Fuzzing with Aliasing Structures

Ruben Backx
R.W.Backx@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Andy Zaidman
A.E.Zaidman@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Andreea Costea
M.A.Costea@tudelft.nl
Delft University of Technology
Delft, The Netherlands

***Abstract*—Rust is a programming language which promises memory safety. That is, as long as programmers work in the "safe" part of the language. A common pattern in Rust code is to *encapsulate* unsafe behaviour in a safe function. These safe encapsulations are a large source of memory bugs in Rust code. Especially bugs caused by unexpected *aliasing structure* can be difficult to detect. We developed a new fuzzing-based approach to target these kinds of bugs in unsafe Rust code. Our approach can generate all test inputs for any function, guided by the Tree Borrows aliasing model. Each generated input is constructable in safe Rust, and no input is ever generated that always exhibits the same behaviour as an earlier input. For future work, we propose an extensive evaluation of our method that will help us answer how prevalent bugs related to aliasing structure are.**

***Index Terms*—dynamic verification, fuzzing, property-based testing, Rust.**

I. Introduction

Rust is a programming language, which supports low-level operations while promising memory safety. This is achieved by having the compiler check a sophisticated *ownership model* [1]. Among other guarantees, these checks ensure that *references* (i.e., pointers with an associated *lifetime*) never outlive the values they point to, and that values are referenced either mutably (notated `&mut T`) or immutably (notated `&T`), but never both actively[1] at the same time. Unfortunately, many use cases, such as interaction with low-level systems or working with the internal representation of data structures, require violating the ownership model, at least temporarily. Rust allows this in *unsafe* code. When the programmer declares their code to be in an unsafe environment (e.g., by putting it in an `unsafe` block), the compiler trusts the programmer to write memory-safe code. In such unsafe code, developers typically work with *raw pointers* (notated `*const T` and `*mut T`). Crucially, the programmer is still required to manually ensure memory safety. For example, when writing to a raw pointer, the programmer needs to ensure the pointer is not null, is aligned, and points to a currently allocated region of memory. These properties of Rust give rise to the *safe encapsulation* pattern, where unsafe code is surrounded by runtime checks ensuring its safety. The combined logic is then made available by way of a regular, "safe" function. Sometimes, no additional checks are necessary because the code *is* safe, but the compiler is unable to prove it. As an example, the snippet below returns mutable references to the first and last element of a vector (a resizable array). An `unsafe` block is necessary, because two mutable references are created to the same `Vec` allocation, which is forbidden in safe Rust. Nevertheless, this function is safe, because we manually ensure the first and last element of the vector are distinct by checking whether it contains at least two elements.

```
fn first_and_last_mut<T>(vec: &mut Vec<T>)
    -> Option<(&mut T, &mut T)> {
    if vec.len() < 2 {
        return None;
    }
    unsafe {
        Some((
            &mut *(vec.first_mut()?
                as *mut T),
            &mut *(vec.last_mut()?
                as *mut T))
        )
    }
}
```

As might be expected, the majority of memory safety bugs in Rust programs stem from unsafe code. In fact, *all* known memory safety bugs in Rust programs stem from unsafe code [2]. It stands to reason, then, that verification efforts should be concentrated on safe encapsulations of unsafe code to gain confidence they are actually safe. One promising validation approach is *fuzzing*, which, since it is a dynamic approach, can profit from runtime information, but also requires little to no user input [3]. Fuzzing can also "target" specific functions, such that efforts can be focused on safe encapsulation functions. In the input generation however, we risk missing many interesting bugs if we do not consider the *aliasing structure* of the generated input, i.e., which references point to which values. To see why, consider the example below.

```
fn f(x: *mut i32, y: *mut i32) -> i32 {
    unsafe {
        *x = 1;
        *y = 2;
        *x
    }
}
```

The programmer might expect `f` to always return `1`. `x` is assigned to `1` after all, and its value is returned at the end, but in actuality, `x` and `y` may *alias* since they are raw pointers, and aliasing raw pointers can be created in safe Rust. If `x` and

[1]The word "actively" is doing a lot of heavy lifting here. In reality, Rust's aliasing rules allow mutable and immutable references to overlap as long as long as they are not being used at the same time.

`y` alias, the value `x` refers to is overwritten to `2` by the write to `y`. The assumption that `x` points to `1` after this function is therefore wrong and can lead to bugs. Such a bug would not be caught by only fuzzing the values `x` and `y` refer to, and fuzzing of memory addresses would only catch undesirable behaviour if the two addresses happen to be the same (which can have a chance as low as $\frac{1}{2^{64}}$, depending on how the addresses are generated). Moreover, it is not enough to naively assign pairs of references to alias at random. While that would generate more interesting aliasing structures, purely random generation can generate aliasing structures that could never be created by a compiling Rust program, and thus are not problematic, even if they cause memory unsafety.

This paper presents a new technique for generating test inputs for Rust, the first explicitly taking aliasing structure into account. Our approach generates inputs that are

- *plausible*: inputs that can be created by a compiling safe Rust program,
- *unique-behaviour probable*: each input has the potential to cause behaviour not encountered before, and
- *countable*: it is possible to generate all inputs in order.

What behaviour an input is capable of causing is determined by an *aliasing model*. This is a formal description of the runtime behaviour of a programs memory operations. Our approach is guided by the Tree Borrows aliasing model [4]. Furthermore, the generation can be configured such that more priority is given to exploring different aliasing structures or to exploring different values. Finally, we describe a thorough evaluation, planned for the future, which will answer our research question: *How effective is aliasing structure fuzzing for detecting bugs?*

We explain more about aliasing models for Rust in Section II. The full generation method is described in Section III. Section IV details the evaluation we have planned. Related work is outlined in Section V. Finally, Section VI concludes with future work.

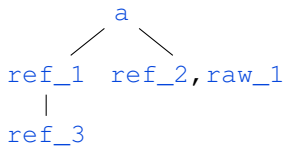
II. Aliasing Models

In unsafe Rust code, the compiler allows more operations than it does in safe Rust. The programmer is allowed to directly read from or write to raw pointers, for example. Even though unsafe operations are always allowed by the compiler in unsafe code, they need to abide by manually checked constraints to not cause undefined behaviour. An *aliasing model* formally describes the allowed runtime behaviour of memory operations. Some proposed aliasing models are Stacked Borrows [5] and its successor Tree Borrows [4].

For our input generation approach, we consider Tree Borrows as the aliasing model. In Tree Borrows, every memory allocation is associated with a tree. The allocation forms the root of the tree. When a value is *borrowed* by creating a reference to it using `&` or `&mut`, a child node is added. Any references created from those (*reborrows*) create children of the nodes they reborrow from. Raw pointers do not create any new nodes, but instead are added directly to the node from which they were created. As an example, consider the following code snippet.

```
let a = 100;
let ref_1 = &a;                    // borrow
let ref_2 = &a;                    // borrow
let ref_3 = &*ref_1;               // reborrow
let raw_1 = ref_2 as *const i32; // raw ptr
```

This corresponds to the following tree:



Additionally, every node has an associated *permission*. Permissions throughout the entire tree are updated on every read and write to any node in the tree. This allows the model to detect undefined behaviour.

Since aliasing models determine the behaviour of memory operations, we can use them to determine which values should be considered equivalent for test input generation. From a set of input values which all exibit the same behaviour under Tree Borrows, i.e. correspond to equivalent borrow trees, only one needs to be generated. Testing more than one input from this set would simply be redundant, that is, wasted resources.

III. Test Input Generation with References

Previous work has introduced a method for uniformly generating test input for languages with algebraic data types [6]. The idea is to define an enumeration of all possible values for any algebraic data type. An algebraic data type consists of a set of *constructors*, each of which can construct a value of their associated type given a list of arguments.

A. Adapting to Rust

Algebraic data types in Rust are declared using the `enum` keyword. The `Option` type from the standard library, for example, is defined as follows.

```
enum Option<T> {
    None, Some(T)
}
```

`Option` has two constructors: `None`, which takes no arguments, and `Some`, which takes one argument of the type `T`, the type parameter passed to `Option`. To enumerate all possible values of a type, the set of all values is split into a countable number of subsets. Each subset corresponds to a positive integer size k, which represents the exact number of constructors used to construct the values in the subset. Following the `Option` example above, its first three subsets are:

$$\texttt{Option}_0 = \{\}$$
$$\texttt{Option}_1 = \{\texttt{None}\} \cup \{\texttt{Some}\,(x) \mid x \in \texttt{T}_0\}$$
$$\texttt{Option}_2 = \{\texttt{Some}\,(x) \mid x \in \texttt{T}_1\}.$$

`T`$_0$ and `T`$_1$ here refer to the set of values of type `T` constructed with 0 and 1 constructors respectively.

Using this construction, an input value can be selected by picking a random integer i and applying the following function starting at $k = 0$.

$$index_S(i,k) = \begin{cases} index_{S,k}(i) & \text{if } i < |S_k| \\ index_S(i - |S_k|, k+1) & \text{otherwise} \end{cases}$$

Here, $index_{S,k}$ is a function that indexes S_k, the set of k-sized values of a data type S, as defined by Claessen et al. [6].

An **`enum`** in Rust always has all its contructors accessible, provided the enum itself is accessible. Rust also has **`struct`**s, which can be thought of as algebraic data types with only one constructor. This constructor is not publicly accessible, unless all of the **`struct`**'s fields are declared public. While it is possible to construct these types using unsafe Rust, this is undesirable, as the data type might require an invariant to hold for it to function properly, which is always upheld by its public API, but would be broken by unsafe construction. Instead, our solution is to look for all public functions that return a value of the data type and consider those the data type's "constructors". If this automated approach fails to find any constructors, they can be provided manually. As an example, `Box` from the standard library, is a struct with non-public fields, but there is a public function `new`, summarised below, which we consider `Box`'s only constructor.

```
pub fn new<T>(value: T) -> Box<T>
```

B. Generation with References

The presence of references poses some challenges. On the one hand, it is desirable to generate many different aliasing structures, so as much as possible different behaviour can be observed. On the other hand, we do not want to generate *all* aliasing structures, lest we waste resources on the same behaviour being tested more than once. Our approach lies perfectly on the midpoint, generating all potentially unique behaviours exactly once. Furthermore, generated values are guaranteed to be constructable in safe Rust. The number of unique behaviours can be large. In fact, it can be infinite. We do not claim to be able to exhaustively test in these cases, but we do never generate the same value or aliasing structure twice. The approach is split into three steps, which we describe in the rest of this section.

Step 1, generate an in-memory value: Using the technique described in Section III-A, we generate a temporary *pseudo-value* in memory. This is done by running the generation logic with special treatment for reference and pointer types. Mutable references in safe Rust must be exclusive, i.e., only one mutable reference to an allocation can exist at any time. We therefore treat mutable references as algebraic data types with one constructor: `NewRef`, which represents creating an allocation and immediately mutably borrowing it for the duration of the test. This corresponds to the following pseudo-code.

```
enum<T> &mut T {
    NewRef(T)
}
```

Shared references can also be immediately borrowed from a new allocation, but additionally can be reborrowed from existing references. As such, we consider shared references to have two constructors: `NewRef` and `Reborrow` [2].

```
enum<T> &T {
    NewRef(T), Reborrow
}
```

The `Reborrow` constructor takes no arguments, as the exact aliasing structure is filled in by later steps in the generation process when the structure of the generated value is fully known.

Pointers can be created by new allocation and reborrowing (using `&raw const` or `&raw mut`) as well, but can also be created by casting an existing reference to a pointer. Pointers therefore have an additional `Cast` constructor. Here again, which reference the pointer is casted from is determined in later steps, so no arguments are required for the `Cast` constructor. `Cast` is different from `Reborrow` in that it does not add a new node to the borrow tree in the Tree Borrows model, but is associated with an existing node instead.

```
enum<T> *const T {
    NewRef(T), Reborrow, Cast
}
enum<T> *mut T {
    NewRef(T), Reborrow, Cast
}
```

Step 2, assign allocations: After generating a pseudo-value which contains any number of `NewRef`, `Reborrow`, and `Cast` constructors, we assign references to allocations. `NewRef` constructors represent the creation of new allocations, so they do not need to be assigned. The assignment process assigns all `Reborrow` and `Cast` constructors to an allocation.

Not all allocations are valid assignment candidates for every constructor. Consider, for example, the following piece of pseudo-code[3].

```
enum BinaryTree {
    Leaf,
    Node(&BinaryTree, &BinaryTree)
}
```

If `Node(NewRef(...), Reborrow)` is generated, the `Reborrow` can be assigned to the allocation of the left child without problems. However, if a nested structure like `Node(NewRef(Node(..., Reborrow)), ...)` is generated, the inner `Reborrow` cannot be assigned to the outer `NewRef`. This is because, in order to generate any value with this structure, a reborrow needs to be done, which

[2]Readers familiar with Tree Borrows might wonder why there is no `Borrow` constructor. This is because we always generate a unique borrow after an allocation and reborrow from there, which is equivalent in behaviour to allowing both borrows and reborrows from any allocation.

[3]Lifetimes have been elided for clarity.

requires the to-be-borrowed allocation to be fully constructed, which, in turn, requires the reborrow to already be done, i.e., there is a cycle in the dependencies. To avoid these cycles, we remove these kinds of dependent allocations from the candidate allocation pool for each constructor.

Due to this filtering of invalid allocations or simply because no `NewRef` constructors were generated, it is possible for a constructor to have no candidate allocations. In this case, we generate an allocation pointing to the smallest possible value of its type, which then acts as the only candidate allocation.

Once the candidate allocation pools for each constructor have been identified, any number of random assignments are generated. The user can configure this number to tailor the level of exploration into different aliasing assignments to their needs.

Step 3, generate borrow trees: In this step, we use the allocation assignment to generate behaviour-unique aliasing structures according to Tree Borrows. With such a *borrow tree*, we can generate a valid Rust value corresponding to the pseudo-value generated in the first step. This is different from the previous step, in which it was determined which allocation each reference refers to. As an example, we consider the three snippets listed below.

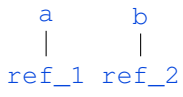
1)
```
let a = Box::new(1);
let b = Box::new(2);
let ref_1 = &*a;
let ref_2 = &*b;
```

2)
```
let a = Box::new(1);
let ref_1 = &*a;
let ref_2 = &*a;
```

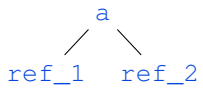
3)
```
let a = Box::new(1);
let ref_1 = &*a;
let ref_2 = &*ref1;
```

Snippet 1 differs from snippet 2 and snippet 3 in allocation assignment. The references `ref_1` and `ref_2` in snippet 1 refer to different allocations altogether. Snippet 2 and 3 only differ in aliasing structure. In snippet 2, both references are borrowed from `a`, whereas in snippet 3, `ref_1` is borrowed directly from `a` and `ref_2` is a reborrowing of `ref_1`.

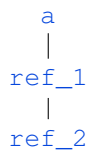
The aliasing model, Tree Borrows, determines which aliasing structures are considered equivalent. In Tree Borrows, the aliasing structure is a borrow tree; the initial value is the root of the tree and borrowings and reborrowings correspond to child nodes. Snippet 1, 2, and 3, for example, corresponds to the following trees.



(a) Snippet 1



(b) Snippet 2



(c) Snippet 3

A borrow tree for each allocation is generated similarly to initial pseudo-value generation. The constructors used are `Tree0(Ref)`, `Tree1(Ref, Tree)`, `Tree2(Ref, Tree, Tree)`, etc. These correspond to trees with zero, one, two, or more children respectively. Since we know the number of nodes we need per allocation, we can generate a tree through $index_{Tree,k}(i)$ where i is a random integer between 0 and $|Tree_k|-1$ and k is the number of nodes. The `Ref` values are assigned from the available `Reborrow` constructors. We use a modified generator for the trees in order to ensure constructors are only assigned once and no equivalent trees are generated. Two trees are considered equivalent if they always exibit the same behaviour under Tree Borrows. Concretely, that means the order of child nodes is irrelevant. This is solved by assigning a unique hash to every possible tree and generating only lists of children ordered by this hash. `Cast` constructors are assigned randomly to any node in the borrow tree after it has been generated.

In the Tree Borrows model, each node in the tree is assigned a *permission*. This permission determines in what circumstances reads and writes are allowed. Once a read or write is done to any reference in the tree, the full tree has its permissions updated to reflect the effects of that action. We currently leave the permissions of the generated trees at their initial value, as we hypothesise the structure of the tree, not the permissions, to have the largest impact on program behaviour. Nevertheless, it is possible to fuzz the permissions as well by applying legal reads and writes to the references after generation.

Once the borrow tree is generated, a concrete test input value can be derived. For each `NewRef(T)` reference, a local variable of type `Box<T>` is created containing the value inside. The constructor is then replaced by `&*box`, where `box` is the name of the variable. The rest of the references are created in topological order of the generated borrow trees. `Reborrow` constructors are converted to local variables reborrowing the earlier-defined variable corresponding to its parent node. For `Cast` constructors, the local variable corresponding to their assigned node is cast using `as *const T` or `as *mut T`. This process removes all the reference constructors from the generated pseudo-value, leaving a valid Rust value. For example, cosider the following pseudo-value.

```
BinaryTree::Node(
    NewRef(BinaryTree::Leaf),
    Reborrow
)
```

This results in a value equivalent to `result` in the following rust code.

```
let ref_1_val = Box::new(BinaryTree::Leaf);
let ref_1 = &*ref_1_val;
let ref_2 = &*ref_1;
let result = BinaryTree::Node(ref_1, ref_2);
```

The three-step nature of the process naturally leads to a nice property: the amount of exploration in each stage of the process is configurable. The user can decide whether they want to spend more time testing different values, allocation assignemnts, or aliasing structures.

IV. Planned Evaluation

In order to gauge the effectiveness of our approach, we have an evaluation planned. The top 250 Cargo crates will be selected. From these crates, all public, safe functions will be extracted, for which we will let our generator generate 20 first-step pseudo-values. We then run the function for each input in debug mode with a timeout of 10 seconds. Any crash will be recorded as a test failure, functions that return normally are considered successful, and timeouts will not be futher analysed.

This entire process will be repeated multiple times with different settings. The percentage of different allocation assignments explored will be set from 0% to 100%, in 20% increments. Similarly, the percentage of aliasing structures explored will be set to the same range of values independently. For any particular pseudo-value, we will set a maximum runtime of 30 seconds and record if exploration stopped early.

The full experiment will be executed twice: once by executing the generated test cases regularly and once with Miri [7]. Miri is an interpreter for Rust, which can be configured to detect Tree Borrows violations. If such a violation is detected, it is a strong indication of a bug, even when the program does not crash regularly. We hypothesise Miri will be able to detect more bugs because of this reason. For example, Miri can identify the bug in the example function `f` from Section I, whereas detecting such a bug regularly can only be done if it somehow causes the program to crash. As not every Rust feature is implemented in Miri, test cases that fail because of unsupported operations will be recorded separately.

All failures will be analysed manually to determine whether there is acutally a bug, and whether that bug is caused by the aliasing structure or has some other cause. The resulting data will help us answer, (1) if real-world bugs can be detected using aliasing structure fuzzing, (2) what level of exploration is effective and feasible, and (3) whether using Miri to run these test suites is advantageous.

V. Related Work

Recent work on targeted fuzzing for Rust has shown promising results already. Paaßen et al. use coverage feedback to focus testing efforts on unsafe Rust code [8], and Xu et al. generate unsafe API sequences with a focus on APIs with generics [9].

Test input generation also has value in the domain of property-based testing [10], [11]. Much work on type-guided input generation comes from this domain [12]–[15]. Some proof assistants, such as Isabelle, use test input generation to automatically falsify statements [16], [17].

McCormack et al. used Miri together with an LLVM interpreter to test detect undefined behaviour in code bases with foreign function calls [18]. Executing our fuzzer with such a runtime is a promising direction to expand to, once our initial evaluation is complete.

Godefroid et al. apply a similar approach to us, generating test drivers that perform random testing [19].

Finally, Carott et al. developed a static approach to validate safe encapsulations of unsafe code [20]. Similar to us, they consider only public facing constructor functions to avoid considering implausible values.

VI. Conclusion

We have developed a new automatic test input generation approach for Rust, which is able to generate all inputs with potentially unique behaviour, both in terms of constructor structure, and aliasing structure according to the Tree Borrows aliasing model. The levels of exploration of differently constructed values, assigned references, and aliasing structures are configurable.

While our approach is ready for evaluation on the majority of Rust code, we have identified some improvements and extensions. Currently, which references can point to which allocations is determined by filtering out infeasible candidate allocations through the place they occur in the constructor tree. This can fail if the reference types in the data structure under construction are annotated with lifetimes. In Rust, lifetimes determine how long a reference is allowed to live, which can put compile-time constraints on what allocations a reference is allowed to point to. These constraints can be more restrictive than our simple filtering, leading to the generation of non-compiling values. We solve this at the moment by discarding any non-compiling value, but in the future, a better solution is to use lifetime annotations to guide generation.

An interesting extension of our work is to apply it to property-based testing. Our planned evaluation can be considered a fuzzing-based approach, as we do not have any information on how functions under test ought to operate. Some projects, however, already specify expected function behaviour, because they use a kind of property-based verification. Extracting these properties and verifying them using property-based testing powered by our generator could help uncover bugs where functions do not behave as specified due to the aliasing structure of allowed inputs. Additionally, a method for "shrinking" values with aliasing information could be developed to help make debugging of such a tool easier.

Finally, we outlined a planned evaluation to be included in a future full-length paper, where we will answer whether this our approach is suitable for detecting real-world bugs, what level of aliasing structure exploration is effective, and how our approach interacts with Miri.

References

[1] N. D. Matsakis and F. S. Klock, "The Rust Language," in *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. Portland Oregon USA: ACM, Oct. 2014, pp. 103–104. [Online]. Available: https://dl.acm.org/doi/10.1145/2663171.2663188

[2] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–25, Jan. 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3466642

[3] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 2016, pp. 1032–1043. [Online]. Available: https://dl.acm.org/doi/10.1145/2976749.2978428

[4] N. Villani, J. Hostert, D. Dreyer, and R. Jung, "Tree Borrows," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 1019–1042, Jun. 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3735592

[5] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked Borrows: an aliasing model for Rust," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, Jan. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3371109

[6] K. Claessen, J. Duregård, and M. H. Pałka, "Generating constrained random data with uniform distribution," *Journal of Functional Programming*, vol. 25, p. e8, 2015. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0956796815000143/type/journal_article

[7] R. Jung, B. Kimock, C. Poveda, E. S. Muñoz, O. Scherer, and Q. Wang, "Miri: Practical Undefined Behavior Detection for Rust," *Proceedings of the ACM on Programming Languages*, vol. 10, no. POPL, pp. 1383–1411, Jan. 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3776690

[8] D. Paaßen, J.-R. Giesen, and L. Davi, "Targeted Fuzzing for Unsafe Rust Code: Leveraging Selective Instrumentation," in *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. Istanbul Turkiye: ACM, Jun. 2025, pp. 488–499. [Online]. Available: https://dl.acm.org/doi/10.1145/3756681.3756956

[9] Z. Xu, B. Wu, C. Wen, B. Zhang, S. Qin, and M. He, "RPG: Rust Library Fuzzing with Pool-based Fuzz Target Generation and Generic Support," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Lisbon Portugal: ACM, Apr. 2024, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3639102

[10] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. ACM, Sep. 2000, pp. 268–279. [Online]. Available: https://dl.acm.org/doi/10.1145/351240.351266

[11] J. Hughes, "QuickCheck Testing for Fun and Profit," in *Practical Aspects of Declarative Languages*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and M. Hanus, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 4354, pp. 1–32, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-540-69611-7_1

[12] C. Runciman, M. Naylor, and F. Lindblad, "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values," *ACM SIGPLAN Notices*, vol. 44, no. 2, pp. 37–48, Jan. 2009. [Online]. Available: https://dl.acm.org/doi/10.1145/1543134.1411292

[13] J. Duregård, P. Jansson, and M. Wang, "Feat: functional enumeration of algebraic types," in *Proceedings of the 2012 Haskell Symposium*. Copenhagen Denmark: ACM, Sep. 2012, pp. 61–72. [Online]. Available: https://dl.acm.org/doi/10.1145/2364506.2364515

[14] B. Fetscher, K. Claessen, M. Pałka, J. Hughes, and R. B. Findler, "Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System," in *Programming Languages and Systems*, J. Vitek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, vol. 9032, pp. 383–405, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-662-46669-8_16

[15] L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce, "Generating good generators for inductive relations," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–30, Jan. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3158133

[16] L. Bulwahn, "Smart Testing of Functional Programs in Isabelle," in *Logic for Programming, Artificial Intelligence, and Reasoning*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, N. Bjørner, and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7180, pp. 153–167, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-28717-6_14

[17] ——, "The New Quickcheck for Isabelle: Random, Exhaustive and Symbolic Testing under One Roof," in *Certified Programs and Proofs*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, C. Hawblitzel, and D. Miller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7679, pp. 92–108, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-35308-6_10

[18] I. McCormack, J. Sunshine, and J. Aldrich, "A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, Apr. 2025, pp. 2075–2086, arXiv:2404.11671 [cs]. [Online]. Available: http://arxiv.org/abs/2404.11671

[19] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. Chicago IL USA: ACM, Jun. 2005, pp. 213–223. [Online]. Available: https://dl.acm.org/doi/10.1145/1065010.1065036

[20] P. Carrott, S.-E. Ayoun, and A. Raad, "Compositional Bug Detection for Internally Unsafe Libraries: A Logical Approach to Type Unsoundness," *LIPIcs, Volume 333, ECOOP 2025*, vol. 333, pp. 5:1–5:28, 2025, artwork Size: 28 pages, 1135949 bytes ISBN: 9783959773737 Medium: application/pdf. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2025.5