Eris: A Tool for Combinatorial Testing of the Linux System Call Interface

Bernhard Garn
SBA Research
Vienna, 1040, Austria
Email: bgarn@sba-research.org

Dimitris E. Simos
SBA Research
Vienna, 1040, Austria
Email: dsimos@sba-research.org

***Abstract*—In this paper, we show the applicability of combinatorial testing to the system call interface of the Linux kernel. Our approach is two-fold: first we analyze the TRINITY fuzz tester and in the aftermath we adapt the input parameter modeling of TRINITY to the field of combinatorial testing. Furthermore, apart from the modeling itself, we target to provide a configurable testing framework for executing tests obtained by the ACTS combinatorial test generation tool, called ERIS.**

***Keywords*—Combinatorial Testing, Linux, Fuzz testing**

I. INTRODUCTION

This paper highlights the research results obtained so far about developing combinatorial design testing for the system call interface offered by the Linux kernel [1]. We strive to provide the modeling and to design a combinatorial testing framework for application programming interfaces (APIs). In this case study, we focus on testing the system call interface of the Linux kernel. We present two modeling concepts for the system call API of the Linux kernel: input parameter modeling based on category partitioning and flattening methodology. Fuzz testing the Linux system call interface has discovered a lot of bugs for some time now [2]. The inherent problem of fuzz testing is the unpredictability of any assertions about the achieved coverage in respect to the total input space. With the help of combinatorial testing, we hope to propose an alternative which offers concrete assertions about the interaction coverage, enhancing the trust one can place into a given API. We note that by testing the Linux system call interface, one can also consider testing security aspects of the underlying operating system. Although combinatorial testing has been applied in a lot of different scenarios (see Section V), to the best of our knowledge, we are not aware of any work focusing only on testing the Linux system call API.

The schematic design overview of the proposed testing framework is depicted in Figure 2.

A. *Trinity and Eris*

Testing is an integral part of software engineering. There are different conceptual ways to approach software testing. During its lifetime a software is most likely tested at least against the following primitives: correctness, performance, usability and regressions. The adherence to all of these goals can be evaluated by different designs of tests, for example there are unit-tests, integration tests, white-box tests, black-box tests, fuzz testing and combinatorial testing. A nice overview of all these testing methodologies can be found in [3]. The immense inherent complexity of today's highly interconnected software systems nearly always excludes the practicability of testing all possible combinations, i.e. using an exhaustive search on the input space as input to the testing procedures.

Fuzz testing aims to challenge software in unexpected ways by passing randomly generated values to parameters. The software TRINITY is a fuzz tester for the Linux system call interface, which uses sanitized random parameter values (see [4]) as input for system calls. A test oracle is provided by the simple fact whether the kernel crashes or not due to the execution of test cases.

So far, TRINITY has found quite an impressive list of bugs [2]. TRINITY is primarily developed by Dave Jones in the C programming language. TRINITY currently supports more than 10 processor architectures. For more details refer to [5].

In this paper, we present a new configurable combinatorial testing framework, called ERIS, which executes generated test cases obtained by the **ACTS** combinatorial test generation tool [6]. **ACTS** is developed jointly by the US National Institute of Standards and Technology and the University of Texas at Arlington, and currently has more than 1400 individual and corporate users. ERIS is written in the C programming language and Shell scripts, and is based on TRINITY. ERIS also integrates the underlying input parameter modeling for the system call API and takes care of translating the modeled parameters to actual values. The features of ERIS that are presented in this paper for the first time are built, apart from the actual execution of system calls, upon a modular design which makes it possible to replace some of its components, including the test case generation and the test oracle, as can be seen in Figure 2.

Currently we are also using the crash oracle in ERIS. However, ERIS is designed in such a way that this component can be configured to comply with different oracles, especially the ones that make use of semantics.

B. *Motivation*

The fact that there already is available data about bugs found by fuzz testing (especially from TRINITY) gives an opportunity to compare it with data generated with our approach based on combinatorial design test procedures. We hope to be able to confirm the earlier findings about the interactions of software stages when the ERIS framework reaches future completion. Furthermore, according to [7], a study about the

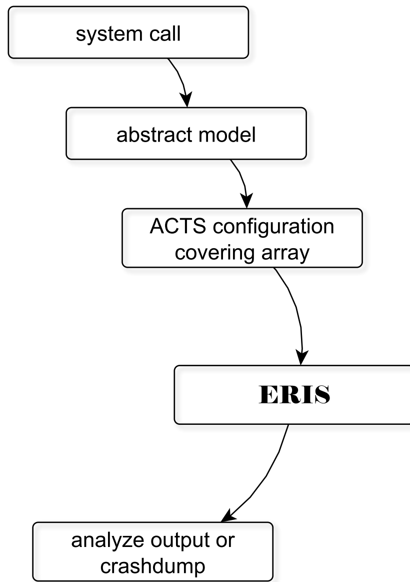

Fig. 1. Testing procedure

Linux kernel, or another prominent example of free and open source software, has not been done yet. The Linux kernel code is actively used today (e.g. 1.5 million Android devices activated per day in April 2013 according to Google) and has a sufficiently large code base to be considered a plausible real-world example.

As the kernel of an operating system is also the central authority to enforce and control security, as much testing as possible is crucial for the safety of our daily tasks, and more tests can only confirm trust in the code or uncover bugs, which will hopefully be fixed afterwards.

Furthermore, system calls often involve flag arguments, i.e. (often binary-) switches that control the behavior of the call, which can easily be modeled. Likewise, the namespaces subsystem, especially the creation of new namespaces, relies on a lot of flags, which control different aspects. A complete testing of all combinations is nearly impossible, therefore our combinatorial testing approach promises good results.

Some remarks regarding special functionality of the Linux kernel: no namespace-hierarchies are considered, i.e. we consider living in the root namespace, and some other features are also not taken into account (for example restricting the kexec system call) for this modeling.

C. Contribution

Our contribution can be summarized in the following three concrete case studies. Firstly, we conducted an extensive analysis of the TRINITY fuzz tester. In particular, we analyzed in great length the system calls executed by this fuzz tester which formed the basis for a classification suitable for a broader combinatorial modeling of APIs. Secondly, we present two methodologies based on input parameter modeling with category partitioning specifically for modeling system calls. In the aftermath, this combinatorial modeling of APIs was integrated into a new configurable framework presented for the first time in this paper, called ERIS, which uses the **ACTS** testing tool for producing test cases and extends the TRINITY fuzz tester by offering additional functionalities such as combinatorial testing.

This paper is structured as follows: in Section II we briefly give the necessary definitions from the field of combinatorial testing that are needed for our approach. In Section III we present two approaches for combinatorial modeling of APIs. In Section IV we present some implementation details and a proof-of-concept for the new ERIS testing framework. Finally, in the concluding section we summarize our work and raise some questions for further work.

II. BACKGROUND

Testing a *system under test* (SUT) requires the existence of test cases, and in particular a method capable of generating such test cases. For developing our ERIS testing framework we can also use methods that arise from the field of combinatorial testing. Combinatorial testing is an effective testing technique to reveal errors in a given SUT, based on input combinations coverage. Combinatorial testing of strength t (where $t \geq 2$) requires that each t-wise tuple of values of the different system parameters is covered by at least one test case.

To design a test suite, the tester identifies possible output values from each of the actions of the SUT. It is important to find a test suite that is not too large, but yet tests for most of the interactions among the possible outputs in the actions of the SUT. Recently, some researchers [8], [9], [10] suggested that some faults or errors in SUTs are a combination of a few actions when compared to the total number of parameters of the SUT.

Combinatorial testing is motivated by the selection of a few test cases that comprise a test suite in such a manner that good coverage is still achievable. The combinatorial test design process can be briefly described as follows:

1) Model the input space. The model is expressed in terms of stages (parameters) and stage (parameter) values.
2) The model is input into a combinatorial design procedure to generate a combinatorial object that is simply an array of symbols.
3) Every column of the generated array is used to output a test case for a SUT.

One of the benefits of this approach is that steps 2. and 3. can be completely automated. In particular, we used the **ACTS** tool and in the aftermath the ERIS framework for these steps.

We followed this approach to model software testing procedures in terms of combinatorial testing methodologies, and in particular we modeled the system call interface of the Linux kernel. The input space of the possible system calls, as this will be later discussed in the paper in Section III, make the procedure of system call testing an ideal candidate for modelling via combinatorial designs. For our approach we use the notion of mixed-level covering arrays (a specific class of combinatorial designs). For the sake of completeness we provide below the definition of mixed-level covering arrays taken from [11] since this is the underlying generated structure in the **ACTS** tool:

Definition 1: A mixed-level covering array which we will denote as $\mathrm{MCA}(t, k, (g_1, \ldots, g_k))$ is an $k \times N$ array in which the entries of the i-th row arise from an alphabet of size g_i. Let $\{i_1, \ldots, i_t\} \subseteq \{1, \ldots, k\}$ and consider the subarray of size $t \times N$ by selecting rows of the MCA. There are $\prod_{i=1}^{t} g_i$ possible t-tuples that could appear as columns, and a MCA requires that each appears at least once. The parameter t is also called the strength of the MCA.

Lastly, we would like to note that examples of the exact modeling of system calls in terms of stages and stage values are given in the following section.

III. Modeling of APIs

Here we describe the process of translating a given API into a combinatorial model for combinatorial test design methodologies.

For the example of the Linux kernel system call interface we take the view of syntactical modeling. We use the term syntactical modeling similarly to purely syntactical constructions in mathematical logic. A priori, it more or less completely ignores semantic aspects. However, this approach does not exclude the possibility to introduce semantic aspects into the modeling and the testing framework.

1) System Call Interface of the Linux kernel: The Linux kernel supports many different processor architectures[1], among others i386 and x86-64. The system call interface provided by the Linux kernel is processor architecture specific.

2) From Source to Modeling: Trinity abstracts the C data types used in the Linux kernel source code to 18 argument types shown in Listing III-2. This definition can be found in `trinity/include/syscall.h` in the source tree of Trinity. Trinity is capable of intelligently fuzzing system call parameters using the classification in Listing III-2 of possible input parameter types. Trinity then further uses different random functions to generate actual values and then employs C data type specific as well as system call specific sanitizing functions.

```
enum argtype {
        ARG_UNDEFINED = 0,
        ARG_RANDOM_LONG = 1,
        ARG_FD = 2,
        ARG_LEN = 3,
        ARG_ADDRESS = 4,
        ARG_MODE_T = 5,
        ARG_NON_NULL_ADDRESS = 6,
        ARG_PID = 7,
        ARG_RANGE = 8,
        ARG_OP = 9,
        ARG_LIST = 10,
        ARG_RANDPAGE = 11,
        ARG_CPU = 12,
        ARG_PATHNAME = 13,
        ARG_IOVEC = 14,
        ARG_IOVECLEN = 15,
        ARG_SOCKADDR = 16,
        ARG_SOCKADDRLEN = 17,
};
```

List III-2: Abstract Parameter Types from Trinity

[1] https://www.kernel.org/linux.html

Under the term *actual parameter type* we view the type of a parameter of a system call in the sense of the C programming language in the sources of the Linux kernel. For example, a pointer has type `void*` in the sense of the C language. Similarly, the term *actual parameter value* refers to a value which is really passed to a system call and the term *abstract parameter type* is understood as an element given in Listing III-2 above. In our combinatorial modeling, we are considering each system call as an independent SUT, as it can be seen in Figure 1. Therefore, for a specific system call, for each of its specific actual parameter types, we assign exactly one abstract parameter, reusing the classification from Trinity. In Subsection III-B we present a direct modeling approach focusing on input parameter modeling with categories for the abstract argument types presented in Listing III-2. However, in terms of III-C one abstract parameter can be modeled with more parameters (stages) which capture more information, and this new approach is referred to as flattening methodology. Therefore our modeling, and also Eris, include a translation layer between abstract stage values and concrete stage values which is especially important in the flattening methodology described in Section III-C.

A. Input Parameter Model

Based on the type classification provided by Trinity, we employed the approach presented in [12] to model each type. The choices are specifically targeted at the environment the Linux kernel operates in. In this paper, we do not consider any network related system calls.

Input parameter modeling is not a new concept. For example, in [13] the authors present a three-step approach applying combinatorial testing to the Siemens Suite while in [14] the authors present the modeling of the input space of **ACTS**. In [15] common patterns that arise in combinatorial models are described. Moreover, in [16] a new two step input space modeling methodology is presented. In [9] research is focused on efficient fault characterization. In addition, in [17] the authors discuss categories and choices in the modeling process while in [18] comprehensive guidance on how to use combinatorial testing is provided. Finally, in [19] two new constructs in the modeling process are introduced, which reduce the overall complexity. For a thorough treatment of the topic we refer the interested reader to [20]. We would like to note that Eris is also enriched from the methodologies used for input parameter modeling given in [13].

We point out that some properties of the modeling, including some actual parameter values, are determined on runtime during the execution of a test case of a system call with Eris, for example valid PID numbers are generated when a program is started. The same holds for addresses of memory mappings. Therefore, sometimes placeholders have to be used in the modeling, which will get translated to actual parameter values during execution.

The overall outline during the execution of Eris is complicated by the fact that (intentionally or as a result of normal execution) even the number of available CPUs can change. Taking this into account, it follows that in order to target specific execution environments, it might be required to adapt some of the stage values in the modeling.

B. Combinatorial Modeling - IPM with Categories

Our first approach is based on the following classification of abstract argument types introduced in Listing III-2 in two disjoint classes: a **local** and a **global** type. Building upon that, system calls are then classified into two disjoint groups, which we also call **local** and **global**, according to the following rule: a system call is referred to as **global**, if and only if all of its abstract parameter types are classified as **global**. The remaining system calls are called **local**. This classification is based on the following observation: a stage value of type ARG_CPU can be passed to any system call that has an abstract argument type ARG_CPU. The same holds for a file descriptor, a path name or an ARG_MODE_T argument. In contrast to that, a flag in a system call is (almost surely) specific to that system call only, so this system call would be classified as **local**. Furthermore, a list of possible values in TRINITY is modeled with the type ARG_OP, and is considered as a **local** argument type in terms of our modeling based on IPM with categories. An example for a **global** system call is `chmod`, expecting a path and a mode argument, while an example for a **local** system call is `mount`, which relies heavily on flags.

The reason for this distinction lies in the fact that by restricting the modeling to system call independent abstract parameter types, it was possible to considerably reduce the amount of work needed during the modeling process. By intentionally considering only the **global** system calls, we are able to present a coherent modeling based on the input parameter modeling with categories. This was also straight forward to implement in the proposed ERIS testing framework. It should be mentioned that one can always add more invalid stage values, but in order to be able to present a clear methodology, we have limited the amount of invalid values.

1) Input Parameter Model with Categories for APIs: As explained above, the argument type ARG_CPU is considered a **global** abstract argument type. We create equivalence classes to partition the input space of possible CPU identifiers. Then we select some members out of each equivalence class and hence create with them a list of stage values. The number N of stage values will determine the **ACTS** configuration for this stage, precisely numbers from 0 to $N-1$. These numbers will then be used in a simple lookup operation to specify actual values.

The abstract argument type ARG_MODE_T corresponds to the actual parameter type mode_t used in the Linux kernel. Quoting[2] the beginning of `man 2 chmod`:

> The new file permissions are specified in mode, which is a bit mask created by ORing together zero or more of the following:
> S_ISUID (04000) set-user-ID (set process effective user ID on execve(2))

Therefore it follows that it can be considered as a list of 12 binary flags, which can be set independent from each other. This discrete input space is then partitioned in all possible 4096 singleton values. The input space for this abstract parameter type is clearly discrete and finite.

[2] http://man7.org/linux/man-pages/man2/chmod.2.html

A process in Linux can be identified by its Process Identifier (PID). When TRINITY has to generate an actual parameter value for a type ARG_PID, one possibility is that it selects a random PID from one of its running child processes. A given non-negative integer i can constitute a valid PID (meaning that there is currently a process running on the system which has that number i as PID) or an invalid PID. We are using 4 specific processes to get valid PID values. We selected two user processes, which have to be started manually beforehand: `w3m` and `ed`. Additionally, two system processes are chosen, namely `cron` and `acpid`. We use three invalid numbers: $-3, -1, 9999999999$ as placeholders. This makes a total of 7 stage values.

Regarding the abstract type ARG_ADDRESS, the memory model of the Linux kernel implies the following classes: NULL and KERNEL_ADDR. The other choices are taken from TRINITY. They represent a proper selection of possible addresses. It should be noted here that in particular the dereferencing of arbitrary memory addresses is not permitted, therefore some care has to be taken in the modeling of this abstract argument type.

On startup, TRINITY generates file descriptors from under `/proc`, `/sys` and `/dev` and then passes a random file descriptor out of this list to a system call. In contrast to that, on startup, a victim directory can be specified, which will be used as the source of file descriptors and pathnames. We are currently reusing this victim directory functionality from TRINITY in ERIS. Specifically, we generated a custom directory structure explicitly for testing purposes, comprising 15 files. This approach can be seen as a limited category partitioning. The determination of actual values in this regard remains open to further research, depending on practical evaluations. Another option available in ERIS for the generation of file descriptors is based on a customized `nftw` function.

TRINITY uses either existing paths, or creates mangled paths or strings that should appear as directories. For simplicity, we also reused the victim directory option of TRINITY for ARG_PATHNAME.

Remark 1: We would like to remark that the presented choices for the generation of file descriptors represent an instantiation of possible file descriptors. The option of using more file descriptors (possibly from many different directories) simply increases the coverage with respect to all files in the file system. The same holds for the abstract argument type ARG_PATHNAME. Furthermore, it follows from our modular design that it is possible to delegate the generation of path names to another entity, which could generate pathnames based on a grammar.

2) Examples for System Call Modeling using IPM with Categories: To present the modeling of a system call, we give below the **ACTS** configuration file for the `chmod` system call (the second list runs through all values between 0 and 4095, it is shortened for presentation here):

```
[System]
Name: chmod

[Parameter]
ARG_PATHNAME (int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
    10, 11, 12, 13, 14
```

TABLE I. ACTUAL STAGE VALUES FOR GLOBAL ABSTRACT PARAMETER TYPES

ARG_CPU	1, 2, 3, 4, 5, 6, 7, 8
ARG_MODE_T	1, 2, 3, 4, . . . , 4095, 4096
ARG_PID	$-3, -1$, *pid_cron*, *pid_acpid*, *pid_ed*, *pid_w3m*, 9999999999
ARG_ADDRESS	*NULL*, *KERNEL_ADDRESS*, *page_zeros*, *page_0xff*, *page_rand*, *page_allocs*, *get_map*(0), *get_map*(1), *get_map*(2), *malloc*
ARG_FD	$fd_1, fd_2, fd_3, \ldots, fd_{15}$
ARG_PATHNAME	$pathname_1, pathname_2, pathname_3, \ldots, pathname_{15}$

```
ARG_MODE_T (int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ... ,
    4095
```

Obviously we cannot exclude the possibility that for a given system call with more than two arguments when we want to generate covering array of strength greater than two, this computation might become intractable.

The actual stage values for ARG_PATHNAME are determined by the victim directory option in ERIS using our directory structure. The numbers in the configuration represent indices into arrays which hold the actual parameter values.

Now, consider the `renameat` system call, which has the following modeling in TRINITY and ERIS:

```
/*
 * SYSCALL_DEFINE4(renameat, int, olddfd, const char
       __user *, oldname,
     int, newdfd, const char __user *, newname)
 */
#include "sanitise.h"

struct syscall syscall_renameat = {
    .name = "renameat",
    .num_args = 4,
    .arg1name = "olddfd",
    .arg1type = ARG_FD,
    .arg2name = "oldname",
    .arg2type = ARG_ADDRESS,
    .arg3name = "newdfd",
    .arg3type = ARG_FD,
    .arg4name = "newname",
    .arg4type = ARG_ADDRESS,
    .flags = NEED_ALARM,
    .group = GROUP_VFS,
};
```

Based on the description above, the combinatorial modeling results in:

```
[System]
Name: renameat

[Parameter]
ARG_FD (int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
    12, 13, 14
ARG_ADDRESS (int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
ARG_FD (int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
    12, 13, 14
ARG_ADDRESS (int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

For example, the actual stage values for ARG_ADDRESS are given below.

```
/*
  init_arg_address sets the values from the function
       _get_address in file random-address.c into
     the array vals_arg_address
*/
void init_arg_address(void)
{
    vals_arg_address[0]="NULL";
    vals_arg_address[1]=(void *) KERNEL_ADDR;
    vals_arg_address[2]=page_zeros;
    vals_arg_address[3]=page_0xff;
    vals_arg_address[4]=page_rand;
    vals_arg_address[5]=page_allocs;
    vals_arg_address[6]=get_map(0);
    vals_arg_address[7]=get_map(1);
    vals_arg_address[8]=get_map(2);
    vals_arg_address[9]=malloc(page_size * 2);
}
```

As it can be seen, we give **ACTS** a list of indices, and the actual values are coded in arrays in ERIS' source code and the values obtained from **ACTS** are used as indices into these arrays.

3) Advantages and Disadvantages of our Input Parameter Modeling with Categories for APIs: One of the biggest advantages of this IPM approach is the direct translation between abstract and actual stages, which also implies a direct translation between abstract stage values and actual parameter values passed to system calls. The necessary configuration management in ERIS was directly build upon available configurations in TRINITY. This modeling further shows that the concept of input parameter modeling with category partitioning can be used to represent the abstract parameter types presented in Listing III-2. **ACTS**-configurations were created with Bash-scripts, parsing the source code of ERIS for the right identifiers.

However, this modeling approach was first implemented with hard-coded values in the source code. This fact makes the whole approach rigid and inflexible to adapt to new requirements. For the practical evaluation it will be beneficial to be able to quickly change some testing parameters during consecutive calls to our testing framework. For example, being able to use different strengths for the covering arrays. In the beginning of our study, we tried to use different covering arrays with hard-coded paths for each system call, which turned out to be impractical. Finally, the separation into the two classes **local** and **global** led to a very small list of system calls available to test when compared with all possible system calls - roughly about ten percent remained.

Remark 2: As a side remark we would like to mention that in this approach we did not need the constraint support of **ACTS**. The manual classification we performed into **local** and **global** system calls handled any possible constraints.

C. Combinatorial Modeling - A Flattening Methodology

Here we present a more flexible approach to the problem of modeling APIs, which also makes better use of some features of combinatorial testing based on the asymptotic growth of

covering arrays. For example, this growth is logarithmic on the number of stages.

The term flattening is to be understood as the process of using several stages for one abstract parameter type of a system call, encoding different properties of this abstract type in several stages. The types of these newly added stages will be adjusted to the necessities for each abstract argument type. If possible, we flatten one abstract type to many new stages. We would like to note that this is not done on stage values but on the stage types.

As a marginal note, we want to mention that with the introduction of the flattening methodology the difference between **global** and **local** system calls is not necessary anymore. The flattening methodology combined with a sophisticated policy management entity in the design of ERIS now allows us to also consider argument types like ARG_OP. The modeling of the latter argument type is firstly realized using category partitions and then it is possibly refined using the flattening methodology. Since the cardinalities of possible actual values in for example ARG_OP are often limited, it is feasible to integrate this abstract argument type into our framework. This gives us the possibility to model considerably more system calls.

Please observe that in this setting a translation layer is needed to convert a test case to actual parameter values, which are then used in ERIS. For further details see Section IV-B.

1) Detailed Description of Flattening Methodology: The abstract argument type ARG_LIST represents a list, where zero or more elements of this list can be bitwise-ored together. This abstract argument type is used to capture different flags. The used list entries are, generally speaking, system call specific and therefore it was not possible to model this argument type with the methodology presented in Section III-B. Lists are canonical examples to be flattened. In [15], the authors mention Multi-Selection, a similar approach to the flattening methodology. The flattening methodology for ARG_LIST is implemented as follows: for each list entry we add a binary stage. This stage decides whether a specific flag is set or not, similar to the identification of the power set $\wp(x)$ and ${}^{x}2$ (the set of all functions from x into 2). Again, a translation from the abstract test obtained by **ACTS** to an actual input value has to be implemented.

The flattening model obtained by our methodology for a system call which has an abstract parameter of type ARG_LIST, is given below:

$$\mathit{syscall}\,(\mathit{type}_1\ \mathit{arg}_1, \mathit{type}_2\ \mathit{arg}_2, \mathit{ARG_LIST}\ \mathit{arg}_3) \qquad (1)$$

The first and second abstract argument type in relation (1) are not further specified. The third parameter has abstract argument type ARG_LIST. Suppose that the respective list has N entries. Now let ν_1, ν_2 be actual parameter values for their respective types. In addition to them we now choose binary flags $(l_i)_{i=1,\ldots,N}$, i.e. $l_{(\cdot)} \in \{0, 1\}$. This results in the following modeling of the system call:

$$\mathit{syscall}\,(\nu_1,\, \nu_2,\, l_1, l_2, \ldots, l_N) \qquad (2)$$

The abstract argument type ARG_MODE_T can be viewed as a special instance of the type ARG_LIST with fixed entries. Therefore ARG_MODE_T will also be flattened out in the modeling. This means that we add 12 additional binary stages for each occurrence of an ARG_MODE_T abstract argument type of a system call. This has to be considered in test case generation and translation of tests into values, because the generated tests will have more stages than the system call has parameters. However, also in this methodology, one still has to decide about how many invalid stage values one wants to add. For example, by adding yet another boolean stage, which decides about the validity of the generated flag, and with the help of the constraint support in **ACTS**, similar to the approach in the case of ARG_ADDRESS below, it is possible to limit the number of added invalid stage values.

The problem with memory addresses in general is that it is mandatory to only read from initialized memory. The abstract argument type ARG_ADDRESS is used quite often in the modeling of system calls and is processor architecture dependent. TRINITY sometimes reuses addresses if a system call has more than one address parameter, which has proven to be useful. Under this consideration, the abstract argument type ARG_ADDRESS is **local** in terms of Section III-B.

The addresses that TRINITY generates are sometimes incremented by values like *page_size* $-$ *sizeof(int)* to find off-by-one-errors. Now we give a description of how to replicate these advanced address transformations in our combinatorial modeling. The flattening methodology for ARG_ADDRESS reads as follows:

- address value of the actual pointer, one actual value taken from the IPM with categories, called x
- updownmunge: ternary stage: 1 means change upwards, 0 means unchanged, -1 means change downwards, called y
- size: value of how much to change the pointer: $\{\mathit{sizeof(int)}, \mathit{sizeof(long)}, \mathit{sizeof(char)}\}$, called z

The translation into actual parameter values *addr* is according to the following relation:

$$\mathit{addr} = x + y \times z \qquad (3)$$

In the case $y = 0$ the values of z are clearly irrelevant, since they will all produce the same result according to relation (3). We are therefore utilizing the constraint support of **ACTS** in the following manner:

$$y = 0 \Rightarrow z = \zeta \qquad (4)$$

where ζ is a fixed element out of the actual possible values for z. This guarantees that **ACTS** will produce only a single test for that.

Another thing to consider is the fact that system calls often expect to find something (i.e. a structure of a specific type) at an address. We therefore want to emphasize again the fact that we are thinking of the address modeling in terms of syntactical modeling, independent from semantic modeling.

2) Examples for System Call Modeling using the Flattening Methodology: As an example resulting from the flattening methodology, we revisit again the `chmod` system call.

```
[System]
Name: chmod

[Parameter]
filename (int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 13, 14
S_ISUID (boolean): TRUE, FALSE
S_ISGID (boolean): TRUE, FALSE
S_ISVTX (boolean): TRUE, FALSE
S_IRUSR (boolean): TRUE, FALSE
S_IWUSR (boolean): TRUE, FALSE
S_IXUSR (boolean): TRUE, FALSE
S_IRGRP (boolean): TRUE, FALSE
S_IWGRP (boolean): TRUE, FALSE
S_IXGRP (boolean): TRUE, FALSE
S_IROTH (boolean): TRUE, FALSE
S_IWOTH (boolean): TRUE, FALSE
S_IXOTH (boolean): TRUE, FALSE
```

Now, we present the `mount` system call. The flattening methodology here leads to many stages. The abstract argument type ARG_ADDRESS is currently modeled with 3 stages and, referring to `man 2 mount`, we firstly present the modeling of `mount` from TRINITY, which is reused in ERIS:

```
/*
 * SYSCALL_DEFINE5(mount, char __user *, dev_name,
     char __user *, dir_name,
     char __user *, type, unsigned long, flags, void
         __user *, data)
 */

#include <linux/fs.h>
#include "sanitise.h"
#include "compat.h"

//TODO: fill out 'type' with something random from /
    proc/filesystems

struct syscall syscall_mount = {
    .name = "mount",
    .num_args = 5,
    .arg1name = "dev_name",
    .arg1type = ARG_PATHNAME,
    .arg2name = "dir_name",
    .arg2type = ARG_PATHNAME,
    .arg3name = "type",
    .arg3type = ARG_ADDRESS,
    .arg4name = "flags",
    .arg4type = ARG_LIST,
    .arg4list = {
        .num = 29,
        .values = {
            MS_RDONLY, MS_NOSUID, MS_NODEV,
                MS_NOEXEC,
            MS_SYNCHRONOUS, MS_REMOUNT, MS_MANDLOCK,
                MS_DIRSYNC,
            MS_NOATIME, MS_NODIRATIME, MS_BIND,
                MS_MOVE,
            MS_REC, MS_VERBOSE, MS_SILENT,
                MS_POSIXACL,
            MS_UNBINDABLE, MS_PRIVATE, MS_SLAVE,
                MS_SHARED,
            MS_RELATIME, MS_KERNMOUNT, MS_I_VERSION,
                MS_STRICTATIME,
            MS_SNAP_STABLE, MS_NOSEC, MS_BORN,
                MS_ACTIVE,
            MS_NOUSER, },
    },
    .arg5name = "data",
    .arg5type = ARG_ADDRESS,
    .group = GROUP_VFS,
};
```

This meta information about the `mount` system call is then translated according to the above mentioned flattening methodology into the following abstract stages given in the following illustrative configuration file of **ACTS** below:

```
[System]
Name: mount

[Parameter]
stage1_pathname (int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
    10, 11, 12, 13, 14
stage2_pathname (int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
    10, 11, 12, 13, 14
stage3_addr (int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
stage3_addr_updownmunge (int): -1, 0, 1
stage3_addr_size (enum): sizeof(int), sizeof(long),
    sizeof(char)
MS_RDONLY (boolean): TRUE, FALSE
MS_NOSUID (boolean): TRUE, FALSE
MS_NODEV (boolean): TRUE, FALSE
MS_NOEXEC (boolean): TRUE, FALSE
MS_SYNCHRONOUS (boolean): TRUE, FALSE
MS_REMOUNT (boolean): TRUE, FALSE
MS_MANDLOCK (boolean): TRUE, FALSE
MS_DIRSYNC (boolean): TRUE, FALSE
MS_NOATIME (boolean): TRUE, FALSE
MS_NODIRATIME (boolean): TRUE, FALSE
MS_BIND (boolean): TRUE, FALSE
MS_MOVE (boolean): TRUE, FALSE
MS_REC (boolean): TRUE, FALSE
MS_VERBOSE (boolean): TRUE, FALSE
MS_SILENT (boolean): TRUE, FALSE
MS_POSIXACL (boolean): TRUE, FALSE
MS_UNBINDABLE (boolean): TRUE, FALSE
MS_PRIVATE  (boolean): TRUE, FALSE
MS_SLAVE  (boolean): TRUE, FALSE
MS_SHARED (boolean): TRUE, FALSE
MS_RELATIME (boolean): TRUE, FALSE
MS_KERNMOUNT  (boolean): TRUE, FALSE
MS_I_VERSION (boolean): TRUE, FALSE
MS_STRICTATIME (boolean): TRUE, FALSE
MS_SNAP_STABLE (boolean): TRUE, FALSE
MS_NOSEC  (boolean): TRUE, FALSE
MS_BORN (boolean): TRUE, FALSE
MS_ACTIVE (boolean): TRUE, FALSE
MS_NOUSER (boolean): TRUE, FALSE
stage5_addr (int): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
stage5_addr_updownmunge (int): -1, 0, 1
stage5_addr_size (enum): sizeof(int), sizeof(long),
    sizeof(char)
```

IV. TESTING FRAMEWORK

A. Components

The general overview of the software design of the components of ERIS is depicted in Figure 2. The framework consists of three components, where one component takes care of the Setup, the Testing and the Analysis, respectively. In the Setup, once a specific system call is selected as SUT, a combinatorial model is used to derive a test suite with the help of **ACTS**. In the Testing step, the system call is executed with actual parameter values, which are created from the abstract test cases. The Analysis makes use of a test oracle, which automatically decides about the passing or failing of a single test or of the test suite as a whole. As can be seen from this high level description, the modular design of ERIS makes the adaption of some (or parts) of the described components quite easy to implement.

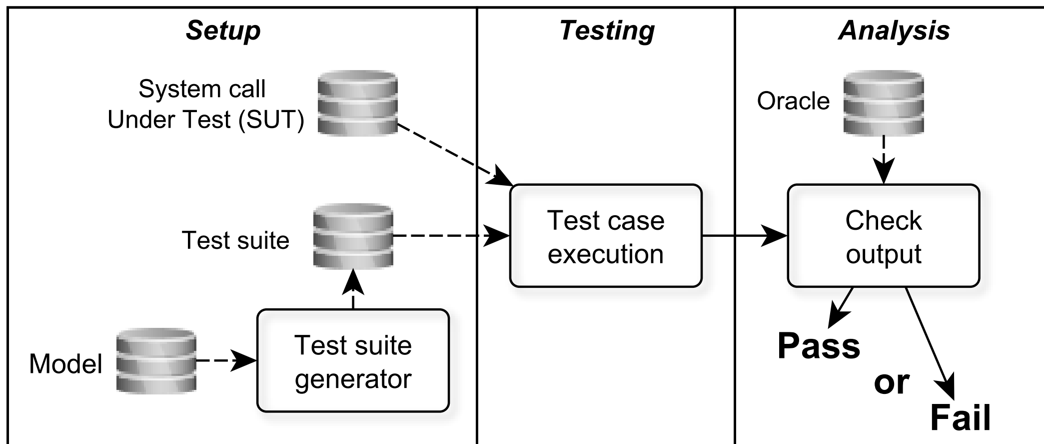

Fig. 2. General overview of components

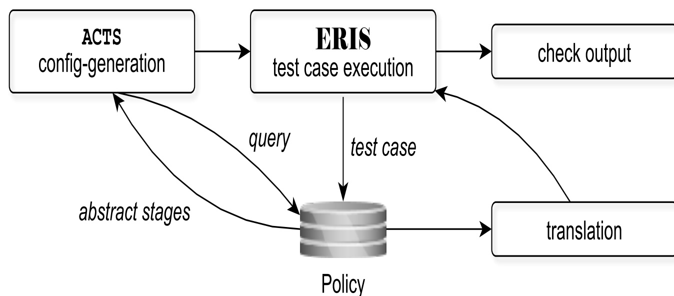

Fig. 3. Implementation Design of Eris

B. Implementation Details

The configurable testing framework ERIS is technically a fork of TRINITY, integrating all necessary logic to operate as a combinatorial tester. Our approach for IPM with categories can be directly translated into ERIS. The flattening methodology relies on a translation layer, which translates abstract tests into concrete tests. This approach is similar to [13], but we want to capture the flexible flattening approach in this translation layer. We are aiming to use a central authority, called policy, which should integrate not only the modeling, but also the configuration options for generating **ACTS** configuration files and used interaction strength. We believe a central managing authority is indispensable regarding the ability to, for example, change among predefined modeling schemes in an automatic manner. This is complicated by the fact that these changes might require a recompilation of ERIS, at least sometimes some changes will require the patching of ERIS' source code. For the concrete implementation we refer to Figure 3.

1) Trinity specific: Apart from the modeling, there are certain practical limitations which arise by using TRINITY as a starting point. Firstly, the TRINITY program is constantly being extended with new features, which entails work to integrate with new upstream versions. Apart from that, the implementation of TRINITY is not finished. For example, not all system calls that TRINITY handles have all their type information annotated. Therefore, such system calls have to be excluded, which leads to complicated parsing of TRINITY's source code. The extension of specific code for the arm architecture further complicates this task. As already mentioned, we did not consider any modeling of network related system calls or network related abstract argument types.

2) Eris: To provide for a most easy deployment of ERIS, we are currently building a complete, contained and independent system structure similar to a continuous integration testing deployment. Some aspects to consider are how the core testing of ERIS should be run in fully virtualized environments like virtualbox/kvm/xen or in containers (lxc). Moreover, the implementation ERIS is able to test each modeled system call per given Linux kernel version and per given interaction strength (on the abstract argument types). This is accomplished by calling ERIS in various wrapper scripts having as parameters the system call, the Linux kernel version and the interaction strength. Finally, ERIS has the functionality to use a top-level script capable of executing a complete test suite for system calls automatically.

C. Experiments

All of our conducted experiments were performed using **ACTS**-2.8 as part of ERIS. We are running the software primarily on the following system: `Ubuntu server` 12.04 with `openjdk-7-jdk`.

Assume the following cardinalities in the system call `renameat`, which has four abstract argument types: twice ARG_FD and twice ARG_ADDRESS. Using the presented methodology on input parameter modeling based on category partition of Section III-B1, ARG_FD has 15 stages and ARG_ADDRESS has 10 stages. Thus, the cardinality of the total search space is $15 \times 15 \times 10 \times 10 = 22500$. For this example we generated a covering array of strength 2 by **ACTS** in less than two seconds with size 229. Moreover, the array with strength 3 has size 2382.

We would like to note that test generation with **ACTS** was quite fast and overall we were impressed by its intuitive interface. However, to build a fully automated test framework, the command line interface was easier to use.

We run **ACTS** with the following calling in our scripts.

```
java <options> -cp acts_gui.jar edu.uta.cse.fireeye.
    console.ActsConsoleManager cmd <input_filename>
    <output_filename>
```

Finally, in the case of a stage having a lot of stage values (i.e. > 1000), we also had to set the java stack size to a bigger value. For example, we used the following parameters to generate arrays: `java -Xms1000m -Xmx5000m -Xss1000m`.

Our proposed configurable testing framework ERIS can be regarded as a proof-of-concept for combinatorial testing of the system call interface of the Linux kernel. We have it running on Debian 6 squeeze and Debian 7 wheezy. The further development of ERIS towards a fully automated and comprehensive tool is a work in progress. We are currently working on implementing the flattening methodology in ERIS, taking advantage of sophisticated features available in the Linux kernel. We are aiming towards an empirical comparison between the two presented modeling methodologies. Based on these results, we aim to provide a detailed comparison with the findings of the original TRINITY fuzz tester in future work.

V. RELATED WORK

We discuss the most related work in two areas, including API testing and empirical studies on combinatorial testing.

In particular, fuzz testing has been applied for OKL4 system calls testing in [21] where research was focused on security issues. In [22] a tool implementing an interesting approach for a completely automated virtual machine monitor based on a mutating fuzzer is developed. Fuzz testing and functional testing has also been applied to APIs. For example, an integrated environment to automate generation of function tests for APIs has been presented in [23], while violating assumptions with fuzzing has been explained in [24].

Last but not least, we give some references of empirical studies related to combinatorial testing in the field of software testing. We do not aim to provide a comprehensive, or by all means complete, treatment of the subject of combinatorial testing, as this is not the purpose of the present paper. This objective is covered by recent surveys of research for combinatorial testing that can be found in [25] and [26]. We are merely interested in giving a flavor of the many different application areas involved, in order to exhibit that while combinatorial testing is a specialized methodology for test case generation, its application to the domain of software testing is of current and growing interest.

For example, three different case studies of combinatorial testing methods in software testing have been given in [27]. Applying combinatorial testing to the Siemens Suite and testing **ACTS** with **ACTS** have been presented in [13] and [14], respectively. A case study for pairwise testing through 6-way interactions of the Traffic Collision Avoidance System (TCAS) has been presented in [28], while in [29] a study was conducted to replicate the Lockheed Martin F-16 combinatorial test study in a simplified manner. Moreover, a proof-of-concept experiment using a partial t-wise coverage framework to analyze integration and test data from three different NASA spacecrafts has been presented in [30]. Finally, combinatorial testing on ID3v2 tags of MP3 files was given in [31].

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented the combinatorial testing workflow of a new testing framework, ERIS, targeted on testing the Linux system call API. In particular, we modeled the input space of the Linux system calls in terms of combinatorial testing and in the aftermath we gave some implementation aspects on how ERIS can be integrated with existing testing infrastructures like the **ACTS** test generation tool and the TRINITY fuzz tester. Moreover, our empirical study so far provided evidence for the applicability of combinatorial testing to the domain of API testing.

As future work, we envision a lot of different opportunities for further development of our testing framework. The durations of development cycles are shrinking, time to market is short, with the help of a lot of frameworks and integration between different layers of the software stack. From a development perspective, regarding quality assurance processes like continuous integration, it is necessary to be able to decide in time whether a given release candidate fulfills certain requirements such are security controls and the availability of the kernel (in terms of being able to recover from non-terminal errors and running correctly). Another possible application for the presented modeling methodologies could target the intra Linux kernel API, and in particular the communication between different subsystems via function calls. Last but not least, we would like to mention that the system call interface is tightly coupled with security frameworks in the Linux kernel like SELinux.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for their helpful comments and suggestions that improved the quality and the presentation of the paper. This research has received funding from the Austrian Research Promotion Agency (FFG) under grant 832185 (MOdel-Based SEcurity Testing In Practice) and the Austrian COMET Program (FFG). In addition, the work of the second author was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. The research leading to these results has received funding from the *European Union* Seventh Framework Programme (FP7/2007-2013) under *grant agreement* no. 246016.

REFERENCES

[1] "The Linux Kernel Archives." [Online]. Available: https://www.kernel.org

[2] "List of bugs found by Trinity." [Online]. Available: codemonkey.org.uk/projects/trinity/bugs-found.php

[3] A. P. Mathur, *Foundations of Software Testing*, 1st ed. Addison-Wesley Professional, 2008.

[4] "LCA: The Trinity fuzz tester." [Online]. Available: https://lwn.net/Articles/536173/

[5] "Trinity fuzz tester." [Online]. Available: codemonkey.org.uk/projects/trinity

[6] NIST, *User Guide for ACTS*. [Online]. Available: http://csrc.nist.gov/groups/SNS/acts/documents/acts_user_guide_v2_r1.1.pdf

[7] "Case studies - combinatorial and pairwise testing." [Online]. Available: csrc.nist.gov/groups/SNS/acts/case-studies.html

[8] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, pp. 437–444, Jul. 1997.

[9] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *Software Engineering, IEEE Transactions on*, vol. 32, no. 1, pp. 20–34, 2006.

[10] L. Yu, Y. Lei, R. Kacker, and D. Kuhn, "Acts: A combinatorial test generation tool," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, 2013, pp. 370–375.

[11] C. J. Colbourn, "Covering arrays," in *Handbook of Combinatorial Designs*, 2nd ed., ser. Discrete Mathematics and Its Applications, C. J. Colbourn and J. H. Dinitz, Eds. Boca Raton, Fla.: CRC Press, 2006, pp. 361–365.

[12] M. Grindal and J. Offutt, "Input parameter modeling for combination strategies," in *Software Engineering*, Innsbruck, Austria, Feb. 2007.

[13] L. Ghandehari, M. Bourazjany, Y. Lei, R. Kacker, and D. Kuhn, "Applying combinatorial testing to the siemens suite," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, 2013, pp. 362–371.

[14] M. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn, "Combinatorial testing of acts: A case study," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, 2012, pp. 591–600.

[15] I. Segall, R. Tzoref-Brill, and A. Zlotnick, "Common patterns in combinatorial models," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, 2012, pp. 624–629.

[16] M. Borazjany, L. Ghandehari, Y. Lei, R. Kacker, and D. Kuhn, "An input space modeling methodology for combinatorial testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, 2013, pp. 372–381.

[17] T. Chen, P.-L. Poon, S.-F. Tang, and T. Tse, "On the identification of categories and choices for specification-based test case generation," *Information and Software Technology*, vol. 46, no. 13, pp. 887 – 898, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584904000606

[18] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical combinatorial testing," Gaithersburg, MD, United States, Tech. Rep., 2010.

[19] I. Segall, R. Tzoref-Brill, and A. Zlotnick, "Simplified modeling of combinatorial test spaces," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, 2012, pp. 573–579.

[20] D. Kuhn, R. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*, ser. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. Taylor & Francis, 2013. [Online]. Available: http://books.google.at/books?id=uWLAt4ocqTsC

[21] A. Gauthier, C. Mazin, J. Iguchi-Cartigny, and J.-L. Lanet, "Enhancing fuzzing technique for okl4 syscalls testing," in *Proceedings of the 2011 Sixth International Conference on Availability, Reliability and Security*, ser. ARES '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 728–733.

[22] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi, "Testing system virtual machines," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 171–182.

[23] A. Paradkar, "Salt - an integrated environment to automate generation of function tests for apis," in *Proceedings of the 11th International Symposium on Software Reliability Engineering*, ser. ISSRE '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 304–316.

[24] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security and Privacy*, vol. 3, no. 2, pp. 58–62, Mar. 2005.

[25] M. Brcic and D. Kalpic, "Combinatorial testing in software projects," in *MIPRO, 2012 Proceedings of the 35th International Convention*, 2012, pp. 1508–1513.

[26] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011.

[27] M. Mehta and R. Philip, "Applications of combinatorial testing methods for breakthrough results in software testing," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 348–351.

[28] D. Richard Kuhn and V. Okum, "Pseudo-exhaustive testing for software," in *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, ser. SEW '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 153–158.

[29] A. M. Cunningham Jr., J. Hagar, and R. J. Holman, "A system analysis study comparing reverse engineered combinatorial testing to expert judgment," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 630–635.

[30] J. Maximoff, M. Trela, D. Kuhn, and R. Kacker, "A method for analyzing system state-space coverage within a t-wise testing framework," in *Systems Conference, 2010 4th Annual IEEE*, 2010, pp. 598–603.

[31] Z. Zhang, X. Liu, and J. Zhang, "Combinatorial testing on id3v2 tags of mp3 files," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 587–590.