be generated.

TABLE IV. EXAMPLE OF **REDAWN**

Original Program (P)	`char *str;` `str = malloc(6*sizeof(char));` `strcpy(str, "hello");`
Mutated Program (P')	`char *str;` `str = NULL;` `strcpy(str, "hello");`

REDAWN replaces instances of memory allocation calls with *NULL* in order to generate mutants which measure the effectiveness of test suites in identifying faults that occur due to unchecked return value of memory allocation functions. Table IV shows an application of this mutation operator.

REDAWZ replaces the request size passed to memory allocation calls with 0 in order to inject *zero allocation* faults. The C specification states that for any of the memory allocating functions, if a memory block of size 0 is requested, the behaviour is implementation-defined, i.e. the value can be a *NULL* pointer or a unique pointer. One of the problems this can cause is that, for those implementations where allocation functions return a unique pointer, *NULL* checks, which are considered adequate when receiving pointers through dynamic allocation, will pass. This assumption can also lead to tests not detecting such faults. Table V shows an application of **REDAWZ** where in P' although allocation failed to return the request size, the program will still trust the allocated pointer and return *true*, unless the programmer wrote special checks to account for such behaviour.

TABLE V. EXAMPLE OF **REDAWZ**

Original Program (P)	`int num = malloc(10*sizeof(int));` `return num!=NULL;`
Mutated Program (P')	`int num = malloc(0);` `return num!=NULL;`

RESOTPE and **REMSOTP** mutate the data type of the *sizeof()* operator that is typically used by programmers when dynamically allocating memory. This mutation aims to generate faults that model the incorrect use of the *sizeof()* operator on pointer data types. On some architectures it may be possible that for a given data type T, *sizeof(*T*)* is equal to *sizeof(*T**)*. This may lead programmers that lack understanding of the C programming language to believe that this is indeed the case. However, the effect of this is: *sizeof(*T*)* returns the size of the data type T itself, while *sizeof(*T**)* returns the size of a pointer (to T). Moreover, the C standard allows pointers to different types to have different sizes, e.g. *sizeof(char*)* is not necessarily the same as *sizeof(int*)* which implies that *sizeof(*T**)* is not guaranteed to always be the same regardless of the type of T. Faults of this nature can cause incorrect memory allocations and lead to buffer overflows and memory leaks. Table VI shows an application of these mutation operators.

3) Faulty Heap Management: The proposed operators in this category generate mutants that model faults that can occur due to improper memory management and also to test the effectiveness of test suites in handling events where allocation functions may fail due to a lack of free memory.

RMFS mutates instances of the *free()* standard C library function by removing instances of *free()* in order to inject memory leaks into the program. **REM2A** and **REC2A** replaces instances of *malloc()* and *calloc()* with *alloca()*. Mutants generated by these operators dynamically allocate memory on the stack instead of the heap. If a pointer to that memory is dereferenced after the function containing the allocating call finished, the pointer is dangling and the memory may have meanwhile been overwritten. Another problem with using *alloca()* is the fact it is not guaranteed to return *NULL* if it fails to find enough space on the stack and, depending on the implementation, it may cause some parts of the stack to be overwritten. On the other hand, as memory allocated using *alloca()* is automatically freed once the function that called it returns to its caller, it can be possible that this mutation may lead to memory leak error correction in the program. This is because as shown in Table VII below, the program P does not free the allocated memory whereas in P' there is no need to manually call *free()* in order to free the allocated memory.

B. Memory Faults

Traditional Mutation Testing only uses the test output of the mutants and the test output of the original to (strongly) kill the mutants. However, executing the mutants against a testsuite generates much richer information that is neglected in traditional Mutation Testing. Memory Fault Detection can generate one kind of such additional information. If a memory fault is detected in a mutant but is not detected in the original, one can be sure that the memory fault must have been introduced into the mutant by the mutation operator, thus the mutant can not be equivalent to the original in every sense

TABLE VI. EXAMPLES OF **RESOTPE** AND **REMSOTP**

	RESOTPE	**REMSOTP**
Original Program (P)	`char *str;` `str = malloc(10*sizeof(char));`	`char *str;` `str = malloc(10*sizeof(char*));`
Mutated Program (P')	`char *str;` `str = malloc(10*sizeof(char*));`	`char *str;` `str = malloc(10*sizeof(char));`

TABLE VII. EXAMPLES OF **RMFS**, **REC2A** AND **REM2A**

	RMFS	**REC2A**	**REM2A**
Original Program (P)	`T *data;` `data = malloc(8*sizeof(T));` `free(data);`	`int *nums;` `nums = malloc(8*sizeof(int));`	`int *nums;` `nums = calloc(8, sizeof(int));`
Mutated Program (P')	`T *data;` `data = malloc(8*sizeof(T));`	`int *nums;` `nums = alloca(8*sizeof(int));`	`int *nums;` `nums = alloca(8*sizeof(int));`

of "equivalent". Memory Fault Detection can reveal presence of memory faults that do not always propagate to the output, so some of the memory faults may not reveal themselves in the test outcome, yet we still have other ways to detect them during the execution. In this paper, we propose using *Valgrind* to detect memory faults as an additional weak mutation killing criterion. Despite other tools that can also detect memory faults, we choose *Valgrind* because it is a stable tool and widely used for memory fault detection. *Valgrind* can easily be changed to other memory fault detection means.

We say a mutant M_i is killed by the Memory Fault Detection (MFD) criterion if $\mathrm{MFD}(M_i, t) > \mathrm{MFD}(P_{\mathrm{UT}}, t)$ for any testcase t, where $\mathrm{MFD}(P, t)$ is the number of memory faults found from program P when running it against testcase t. Notice that the original program may also contain some memory faults, we only care whether a mutant contains 'more' memory faults than that in the original.

Since the MeMOs are designed to simulate real memory faults, we expect to see a large number of mutants generated from MeMOs can not be killed by the traditional criterion but are killed by the MFD criterion.

C. Control Flow Deviation

A program can be considered as a network of nodes which represent branches in the program and each test as a possible entry point into the network. If we consider the execution of each mutated program as a path from a starting node (the point in the program execution at which the mutant is executed) to a sink node (the point at which the program terminates), then the set of mutants which generate a path that is different in comparison to the path generated by the PUT can be classified as weakly non-equivalent mutants.

Similar to the representation of control flow graphs [27], in Fig. 1 test cases are represented by circles labelled $t_1, \ldots, t_7$, the mutation execution point is labelled as M, the set of program branching states that makeup the control flow for a given program are labelled $s_1, \ldots, s_7$ and the sink state (program termination) is labelled s_8. Fig. 1 also shows an example execution of test case t_4 (two graphs to the left) where the control flow of the mutated program M $\langle t_4, s_1, s_3, s_5, s_6, s_8\rangle$ is able to deviate from the control flow of the original program P $\langle t_4, s_1, s_3, s_7, s_8\rangle$, whereas in the execution of test case t_7 (the graph to the right), the execution paths are the same $\langle t_7, s_1, s_2, s_3, s_7, s_8\rangle$. We consider this kind of deviation as a distinguishing factor that can help reduce the set of survived mutants. We also consider this deviation as a sign of test suite weakness, since the deviation shows that a mutant was able to execute undesired code and pass all the test cases regardlessly.

Let $CFG(P)$ be the control flow graph of program P and let $E(G, t)$ be the edge set of graph G for the execution of testcase t. We say a mutant M is killed by the Control Flow Deviation (CFD) criterion, iff

$$\begin{aligned} \exists t \in T : \quad & E_O = E(CFG(P_{\mathrm{UT}}), t) \\ \wedge \quad & E_M = E(CFG(M), t) \\ \wedge \quad & (E_O \cup E_M) - (E_O \cap E_M) \neq \emptyset \end{aligned}$$

On the other hand, if $E(CFG(M), t)$ is the same as $E(CFG(P_{\mathrm{UT}}), t)$, the mutant survives.

In our experiments, we use the CFD criterion to further reduce the number of survived memory mutants. However, CFD criterion is not designed to detect memory faults, thus can be adopted by any Mutation Testing framework.

D. Research Questions

We try to answer the following research questions:

RQ1 What are the characteristics of the proposed Memory Mutation Operators?

- **RQ1a** What is the prevalence of Memory Mutants?
- **RQ1b** How effective is each Memory Mutation Operator in generating memory faults?
- **RQ1c** What is the Mutation Score for the traditional criterion applied against the Memory Mutants?

RQ2 What is the reduction rate of survived mutants after introducing Memory Fault Detection and Control Flow Deviation as additional killing criteria?

RQ3 What is the relation between Memory Fault Detection and Control Flow Deviation killing criteria?

We ask the first research question to understand some basic properties of the proposed MeMOs. To obtain different aspects of the properties, we ask three sub-questions: Firstly, we would like to know how effective the Memory Mutation Operators are in injecting memory faults into subject programs in terms of the number of generated mutants. Then we ask a further question to understand the effectiveness of each one of the Memory Mutation Operators in generating mutants. Additionally, we are interested in how many mutants generated from these Memory Mutation Operators survive when applying

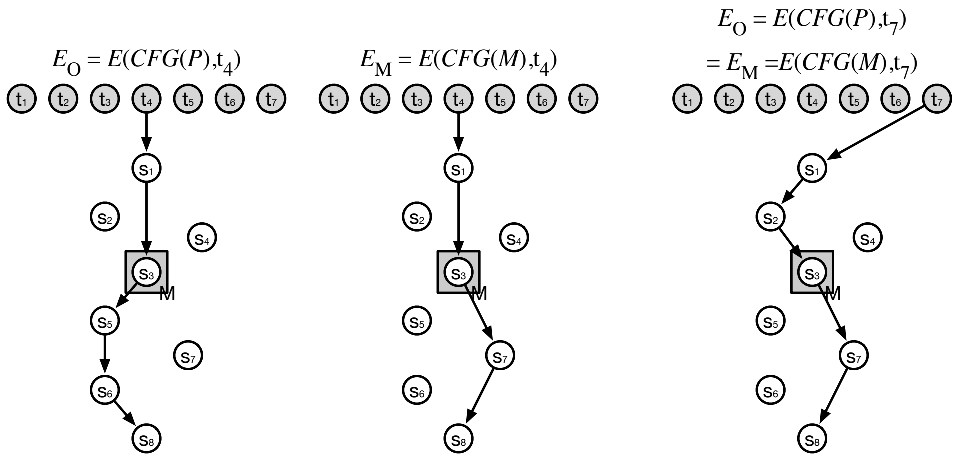

Fig. 1. Control Flow Deviation. Test t_4 generates different control flow before and after the mutation at state s_3 (the two graphs to the left). Test t_7 generates the same control flow (the graph to the right).

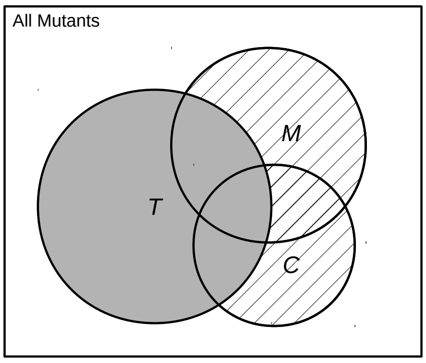

Fig. 2. Venn Diagram of the relation between mutants killed by each criterion. Sizes of the circles do not correspond to the real data.

only the traditional killing criterion, i.e. using only the test output of the original program and the test output of a mutant. The third subquestion can help us understand how weak traditional Mutation Testing is in revealing memory faults.

After we understand the weakness of traditional Mutation Testing in terms of memory faults, we ask the second research question to understand how effective the two newly proposed criteria are in reducing the number of equivalent mutants. Furthermore, we want to understand the relation between these two proposed killing criteria in terms of the number of mutants only killed by each criterion. By answering this question, we can see the effectiveness of each criterion and whether one criterion subsumes the other.

The relation between the mutants killed by each criterion can be easily understood by the Venn Diagram in Fig. 2. Sets T, M and C contain the mutants killed by a testsuite when the traditional (T), Memory Fault Detection (M) and Control Flow Deviation criterion (C) area applied respectively. Moreover, $\overline{S}$ represents the set of mutants that survive the corresponding criterion for $S \in \{T, M, C\}$. The Mutation Score for a criterion can be calculated as:

$$MS = \frac{|S|}{|\text{AllMutants}|}$$

In the Venn Diagram, set $\overline{T}$ (white area outside the grey circle) contains all mutants surviving traditional mutation testing, and from those the ones that interest us are the ones that can be killed by the MFD and CFD criterion (the shaded part). More formally, we answer **RQ2** by calculating the ratio of the shaded area contained in the white area (where $\overline{T} \cap (M \cup C)$ is the shaded area):

$$R_{\text{MFD+CFD}} = \frac{|\overline{T} \cap (M \cup C)|}{|\overline{T}|}$$

In order to answer **RQ3**, we calculate the size of two sets: $M_{\text{only}} = M \cap \overline{T} \cap \overline{C}$ and $C_{\text{only}} = C \cap \overline{T} \cap \overline{M}$. A bigger size of each of these two sets means that more mutants can only be killed by the corresponding criterion, thus the criterion contributes more in reducing the number of survived mutants.

IV. Experiments

We have built up a Mutation Testing tool targeting C programs which can support Memory Fault Detection and Control Flow Deviation killing criteria. Some key components of the framework are introduced in Section IV-A and the experiment setup for this work is described later in Section IV-B.

A. Mutation Testing Framework

The overall framework of our Mutation Testing tool is illustrated in Fig. 3. It takes a program under test (PUT) and associated test suite as input, as well as a configuration file containing paths to the source files and other parameters. After generating mutants and testing them against the test suite, the mutation report for each mutant is generated, which contains the mutated point with the Mutation Operator applied, whether the mutant has survived or is killed, and a list or criteria that kill it if it is killed.

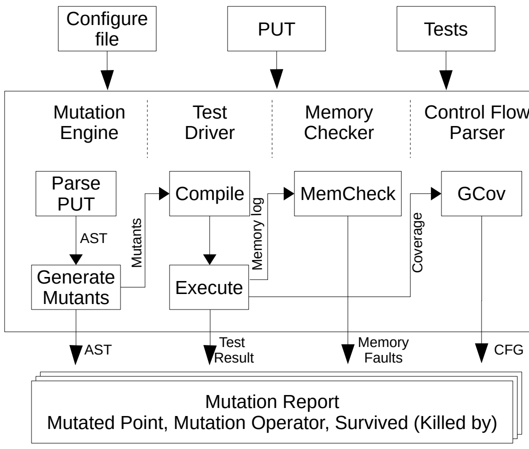

Fig. 3. Mutation Testing Tool work flow. It takes a program under test, a test suite and a configure file as input, and ourputs the mutation report for each mutants generated.

The mutation engine in our tool was developed using the TXL source transformation language [28]. TXL is a functional language, which provides a rule-based way of traversing and mutating the generated abstract syntax tree of a source file. One big advantage of using TXL is that the mutations are guaranteed to be safe, due to the fact that the transformational rules are confined by a grammar which guarantees the transformations will always be compilable and abide by the grammar of the language. Another advangate of TXL is that it has a very simple rule based transformation definition system that allows users to quickly develop their own rules without knowing the lower level details. However, in comparison to another widely used platform, Clang/LLVM, which is used by many other mutation engines, TXL is slower for Mutation Testing due to repeated source file parsing, once per mutant [14]–[19]. To overcome this problem, we develop a driver that allows generation of all mutants for a given source file in a single parse of the source file (similar to a meta-mutant [29]). We leave the comprehensive comparison of efficiency between TXL and Clang/LLVM as future work.

After the mutants are generated, some basic mutation information including the mutated point and the Mutation Operator applied are stored as part of the mutation report for each mutant. The test driver then instruments the mutants and compiles them, while those that fail to compile are labeled as killed. After the compilation, all the mutants are executed against the test suite, which generates three independent results: the test result indicating whether a mutant passes the regression test, the memory log including potential memory faults and the coverage information.

To detect memory faults, the mutants are executed by *Valgrind*'s *memcheck* tool [26] which generates the memory fault report for each mutant. The control flow information is gathered by using *Gcov* [30] to generate the control flow graph and the control flow coverage for each mutant. All of this information is summarised in a single mutation report for each mutant as the output of our tool.

B. Experiment Setup

With the framework introduced in Section IV-A, we applied MeMOs to 16 "real world" C programs that implement unit tests using the CuTest C testing framework [31] (Table VIII).

TABLE VIII. SUBJECT PROGRAMS UNDER TEST

PUT NO.	Program	LoC
1	PeerWireProtocol	1547
2	Craft	731
3	CfixedArraylist	497
4	ChashMapViaLinkedList	488
5	CAVLTree	405
6	CpseudoLRU	384
7	CHashMapViaQuadraticProbing	1097
8	CtextureAtlas	745
9	Csplaytree	834
10	CstreamingBencodeReader	371
11	CSparseCounter	328
12	Cheap	207
13	CcircularBuffer	118
14	ClinkedListQueue	200
15	CbipBuffer	118
16	Cbitfield	87

PeerWireProtocol, a C based implementation of the *BitTorrent* peer wire protocol, is the largest program (based on lines of code) of the set. This program facilitates the exchange of file blocks between peers over the wire in a P2P Bittorrent system. The program also handles message flows, handshakes and manages client negotiations. *CRaft* is a C based implementation of the Raft Consensus protocol which implements the Raft Consensus Algorithm in order to manage multiple fault-tolerant distributed systems. *CpseudoLRU* implements the Least Recently Used (LRU) caching algorithm and *CstreamingBencodeReader* is a library for reading and manipulating bencoded data, which is the encoding used by the peer-to-peer file sharing system *BitTorrent* for storing and transmitting loosely structured data. The rest of the programs are implementations of common memory intensive data structures whereas.

The main reason for choosing these programs is the fact that they are non-trivial real-world programs ranging from 87 to 1547 lines of code, making them of manageable size for our simple proof-of-concept research tool. Moreover, they are memory intensive, making extensive use of memory based operators. Additionally, these programs come with real-world test suites. All of the subject programs can be found in GitHub repositories.

After we generated mutants from these subject programs, we use both, the traditional killing criterion and additionally the Memory Fault Detection and Control Flow Deviation criteria to kill the mutants.

V. Results

In this section, we answer the Research Questions one by one using the experiment results.

A. Characteristics of MeMOs

To answer **RQ1**, we need to answer the sub-questions from three aspects.

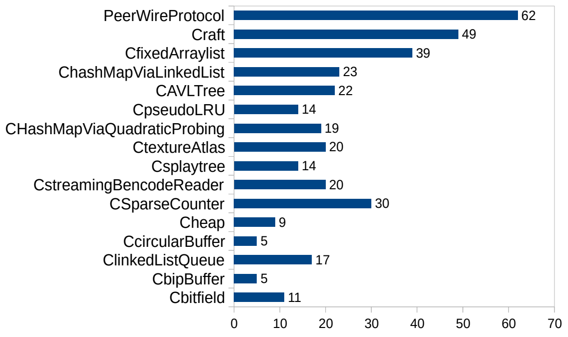

Fig. 4. Number of generated Memory Mutants from each subject program

We gathered the number of mutants generated from each of the subject programs and report it in Fig. 4. According to the figure, the MeMOs generate a considerable number of mutants, ranging from 5 to 62 for each subject program. For half of the programs, MeMOs generate more than 20 mutants. So the answer to **RQ1a** is that the Memory Mutation Operators are effective in injecting memory faults into programs.

Specifically, we want to understand about how effective each proposed Memory Mutation Operator is in injecting memory faults. The number of mutants generated by each Memory Mutation Operator is reported in the third column in Table IX. The numbers of generated mutants from all Memory Mutation Operators are comparable (Except **REMSOTP**), indicating the operators are similarily effective in injecting memory faults. Recall that **REMSOTP** only applies to the unary operator *sizeof()* when the argument is a data type pointer and in most cases it is a data type. Therefore it is expected that **REMSOTP** generates fewer mutants than its counterpart **RESOTPE**. So the answer to **RQ1b** is that all MeMOs are equally effective in injecting memory faults, while **REMSOTP** is slightly less effective than others.

TABLE IX. Number of generated mutants and survived mutants for each operator

Category	Mutation Operator	Generated Mutants	Survived Mutants	Mutation Score
Uninitialized Memory Access	REC2M	30	25	0.167
	RMNA	39	21	0.462
Faulty Memory Allocation	REDAWN	65	12	0.815
	REDAWZ	63	35	0.444
	RESOTPE	48	28	0.417
	REMSOTP	5	5	0.000
Faulty Heap Management	RMFS	53	53	0.000
	REM2A	27	16	0.407
	REC2A	29	6	0.793
All		359	201	0.440

To answer **RQ1c**, we calculate how many mutants survived under the traditional killing criterion (comparing test outputs) for each Memory Mutation Operator. The number of survived mutants and the Mutation Score are reported in Column 4 and 5 respectively in Table IX. From the table we can see that for 7 out of 9 operators the Mutation Scores are below 0.5. The highest Mutation Score is 0.815, while the lowest is 0 – meaning that none of the mutants generated by two operators (5 for **REMSOTP**, 53 for **RMFS**) were killed. The answer to **RQ1c** is that most of the MeMOs generate more than 50% mutants that cannot be killed by applying the traditional killing criterion, with a lowest Mutation Score of 0. There are quite a lot survived mutants with respect to MeMOs, which may severely bias the Mutation Score.

B. *Reduction of Survived Mutants*

In order to answer **RQ2**, we calculate how many survived mutants can be killed by either the Memory Fault Detection or the Control Flow Deviation killing criterion. We report the reduction rate of survived mutants in Fig. 5. In the graph, each 100% bar represents all the survived mutants for each Memory Mutation Operator in terms of the traditional criterion, or $\overline{T}$ refering to the Venn Diagram in Fig. 2. The darker part represents the ratio of these mutants killed by the Memory Fault Detection and/or Control Flow Deviation criteria, or $R = \overline{T} \cap (M \cup C)$. The percentage number on the dark bars are calculated as $\frac{|R|}{|\overline{T}|}$.

From the graph, we can see there is a reduction rate of more than 60% for 4 out of 9 MeMOs, after introducing Memory Fault and Control Flow Deviation killing criteria. The least reduction rate is 10% while the biggest reduction rate is as high as 75%. So the answer to **RQ2** is, Memory Fault and Control Flow Deviation criteria are effective in reducing the number of survived mutants with the highest reduction rate of 75% for operator **RESOTPE**.

C. *Contribution of MF and CFD*

We calculate $c_{\text{MFD}} = \frac{|M_{\text{only}}|}{|R|}$ and $c_{\text{CFD}} = \frac{|C_{\text{only}}|}{|R|}$ for each of the subject programs and report them in Table X. The value of c_{MFD} represents how much the MFD criterion exclusively contributes to the reduction of survived mutants and is always in the range of $[0, 1]$. When $c_{\text{MFD}} = 0$, the MFD criterion contributes none, since all the mutants, if any, killed by MFD can also be killed by CFD ($M_{\text{only}} = \emptyset$), on the other hand, if $c_{\text{MFD}} = 1$, all of the mutants killed by MFD can only be killed by MFD, indicating that CFD contributes none. Similarly, the larger c_{CFD} is, the more the CFD criterion contributes to the reduction of survived mutants. Specially, $c_{\text{MFD}} + c_{\text{CFD}} \leq 1$ and $1 - (c_{\text{MFD}} + c_{\text{CFD}})$ is the ratio of mutants killed by both criteria. **RQ3** can be answered by comparing c_{MFD} and c_{CFD} for all of the PUTs.

TABLE X. Contribution of MF and CFD in reducing survived mutants

PUT NO.	Number of mutants				c_{MFD}	c_{CFD}
	survived	killed by MF/CFD	killed by MF only	killed by CFD only		
1	37	9	4	2	0.444	0.222
2	29	29	25	0	0.862	0
3	37	6	6	0	1	0
4	8	2	1	1	0.5	0.5
5	14	5	5	0	1	0
6	4	1	0	1	0	1
7	3	0	0	0	0	0
8	8	8	8	0	1	0
9	3	1	0	1	0	1
10	6	4	4	0	1	0
11	22	13	13	0	1	0
12	3	0	0	0	0	0
13	3	2	2	0	1	0
14	12	6	4	2	0.667	0.333
15	3	2	2	0	1	0
16	9	3	2	1	0.667	0.333
All	201	91	76	8	0.835	0.088

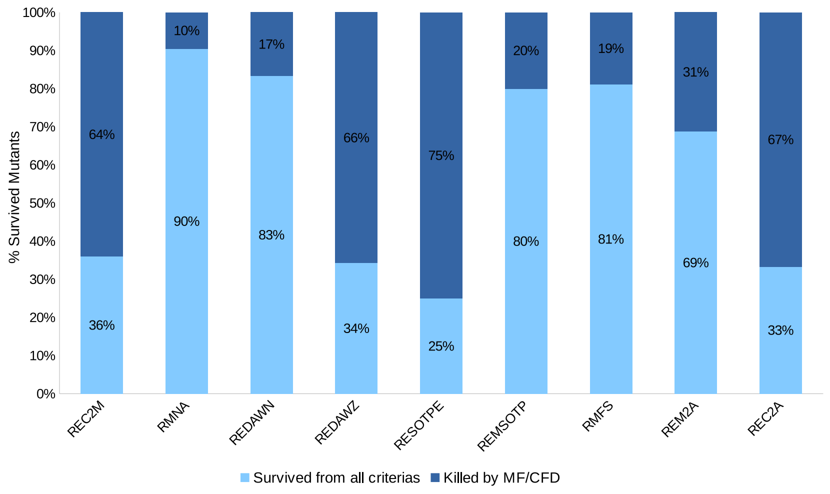

Fig. 5. Reduction of equivalent mutants after introducing Memory Fault Detection and Control Flow Deviation killing criteria. The percentage of the darker bars is calculated as “number of mutants survived from strong killing criterion but killed by MFD and/or CFD” divided by ”number of mutants survived from strong killing criterion”.

From Table X we can see that for most of the cases c_{MFD} is bigger than c_{CFD} ($c_{\text{MFD}} = 0.835$ and $c_{\text{CFD}} = 0.088$ for all subjects), indicating that the MFD criterion is more “powerful”, i.e. it reduces more survived mutants. This is not surprising since these mutants are generated by inserting memory faults by MeMOs. Since both MFD and CFD criteria can be applied to traditional Mutation Testing as well, whether MFD is more powerful than CFD in traditional Mutation Testing remains a question for future work. In our experiments, even though c_{MFD} is generally bigger than c_{CFD}, on 6 PUTs c_{CFD} is larger than 0 with in total 8 mutants only killed by CFD, suggesting that the CFD criterion is still useful in reducing survived mutants since $c_{\text{CFD}} > 0$ suggests that there are some mutants that can only be killed by the CFD criterion.

VI. Threats to Validity

A. Internal Validity

In this paper, we use *Valgrind* and *Gcov* to instrument the PUTs to gather memory fault and control flow coverage information during the execution. Whether these two instrumentations affect each other remains untested. This could be an internal threat to the validity of the reduction rate of survived mutants. Another threat is the execution environment. The programs and the mutants are run in *Valgrind*’s virtual environment, which may differ from a real environment. This could bias the number of survived mutants with respect to each killing criterion and is worth of investigation in the future.

B. External Validity

All of the conclusions are drawn from the PUTs ranging from small sizes to moderately large sizes. But they don’t garantee to hold on very large programs. Recall that most the MeMOs apply only on *malloc*/*free* routines, the number of mutants generated must relate to how frequently a program uses the routines. This could lead to a threat to the conclusion that MeMOs effectively insert memory faults to the subject programs in terms of the number of generated mutants. From another perspective of view, if a program uses dynamic memory allocation less frequently, the program is less likely to have memory faults and needs memory related Mutation Testing less, thus the effect of this threat is reduced. The contribution of MFD and CFD criteria reported in this paper is only applicable on the 9 Memory Mutation Operators. It is likely the values of their contribution will be different when applied to other traditional Mutation Operators. This is merely a threat since it is also interesting to see how these criteria perform in detecting memory-specific faults.

VII. Related Work

Shahriar [32] proposed 12 mutation operators which primarily focus on Buffer Overflow vulnerabilities, vulnerable library functions and program statements. The tool which they developed uses mutation testing to generate test cases that expose vulnerabilities in the program under test. Their proposed mutation operators also suggest that they are interested in identifying faults that aim to identify a programmers lack of fundamental knowledge about the programming language instead of identifying trivial syntactic faults. However, the proposed operators in this study do not consider vulnerabilities that are caused due to uninitialized memory access such as *NULL* pointer dereferencing or memory leaks caused due to faulty heap management, while we focus on and mutate the heap management routines in this work. They also proposed some killing criteria aiming at buffer overflows, but in our work, using *Valgrind* is a more general way to detect memory faults including buffer overflows.

Work done by Vilela [33] was an inspiration behind the mutation operators that mutate the parameters of *malloc()*, *calloc()* and other memory allocation/deallocation library functions as well. Static memory allocations (MSMA) and Dynamic memory allocations (MDMA) are proposed in this paper, each of which mutate the buffer size in order to identify buffer overflow and buffer underflow vulnerabilities. Although their proposed mutation operators mutate memory related operations, they do not expose vulnerabilities that are caused due to uninitialized memory faults like we propose in this project as well as buffer vulnerabilities that can be caused due to an incorrect argument to the *sizeof()* function generally used in memory allocation.

Although Zhivich [34] use a mutation approach to identify memory based faults as well as integrate dynamic memory analysis tools in their work, they primarily focus on buffer overflow vulnerabilities and do not consider the other classes of vulnerabilities that we consider in this work. Also, their work is mainly concerned with using mutation to mutate test data and use code instrumentation along with dynamic memory analysers to identify vulnerabilities, in comparison to using mutation as a way of injecting faults into the source code itself in our work. Moreover, due to the fact they do not perform any source code transformation and much of their work is based on dynamic analysis of the program under test, their work can be classified as taking a more black-box testing approach in comparison to the white-box testing approach that is taken in this project.

Kosmatov [35] developed a runtime memory monitoring library for runtime assertion checking in Frama-C [26], which is a platform for analysis of C code. Their work aims to detect dynamic memory related faults such as invalid pointers, out-of-bounds memory accesses, unitialized variables and memory leaks. *Valgrind* is also used in this work, though only as a benchmark to compare results. Although this study aims to detect the same classes of memory faults we do in this project, this study does not use mutation operators to inject such vulnerabilities. Instead, mutation operators that mutate numerical arithmetic operators, pointer-arithmetic operators and comparison operators are used to generate faults that may or may not introduce memory based vulnerabilities to be picked up by the runtime memory analyser. Their work also acknowledges the fact that memory related faults are more likely to occur in C programs due to the lack of infrastructure available to detect them.

Interesting research such as a detailed evaluation of seven modern dynamic buffer overflow detection tools was also carried out by Zhivich [34]. They end up using *CRED* (C Range Error Detector) in order to analyse buffer overflow faults over *Valgrind*'s *Memcheck*, which is the tool we use in our study, mainly because *Memcheck* uses sampled bound checking to detect buffer overflow errors instead of more precise bound checking techniques in *CRED*. They also found that *Valgrind* runs 25-50 times slower than gcc due to the fact it simulates program execution on a virtual x86 processor. Although they observed a speed issue with using *Valgrind*, *Valgrind*'s *Memcheck* is used as the primary memory analysis tool in this project due to its ability to analyse a range of other memory faults as well as its ability to handle faults due to uninitialized memory especially well in comparison with other alternative tools available.

VIII. Conclusion

Mutation Testing is notably good at testing the adequacy of a test suite. However, traditional Mutation Operators only mimic general faults while memory faults may be missed by these operators. In this paper we propose 9 Memory Mutation Operators (MeMOs) simulating different kinds of memory faults. By applying MeMOs to 16 real world programs, we find that these operators are effective in inserting memory faults in the subject programs, but the traditional Mutation Testing (comparing the output of a mutant against the original) achieves relatively low Mutation Score. We find that richer information can be used to reduce the number of survived mutants. We collect the memory faults and the control flow graph coverage of mutants and of the original program and use this information to reduce the survived mutants that survive the traditional strong killing criterion, proposing two new Memory Fault Detection and Control Flow Deviation killing criteria. The experimental results show that introducing these two killing criteria can further reduce the survived mutants by up to 75%. Furthermore, we summarise the number of survived mutants (from traditional Mutation Testing) that can only be killed by MFD or CFD criterion. The results suggest that both criteria are useful in reducing the number survived mutants, but no criterion subsumes the other.

Acknowledgment

Fan Wu is sponsored by Chinese Scholarship Council. Mark Harman is supported by the UK EPSRC project EP/J017515 (DAASE) and the UK EPSRC projects EP/I033688 (GISMO). Yue Jia is supported by the Microsoft Azure Grant 2014.

References

[1] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," in *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '80. New York, NY, USA: ACM, 1980, pp. 220–233. [Online]. Available: http://doi.acm.org/10.1145/567446.567468

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

[3] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic diagnosis and response to memory corruption vulnerabilities," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: ACM, 2005, pp. 223–234. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102151

[4] M. Rinard, C. Cadar, D. Dumitran, D. Roy, and T. Leu, "A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors)," in *Computer Security Applications Conference, 2004. 20th Annual*, Dec 2004, pp. 82–90.

[5] E. C. Sezer, P. Ning, C. Kil, and J. Xu, "Memsherlock: An automated debugger for unknown memory corruption vulnerabilities," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 562–572. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315314

[6] S. Christey, M. Brown, D. Kirby, B. Martin, and A. Paller, "CWE/SANS top 25 most dangerous software errors," 2011, http://cwe.mitre.org/top25.

[7] A. T. Acree Jr, "On mutation." DTIC Document, Tech. Rep., 1980.

[8] T. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982. [Online]. Available: http://dx.doi.org/10.1007/BF00625279

[9] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification and Reliability*, vol. 4, no. 3, pp. 131–154, 1994. [Online]. Available: http://dx.doi.org/10.1002/stvr.4370040303

[10] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165–192, 1997. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U

[11] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379 – 1393, 2009, source Code Analysis and Manipulation, SCAM 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584909000688

[12] K. Adamopoulos, M. Harman, and R. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Genetic and Evolutionary Computation GECCO 2004*, ser. Lecture Notes in Computer Science, K. Deb, Ed. Springer Berlin Heidelberg, 2004, vol. 3103, pp. 1338–1349. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24855-2_155

[13] E. Miller and W. E. Howden, *Tutorial: software testing & validation techniques*. IEEE Computer Society, 1978, vol. 138, no. 8.

[14] H. Dan and R. Hierons, "SMT-C: A semantic mutation testing tools for C," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, April 2012, pp. 654–663.

[15] C. Roy and J. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, April 2009, pp. 157–166.

[16] B. Kam and T. Dean, "Linguistic security testing for text communication protocols," in *Testing Practice and Research Techniques*, ser. Lecture Notes in Computer Science, L. Bottaci and G. Fraser, Eds. Springer Berlin Heidelberg, 2010, vol. 6303, pp. 104–117. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15585-7_10

[17] J. Bradbury, J. Cordy, and J. Dingel, "Exman: A generic and customizable framework for experimental mutation analysis," in *Mutation Analysis, 2006. Second Workshop on*, Nov 2006, pp. 4–4.

[18] A. Dereziska and A. Szustek, "Object-oriented testing capabilities and performance evaluation of the C# mutation system," in *Advances in Software Engineering Techniques*, ser. Lecture Notes in Computer Science, T. Szmuc, M. Szpyrka, and J. Zendulka, Eds. Springer Berlin Heidelberg, 2012, vol. 7054, pp. 229–242. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28038-2_18

[19] S. Zhang, T. Dean, and S. Knight, "A lightweight approach to state based security testing," in *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '06. Riverton, NJ, USA: IBM Corp., 2006. [Online]. Available: http://dx.doi.org/10.1145/1188966.1189004

[20] M. Kusano and C. Wang, "CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 722–725.

[21] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: A mutation system for java," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 827–830. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134425

[22] H. Riener and G. Fey, "FAuST: A framework for formal verification, automated debugging, and software test generation," in *Model Checking Software*, ser. Lecture Notes in Computer Science, A. Donaldson and D. Parker, Eds. Springer Berlin Heidelberg, 2012, vol. 7385, pp. 234–240. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31759-0_17

[23] Y. Jia and M. Harman, "MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language," in *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference*, 2008, pp. 94–98.

[24] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 158–168. [Online]. Available: http://doi.acm.org/10.1145/1133981.1134000

[25] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *In Proc. of the Winter 1992 USENIX Conference*, 1991, pp. 125–138.

[26] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746

[27] K. Androutsopoulos, D. Clark, H. Dan, R. M. Hierons, and M. Harman, "An analysis of the relationship between information squeeziness and failed error propagation in software testing," *RN*, vol. 13, p. 18, 2013.

[28] J. R. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190 – 210, 2006, special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 04). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642306000669

[29] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '93. New York, NY, USA: ACM, 1993, pp. 139–148. [Online]. Available: http://doi.acm.org/10.1145/154183.154265

[30] A. Griffith, *GCC: The Complete Reference*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2002.

[31] N. Gould, D. Orban, and P. Toint, "Cutest: a constrained and unconstrained testing environment with safe threads for mathematical optimization," *Computational Optimization and Applications*, pp. 1–13, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10589-014-9687-3

[32] H. Shahriar and M. Zulkernine, "Mutation-based testing of buffer overflow vulnerabilities," in *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, July 2008, pp. 979–984.

[33] P. Vilela, M. Machado, and W. Wong, "Testing for security vulnerabilities in software," *Software Engineering and Applications*, 2002.

[34] M. A. Zhivich, "Detecting buffer overflows using testcase synthesis and code instrumentation," Ph.D. dissertation, Massachusetts Institute of Technology, 2005.

[35] N. Kosmatov, G. Petiot, and J. Signoles, "An optimized memory monitoring for runtime assertion checking of C programs," in *Runtime Verification*, ser. Lecture Notes in Computer Science, A. Legay and S. Bensalem, Eds. Springer Berlin Heidelberg, 2013, vol. 8174, pp. 167–182. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40787-1_10