CRUST: A Bounded Verifier for Rust

John Toman, Stuart Pernsteiner, and Emina Torlak
Department of Computer Science and Engineering
University of Washington, Seattle, Washington
{jtoman,spernste,emina}@cs.washington.edu

***Abstract*—Rust is a modern systems language that provides guaranteed memory safety through static analysis. However, Rust includes an escape hatch in the form of "unsafe code," which the compiler assumes to be memory safe and to preserve crucial pointer aliasing invariants. Unsafe code appears in many data structure implementations and other essential libraries, and bugs in this code can lead to memory safety violations in parts of the program that the compiler otherwise proved safe.**

We present CRUST, a tool combining exhaustive test generation and bounded model checking to detect memory safety errors, as well as violations of Rust's pointer aliasing invariants within unsafe library code. CRUST requires no programmer annotations, only an indication of the modules to check. We evaluate CRUST on data structures from the Rust standard library. It detects memory safety bugs that arose during the library's development and remained undetected for several months.

***Keywords*—SMT-based verification, test generation, memory safety**

I. Introduction

While much of today's application code is written in safe languages, system software—OS kernels, device drivers, and web browsers—is still written in unsafe languages such as C and C++. For these programs, runtime performance is critical, and using safe languages with managed runtimes is not an option. The price of using unsafe languages, however, is code with dangling pointers, buffer overflows, and other memory errors that cause not only crashes but security vulnerabilities [1].

Rust [2] is a new systems programming language that provides memory safety *without* relying on a managed runtime. Instead, it employs an advanced type system that restricts the use of object references. The Rust compiler ensures that (1) every reference points to valid, initialized memory, and (2) every accessible mutable reference has no accessible aliases. The compiler rejects all programs that cannot be conservatively proven to satisfy these safety properties.

Rust's restrictions on references are sufficient to guarantee memory safety [3], while also being sufficiently liberal to admit most real-world application code. They are, however, too strict to admit some critical low-level libraries, particularly code that implements (mutable, variable-sized) data structures such as vectors, hash maps, and linked lists. In Rust, these data structures are implemented using an escape hatch, special `unsafe` blocks that the compiler does not analyze for memory safety. These `unsafe` blocks are the only potential source of memory errors in otherwise safe Rust programs.

This paper introduces CRUST, a bounded verifier for Rust that aims to bridge the memory safety gap created by unsafe library code. CRUST works fully automatically, requiring no manual annotations, assertions, or test drivers. It takes as input a module to be analyzed and a simple filter indicating the functions of interest within that module, and verifies the memory safety of every function under test that contains unsafe code, on all inputs that can be constructed using a bounded number of calls. The result is either a bounded safety guarantee or a concrete program and input that leads to a memory error or a violation of an aliasing invariant.

CRUST works by a novel combination of test sequence generation and SMT-based bounded verification. Test sequence generation is used to construct *drivers*—sequences of function calls from the target module. CRUST automatically inserts assertions into each driver to check for Rust memory safety violations, then translates the drivers to finitized C programs. The resulting C programs are verified with the CBMC [4] bounded model checker, which works by reduction to SMT. CRUST's analysis is embarrassingly parallel and exhaustively checks complex libraries in a few hours of CPU time.

We evaluated CRUST on three modules from the Rust standard library. The Rust library benchmarks include older versions of data structure implementations with known (now patched) memory safety violations, which CRUST found in 8 hours of CPU time. These bugs originally took 1–3 months of use to emerge in practice. CRUST was able to show that these modules are free of memory safety errors for inputs produced by call sequences of length up to 3.

In summary, this paper makes the following contributions:

- A technique for combining test sequence generation and SMT-based bounded verification to check memory safety of systems code. Our technique employs a new algorithm for generating bounded sequences of API calls and a new translation from Rust to C that produces code amenable to automated analysis with CBMC.
- Implementation of our technique in CRUST, a bounded verifier for Rust that scales to real Rust libraries and runs on unmodified Rust code.
- Evaluation of CRUST on three modules from the Rust standard library, showing that it can easily find memory safety violations that took months to discover in practice.

The rest of the paper is organized as follows. Section II illustrates key features of Rust and CRUST on a small example. Section III presents our test sequence generation technique and our translator from Rust to C. Section IV briefly describes our implementation of CRUST. Section V evaluates CRUST on

```
struct Array<T> {
  ptr: *mut T,
  len: uint,
}
impl<T> Array<T> {
  fn new(len: uint) -> Array<T> {
    let ptr = unsafe {
      alloc::allocate(len * mem::size_of::<T>())
    };
    Array {
      ptr: ptr as *mut T,
      len: len
    }
  }
  fn get_mut(&self, index: uint) -> &mut T {
    if index >= self.len { panic!(); }
    unsafe {
      let ptr = self.ptr.offset(index);
      &mut *ptr
    }
  }
  fn fill(&mut self, value: T) {
    for i in 0 .. self.len {
      *self.get_mut(i) = value;
    }
  }
}
```

Fig. 1: A simple fixed-sized array implementation.

```
  let a: Array<u32> = Array::new(10);
  let r1: &mut u32 = a.get_mut(0);
  let r2: &mut u32 = a.get_mut(0);
  assert!(r1 as uint == r2 as uint);
```

Fig. 2: Client code exploiting a bug in `Array::get_mut` (Figure 1) to violate the Rust aliasing invariants.

modules from the Rust standard library. We discuss related work in Section VI, and Section VII concludes the paper.

II. Overview

This section demonstrates our approach, using the `Array` data structure in Figure 1 as a running example. The type `Array` is a heap-allocated array supporting three public operations: `new` creates a new array of a given length; `get_mut` obtains a mutable reference to an element of the array; and `fill` sets every element to a particular value. The `Array` destructor and other details are omitted in this overview.

The goal of CRUST is to find memory-safety bugs in a library if any exist. The `Array` example does contain such a bug: the `get_mut` method wrongly declares its receiver argument as `&self` rather than `&mut self`. This allows client code, such as that in Figure 2, to obtain multiple mutable references to the same element, in violation of the aliasing invariants Rust uses to ensure memory safety. If `get_mut` were properly declared, the Rust compiler would reject the code in Figure 2. CRUST finds the `get_mut` bug quickly by exhaustively exploring the space of valid inputs for each method in the `Array` API.

Figure 3 shows a high-level overview of our approach. The user provides a library to test, a *library filter* to identify

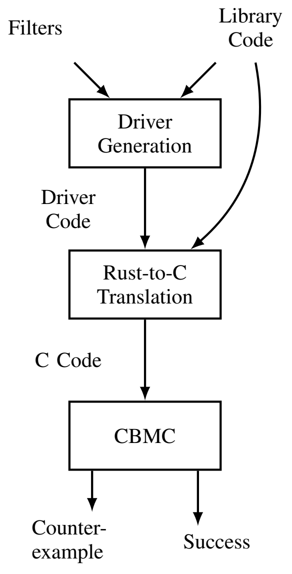


Fig. 3: An overview of the CRUST process

```
example_array::*
```

(a) Library filter, matching all functions in the `example_array` module.

```
example_array::Array::new
example_array::Array::fill
```

(b) Construction filter, matching only the `new` and `fill` methods.

Fig. 4: Additional inputs to CRUST used to find the `get_mut` bug in the code from Figure 1.

the portion of the library to test, and a *construction filter* to identify functions that may be useful for constructing inputs. For each function matching the library filter, CRUST generates a collection of drivers to test that function, possibly using functions matching the construction filter to construct input values of nonprimitive types. CRUST then translates the library and all generated drivers from Rust to C and invokes CBMC to check for bugs. The remainder of this section describes the behavior of each step on the `Array` example in Figure 1.

A. Driver Generation

CRUST begins by collecting relevant API functions to use as targets for driver generation. For this, it consults the user-provided library filter, such as the one in Figure 4a that matches all functions in the `example_array` module. In this example, and for most idiomatic Rust code, the filter is trivial because each data structure is declared in a separate module. Any function containing unsafe code whose qualified name matches the filter is a candidate for testing. Functions without unsafe code are ignored because their memory safety follows from the memory safety of the functions they call and the assumed correctness of Rust's memory safety analyses.

```
fn crust_test_1() {
  let result = Array::new(⋆);
}

fn crust_test_2() {
  let v1 = Array::new(⋆);
  Array::fill(&mut v1, ⋆);
  let result = Array::get_mut(&v1, ⋆);
  assert!(result as uint != 0);
}

fn crust_test_3() {
  let v1 = Array::new(⋆);
  let result1 = Array::get_mut(&v1, ⋆);
  let result2 = Array::get_mut(&v1, ⋆);
  assert!(result1 as uint != 0);
  assert!(result2 as uint != 0);
  assert!(result1 as uint != result2 as uint);
}
```

Fig. 5: Representative driver functions for the `Array` type defined in Figure 1

For the `Array` example and the library filter from Figure 4a, CRUST collects two target functions for testing: `Array::new` and `Array::get_mut`. The `Array::fill` function matches but is ignored because it has no unsafe code.

After collecting relevant API functions, CRUST generates a set of drivers to cover the space of valid inputs to each target function. For inputs of primitive type, CRUST uses the underlying model checker's support for nondeterminstic choice. For all other inputs, CRUST relies on the user-provided construction filter to identify functions that may be useful for constructing input values. Concretely, when a function call requires an argument of nonprimitive type, CRUST generates sequences of calls to functions matching the construction filter in order to produce values of that type.

An example construction filter appears in Figure 4b. This filter provides CRUST with two ways to produce `Array` values: it can produce an `Array` from scratch using `Array::new`, or it can transform an existing `Array` using the mutating function `Array::fill`.

The construction filter often identifies a subset of the functions in the library set, but distinguishing the two is useful for performance tuning. If the user wishes to reduce the time spent on testing, they may independently adjust the number of functions to test (the library set) and the strength of the guarantee provided for each function (the construction set).

Figure 5 shows three representative driver functions for the `Array` example. Each consists of zero or more calls to construction functions, one or two calls to target functions, and zero or more assertions to check for violations of Rust's reference validity and aliasing invariants. The first driver checks target function `Array::new` to ensure that no input value results in a memory error. The second uses construction functions `Array::new` and `Array::fill` to create an `Array`, then checks that calling `get_mut` on that array produces a non-null reference. The third makes two calls to `Array::get_mut`, and because the resulting references are mutable, it also checks that the references do not alias. This aliasing check will expose the bug in `get_mut`.

```
struct Array Array_new_u32(uintptr_t len) {
  uint8_t* ptr =
        alloc_allocate(len * sizeof(uint32_t));
  struct Array __result = {
    .ptr = (uint32_t*)ptr,
    .len = len,
  };
  return __result;
}

uint32_t* Array_get_mut(
        struct Array* self,
        uintptr_t index) {
  if (index >= self->len) panic();
  return self->ptr + index;
}

void crust_test_3() {
  struct Array v1 = Array_new(nondet_uint());
  uint32_t* result1 =
        Array_get_mut(&v1, nondet_uint());
  uint32_t* result2 =
        Array_get_mut(&v1, nondet_uint());
  assert((uintptr_t)result1 != 0);
  assert((uintptr_t)result2 != 0);
  assert((uintptr_t)result1 != (uintptr_t)result2);
}
```

Fig. 6: C translation of `Array::new` and `Array::get_mut` from Figure 1 and `crust_test_3` from Figure 5. For clarity, this figure omits syntactic artifacts that arise from the Rust-to-C translation.

Driver generation sometimes produces drivers that are not valid Rust programs. For example, even with a correctly-defined `get_mut`, CRUST still generates `crust_test_3`, but the Rust compiler rejects that code because it creates overlapping mutable references into `v1`. This occurs because the driver generation step uses only basic type correctness, not the full set of analyses used in the Rust compiler, to guide the generation of drivers. However, this simplification does not result in false positives because the erroneous drivers are detected and discarded when CRUST invokes the Rust compiler as part of the conversion from Rust to C.

B. Code Generation

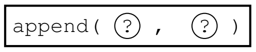
After generating all necessary drivers for each target function, CRUST translates both the library and the driver code from Rust to C. Figure 6 shows part of the translation for the `Array` example. In this case, the translation is straightforward because most Rust features used in `Array` correspond directly to features supported by C. To perform this translation, CRUST first invokes the Rust compiler frontend to produce an AST and collect semantic information (such as the type of each expression), and then uses that information to produce corresponding C code. The Rust compiler also serves to detect erroneous driver functions so they can be discarded.

C. Checking

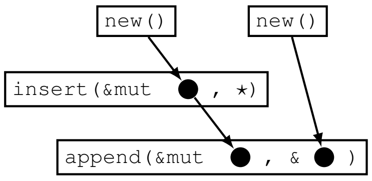
After translating the library and the generated drivers to C, CRUST invokes CBMC, a bounded model checker for C programs. CBMC checks both built-in memory safety properties and the assertions inserted by CRUST to detect violations of Rust's reference invariants. For each driver, CBMC either

```
fn new() -> List { ... }
fn insert(list: &mut List, val: u32) { ... }
fn append(dest: &mut List, src: &List) { ... }
```

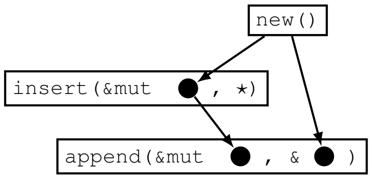
(a) API for a simple `List` data structure. Each function is part of both the library set and the construction set.



(b) Driver generation begins with a sketch of a call to the target function.



(c) CRUST recursively fills in holes until the driver is complete and well typed.



(d) CRUST merges identical branches to explore aliasing behavior.

```
fn crust_test_0() {
  let v = List::new();
  List::insert(&mut v, __crust::nondet());
  let result = List::append(&mut v, &v);
  assert!(...);
}
```

(e) Rust code corresponding to the driver graph in part (d).

Fig. 7: Example showing the major steps of driver generation for a simple List data structure.

reports that no choice of nondeterministic inputs can lead to a memory safety violation, or it produces a counterexample trace containing concrete inputs that lead to an error condition.

III. Approach

Driver generation in CRUST is type-directed, with a separate post-processing step for exposing aliasing behavior. First, CRUST generates well-typed sequences of calls to invoke every library function with every combination of values produced by construction functions, subject to the bound on call sequence length. Second, CRUST merges identical subexpressions and call sequences to expose all possible combinations of aliasing between inputs.

A. Call Sequence Generation

Call sequence generation begins by selecting a target function from the library set. CRUST produces an initial *sketch* [5], consisting of a call to the target function with a hole in each argument position. Figure 7b shows an example. CRUST then proceeds recursively, filling each hole in the sketch with an expression of appropriate type, until either no holes remain or the sketch exceeds the bound on call sequence length (Figure 7c). During this process, each time CRUST needs to make a choice, it separately explores the outcomes of all available options, collecting all resulting drivers into a single set that includes all possible variations.

CRUST's method of generating expressions of primitive and built-in types is straightforward and does not rely on the provided construction functions. For primitive numeric and boolean types, CRUST generates an invocation of nondeterministic choice, relying on the underlying model checker to consider all possible values. For reference types (`&T` and `&mut T`), CRUST generates an application of the appropriate address-of operator to a hole of the referent type (`T`). For tuple types, it generates a tuple construction expression of appropriate arity with a hole in each subexpression position. CRUST could easily support additional built-in expression types, though we did not find these necessary for any of the libraries we tested.

For nonprimitive types, CRUST generates a call to a function from the construction set that has an appropriate return type. CRUST initially generates a hole for each function argument, then later fills each hole by recursively applying the same code generation procedure. The return type of the function need not be an exact match for the desired expression type—CRUST will automatically insert primitive expressions, such as dereferences and tuple element accesses, to extract a value of the appropriate type from the function's result.

CRUST's handling of nonprimitive types allows it to generate drivers involving multiple values of distinct nonprimitive types. For example, a driver call a data structure constructor, then call a method on the data structure to obtain an iterator, in order to test a method of the iterator. This is important because many iterators use unsafe code to traverse the underlying data structure, but iterators themselves often cannot be constructed directly without an existing data structure.

For many complex data types, including most containers, the only method that returns a value of the type is the constructor, which typically returns an empty or otherwise trivial instance. Obtaining a more complex instance requires applying mutating functions such as the `insert` method of a container, to update the data structure in-place. Thus, CRUST includes these mutating functions when generating code to produce values of nonprimitive types. Specifically, when generating code to produce a value of type `T`, CRUST generates a call to a function that produces `T`, then additionally generates zero or more calls to functions that accept `&mut T`, passing a reference to the `T` value so that it can be mutated before use.

Due to the design of the Rust language, CRUST has no need to generate code that uses arithmetic, comparison, bitwise, or array indexing operators. For values of primitive type, these operators are unnecessary because nondeterministic choice already covers the entire space of possible values. For more complex types, the operators are implemented via operator overloading, by defining a specially named method for the type. CRUST can thus cover all behaviors of these operators by invoking the overload method directly, using its existing support for generating function calls. However, this does mean that the expression `x + y` should be considered a function call for purposes of the call sequence bound.

B. Aliasing

The driver generation algorithm described so far explores each distinct value for each function argument, but does not explore the full range of aliasing relationships between arguments. To explore combinations of aliasing behavior, CRUST applies a postprocessing step to each generated driver. This step generates additional drivers by merging identical subexpressions, so that the result of a single evaluation of the subexpression is used in multiple locations (Figure 7d). For example, starting from `f(g(), g())`, CRUST will generate an additional driver `let x = g(); f(x, x)`. When the merged subexpressions have reference types, the resulting driver now uses aliased references where the original used distinct ones. Merging subexpressions may expose aliasing behavior even for subexpressions of non-reference types, as structures often contain references or raw pointers. In particular, data structure iterators typically contain a pointer to the current element, and merging iterator subexpressions results in a pair of iterators pointing to the same element of the same data structure.

C. Code Generation

After generating drivers, CRUST translates both the library and the generated driver functions from Rust to C (Figure 7e). To carry out this translation, CRUST invokes the Rust compiler frontend to parse the code and perform type inference and name resolution, then passes the data to a custom compiler backend that emits C code. This translation is feasible because Rust and C operate at similar levels of abstraction, and most high-level Rust features admit a straightforward implementation in terms of lower-level C code. The remainder of this section discusses the more challenging aspects of the translation.

Lifting: Rust, inspired by functional programming languages, allows control flow structures such as conditionals, pattern matching, and loops to appear within any expression. C, in contrast, allows these structures to appear only in statement positions. CRUST recursively transforms the AST, lifting control flow structures to statement positions, introducing temporaries where necessary to store the resulting values.

Error Handling: In Rust, precondition violations such as accessing an out-of-bounds index of an array may lead to a *panic*, which is similar to an exception. However, Rust does not support any mechanism for user code to catch a panic and continue executing—once a panic occurs, the runtime unwinds the entire stack and terminates the program. No further memory safety violations can occur after the program terminates, so if a program reaches a panic without violating memory safety, CRUST reports no error for that program and input.

TABLE I: Modules and parameters used for evaluating CRUST.

Module	LOC	Driver bound	Filter lines
Vector	1,015	3	4
Slice	1,244	2	4
Ringbuf	1,030	3	4

IV. IMPLEMENTATION

We implemented CRUST as two distinct components. First, a modified `rustc` compiler driver runs the standard parser and frontend passes, followed by a custom backend that emits a high-level intermediate representation. Then, a collection of external tools analyzes and processes the IR, generating driver code or performing translation to C. The `rustc` integration consists of 1,934 lines of code (based on a snapshot of `rustc` taken on April 1, 2015), and the IR processing tools are 3,508 lines of Haskell and 4,380 lines of OCaml. For verification, we used CBMC [4] version 4.9 with the Minisat backend.

Our implementation supports a large subset of Rust features, but it currently lacks support for dynamic dispatch of trait methods and for closures. Code using these features internally cannot be invoked from any CRUST-generated driver. Extending CRUST's Rust-to-C translation to support these features remains as future work.

V. EVALUATION

We evaluated CRUST's ability to detect real memory safety errors by using it to check the vector (`Vec`), slice (denoted `&[T]`), and ring buffer (`VecDeque`) data structures from the Rust standard library. The vector type is a growable, heap-allocated array. The slice type is a view into an array, commonly used to manipulate vectors and other array-based data structures. The ring buffer type is a dynamically-resizable heap-allocated ring buffer. The vector and slice types are essential data structures used by nearly all Rust programs.

For testing, we examined the change history of the Rust standard library and found two memory safety bugs, one in the vector module and one in the slice module. We reintroduced each bug into the modern Rust standard library and ran CRUST on the faulty versions to confirm that it detects the bugs. We also ran CRUST on the unmodified Rust standard library, which has been thoroughly tested and has no known memory safety bugs, to confirm that CRUST does not report false positives.

Table I shows the parameters we used for the evaluation. We wrote a short filter (fewer than five lines) for each module, specifying the entire module as the library set and a small number of selected functions as the construction set. We used the same filter for both the unmodified and faulty versions of each module—in particular, we did not tune the filter for the faulty versions to guide the search toward the known bug.

We set a smaller driver bound when testing the slice module because the unique characteristics of the slice API result in a large number of drivers for a given bound. Specifically, the slice library contains an unusually large number of `unsafe` blocks, and because Rust considers slices to be a special kind of reference, CRUST must check all pairs of functions (quadratic in the size of the library) for aliasing violations.

TABLE II: Results of evaluating CRUST on Rust standard library components.

Module	Has bug?	Drivers generated	Gen. time	Checking time	Errors found
Vector	N	138	2.0m	3.1m	0
Vector	Y	174	2.0m	4.0m	64
Slice	N	9,722	14.5m	446.3m	0
Slice	Y	9,740	14.4m	448.4m	9
Ringbuf	N	2,150	10.6m	121.2m	0

Results: The results of the evaluation are summarized in Table II. Running times are reported in CPU-minutes. In each case, CRUST produced the expected results: no errors on the unmodified library, and multiple errors on each faulty version (due to multiple drivers triggering the single bug).

For the vector library, CRUST requires fewer than 200 drivers to cover the input space of all unsafe vector functions. The vector type supports only two reference-producing functions that contain unsafe code, along with many other methods implemented by calling one of the two functions and performing safe operations on the result. Only the two reference-producing functions require testing by CRUST, and CBMC checking for the resulting drivers took only 4 minutes of CPU time. As each CBMC invocation is independent, the checking phase is embarrassingly parallel. The machine we used for testing (a quad-core Intel Core i7 CPU with 16GB of RAM) supports 8-way parallelism, so the CBMC invocations took only 30 seconds of wall-clock time.

The slice library requires several thousand drivers, for the reasons described previously. But once again, the parallel nature of the checking process allows for speedups: on our test machine, checking all slice drivers took less than an hour of wall-clock time.

Finally, the ring buffer library produces roughly 2,000 drivers, and spends 2 CPU-hours checking them. These results are the most representative out of the three modules we tested. The ring buffer type does not include a disproportionate number of reference-producing functions, nor does it dispatch its unsafe indexing operations to a separate module.

Omitted from the table is the time spent preprocessing each library version to prepare it for testing with CRUST. This step, which took about 3 minutes for our benchmarks, is a one-time cost for each library version to be tested. The results can be reused with a variety of filters and bounds to test multiple portions of the library.

VI. Related Work

CRUST fundamentally relies on the use of a lower-level program analysis tool to explore the space of all primitive-typed inputs. For our implementation, we chose to use CBMC [4], a robust bounded model checking tool for C programs. Other bounded model checkers that can operate on C, such as LLBMC [6], may also be suitable as a backend for CRUST.

CRUST's strategy of generating many drivers to cover the space of possible inputs is similar to techniques for test case generation, which form the basis of tools such as Randoop [7]. A key difference is that CRUST generates sufficient drivers to cover the entire input space of each function under test, subject to a bound on driver size, allowing it to provide a strong guarantee akin to that of bounded model checking. In situations where the provided guarantee need not be as strict, a Randoop-like strategy may be useful to improve performance.

Several previous tools use a similar approach to CRUST, combining test generation with bounded verification. For example, Symstra [8] uses symbolic execution both to generate call sequences that explore the input space for object-oriented data structures and to explore all possible combinations of primitive-typed input values for each call sequence. JPF [9] and Evacon [10] similarly combine different forms of test generation to generate non-primitive inputs with model checking or concolic testing to handle the primitive inputs. Compared to these techniques, CRUST produces not only sequences of method calls on a single object, but also drivers that create and manipulate values of Rust's nonprimitive built-in data types and operate on multiple values of different user-defined types. This flexibility allows CRUST to analyze more complex APIs, particularly those involving "helper types" such as the iterator types associated with a data structure.

VII. Conclusion

We have presented CRUST, a tool for detecting memory safety errors and aliasing invariant violations in unsafe Rust code. CRUST uses a combination of test generation and bounded model checking, allowing it to efficiently conduct exhaustive exploration of large input spaces. CRUST's test generation strategy is type-directed and uses several conservative heuristics to avoid generating redundant test harnesses. We have demonstrated CRUST's effectiveness by testing it on portions of the Rust standard library. CRUST easily detects memory safety bugs, even those involving complex interactions between multiple library components, with no programmer annotations.

References

[1] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *PLDI*, 2006.

[2] The Rust Project Developers, "The Rust programming language," 2015. [Online]. Available: http://www.rust-lang.org

[3] E. Reed, "Patina: A formalization of the Rust programming language," University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02, March 2015.

[4] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, 2004.

[5] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, 2006.

[6] F. Merz, S. Falke, and C. Sinz, "LLBMC: bounded model checking of C and C++ programs using a compiler IR," in *VSTTE*, 2012.

[7] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*, 2007.

[8] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *TACAS*, 2005.

[9] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," in *ISSTA*, 2008.

[10] K. Inkumsah and T. Xie, "Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs," in *ASE*, 2007.