Type Checking beyond Type Checkers, via Slice & Run

Justus Adam
University of Kent
School of Computing
Canterbury, United Kingdom
me@justus.science

Stephen Kell
University of Kent
School of Computing
Canterbury, United Kingdom
S.R.Kell@kent.ac.uk

Abstract

Type checkers are the most commonly used form of static analysis, but their design is coupled to the rest of the language, making it hard or impossible to bring new kinds of reasoning to existing, unmodified code. We propose a novel approach to checking advanced type invariants and properties in unmodified source code, while approaching the speed and ease of simple, syntax directed type checkers. The insight is that by combining a deep program analysis (symbolic execution) with a cheaper program abstraction (based on program slicing), it appears possible to reconstitute type-checking in the context of an *underapproximate* analysis. When the program's 'type level' can be opportunistically disentangled from the 'value level', this is done by the program abstraction step, in some cases removing the underapproximation. We implement a simple prototype that demonstrates this idea by checking the safety of generic pointers in C, pointing to benefits such as safe homogeneous and heterogeneous generic data structures.

CCS Concepts: • **Software and its engineering → Software safety**; *Polymorphism*; *Model checking*; • **Theory of computation → Program analysis**; **Abstraction**; *Invariants*; *Assertions.*

Keywords: static analysis, type checking, symbolic execution, program slicing

ACM Reference Format:
Justus Adam and Stephen Kell. 2020. Type Checking beyond Type Checkers, via Slice & Run. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Tools for Automatic Program Analysis (TAPAS '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3427764.3428324

1 Introduction

Simple syntax-directed type-checkers are arguably the most successful family of static analyses in existence. However, as traditionally conceived they are also limited in applicability and limiting to programmers. In applicability, they are limited to a carefully constrained correctness property that is coupled tightly to a wider language design. For programmers, they bring the reality of compromising their code for the sake of pleasing the checker: either factoring their code in less-than-ideal ways (e.g. by duplication), or by using features that circumvent checking, such as casts that are checked only dynamically or not at all. Research on more advanced type systems has generally progressed by co-design with newer languages, in which type-checking involves a deeper analysis (often flow-sensitive) but is designed to be efficient and sound. In this paper we describe ongoing work towards a radically different approach, which retains the core benefits of type-checking but *without* disrupting existing language designs, and offering new trade-offs in the areas of code factoring and check circumvention.

The core idea is to start with a very deep analysis—symbolic execution—and ask how to *recover* the efficiency and soundness of conventional type checking, *for those programs* that are syntactically amenable to it (so would pass a conventional check), while also still providing value for programs that are *not* so amenable. The latter are cases where code is more 'gnarly', perhaps owing to casts or to code structure that defeats a traditional checker. Our analysis can still proceed usefully on these—albeit perhaps more slowly and/or with loss of a soundness guarantee (since the symbolic execution is underapproximate).

Our contributions are the following.

- We introduce an alternative formulation of static type checking as symbolically executing an instrumented program. The instrumentation adds 'type assertions' (§3), systematically, according to an *invariant protocol.*
- We outline a *program abstraction* (§4), based on program slicing [28], to recover performance and (in some cases) termination. We also sketch how to generalise the invariant protocol to cover Hindley-Milner-style polymorphism.
- Lastly we present a prototype (§5) based on Klee that checks the safety of casts in C. It demonstrates the feasibility of the approach and motivates further improvements to the program abstraction.

2 Motivation

Consider C and Java: two widely-used languages for building highly dependable, long-lived systems. Both perform simple, syntax-directed static type checking, which developers 'opt out of' in at least the following ways.

- in C, parametrically polymorphic data structures are expressed using `void *` and unchecked downcasts;
- in C, extensibility frequently involves `void *` and casting (e.g. when registering callbacks, or decorating library-defined structures with client-specific data);
- in Java, some older code also uses downcasts from Object, as does much reflective code;
- in C, algebraic data types are simulated using unchecked `union`;
- in Java, idioms for simulating pattern-matching often use 'type case' constructs performing downcasts.

Mastrangelo et al. [19] surveyed different uses of casts in Java, finding that 8.7% of methods contain casts, of which 50% of which are not locally protected against errors. Polymorphism in C code has been studied by Hackett and Aiken [10] and is especially prevalent in systems code. In C, casts are particularly dangerous because there is usually no check at all, even at run time. In our work so far, we focus on C, although our technique is by no means specific to C.

Although one could imagine extending the type-checking in Java or and C to cover most of the patterns above, doing so would need to elicit additional information from the programmer, hence create a need to extend the language. Such an approach was taken, for example, in CCured [20], Cyclone [12] and Checked C [22] projects. We instead seek an analysis that is able to check statically the safety of casts without modification of the language syntax or addition of annotations. Such an analysis must (somehow) track the 'real' run-time type behind any value having a generic type like `void *` (or `Object`). In general, this may depend on program flow—for example, the run-time type behind a `void *` function parameter may *in extremis* be different for every distinct program path reaching that function. This means we need a whole-program, path-sensitive analysis if our checking is to be complete. For this we use symbolic execution [4, 6, 17]. Modern symbolic executors such as KLEE [5] perform object-level modelling of memory, and this model is naturally amenable to extension with per-object type information.

Program Slicing. To combat the poor scalability and underapproximation of symbolic execution, we investigate *program slicing* [28] as a program abstraction technique. Slicing 'reduces' programs by removing 'uninteresting' parts, according to a defined *criterion*—often a program location and a set of variables defined at that location. The slicer will follow data and control flow dependencies backwards from these variables, and transitively preserve anything that might influence them; everything else is discarded.[1] Usually slicing uses fast, syntax-driven algorithms that are overapproximate, although a variety of algorithms are available [27]; computing the smallest slice in general is undecidable.

How much of the original program is discarded depends on the complexity of the dependency structure leading to the sliced-on expressions. In our case, we are interested in the 'real' run-time type behind a generic pointer, but we often do not need the data stored behind the pointer. The slicer can thus discard much of the 'work' of the program, leaving only what influences the type of data found or used in a generic structure. We expect that straightforward code, which rarely modifies types, should be mostly discarded by the slicer, producing a small program which is fast to symbolically execute. 'Gnarly' parts, perhaps featuring lots of casts, generic pointers, and so on, will leave a larger program behind, hence create more work for the more expensive analysis. In this sense we target a 'pay-as-you-go' property.

3 Finding Type Errors by Symbolic Execution

Consider the following linked-list code.

```c
struct list {
  void *p;
  struct list *next;
};
int f(struct list *l) {
  int max = -1;
  while (l) {
    int i = *((int*)l->p);
    max = i > max ? i : max;
    l = l->next;
  }
  return max;
}
```

To allow symbolic execution to find failures of the pointer cast, we augment the engine to carry type information for every object. This allows us to instrument the program with *type assertions*, which the symbolic execution engine can check much like any other assertion. This instrumentation is done systematically; systematic insertion of assertions and assumptions is sometimes called an invariant protocol [26]. The following is an illustration (not concretely reflective of our implementation).

```c
int f(struct list *l) {
  Type _type_at_l =_type_list; // from prototype
  int max = -1;

  while (l) {
    assert(type_at(l->p) // required
      == _type_int);      // before cast
    int i = *((int*)l->p);
    max = i > max ? i : max;
    assume(type_at(l->next) // consequence of
          == _type_list      // invariants on l
       || type_at(l->next)   // and field 'next'
          == _type_None);
```

[1]Formally the slicer will ensure that the output program behaves the same as the input program with respect to the slicing criterion. This means that during any execution the values of the selected variables, at the selected locations, would be the same in both programs.

```
    l = l->next;
    _type_at_l =          // consequence of
      type_at(l->next);   // assignment
    assert(_type_at_l
          == _type_list   // required after
      || _type_at_l       // assignment to l
          == _type_None);// (preserve invariant)
  }
  return max;
}
```

Such an approach can allow us to find bad downcasts ahead of time. This is already useful, since a conventional type-checker will not find such bugs. However, it is not yet a substitute for a conventional type checking, because it is both slow and does not terminate for certain programs.

4 Slicing as a Program Abstraction

To recover something performing comparably to type-checking we could try to combine it with the right *program abstraction.* Unlike conventional checkers, which require all programs to pass the check, our goal is to recover the scalability of syntax-directed checking *on those programs that allow it*, but still permit analysis of other code that is too gnarly for a traditional checker to reason about. Similar to previous work [23, 25] we rely on program slicing [28] to simplify the program ahead of symbolic execution. Since *types* exist 'at the meta level', and the *value-level* computation only *infrequently* makes choices influencing those types, we might expect to slice away most of the computation and be left with a 'types-only program', i.e. the type assertions and reified types with which we augmented the program. For syntactically type-checkable programs, types are by definition straightforwardly separable from 'value-level' computation; for example, types are invariant around a loop. The ability to eliminate loops in our technique is highly desirable, because loops are a frequent source of non-termination for symbolic execution.

4.1 Reformulating Type-Checking

Our experience so far suggests that given the right approach to augmentation, and a specially crafted slicer, we can reformulate conventional type-checking in at least some cases as symbolically executing a loop-free program that was produced by slicing. To illustrate, consider a simpler, monomorphic version of the program. Since it does not have the problematic cast, it should be straightforward to check statically. (Note that our earlier example is intentionally *not* checkable by a simple Java-style type checker: the downcast is left for dynamic checking, and indeed it is possible to construct a heterogeneous list that would cause a type error at run time.)

```
struct list {
  int *n; // Now not declared as void*
  struct list *next;
};

int f(struct list *l) {
  int max = -1;
  while (l) {
    int i = *l->n;
    max = i > max ? i : max;
    l = l->next;
  }
  return max;
}
```

After augmentation with reified types and assertions, this becomes the following.

```
 1 int f(struct list *l) {
 2   Type _type_at_l =_type_list; // from prototype
 3
 4   while (l) {
 5     int i = *l->n;
 6     max = i > max ? i : max;
 7     assume(type_at(l->next) // consequence of
 8           == _type_list     // invariants on l
 9       || type_at(l->next)   // and field 'next'
10           == _type_None);
11     _type_at_l =          // consequence of
12       type_at(l->next);   // assignment
13     l = l->next;
14     assert(_type_at_l
15           == _type_list   // required after
16       || _type_at_l       // assignment to l
17           == _type_None);// (preserve invariant)
18   }
19   return max;
20 }
```

Using purely local reasoning we can deduce that `_type_at_l` is always `_type_list`. Furthermore the assumption bounds `type_at(l->next)` to either `_type_list` or `_type_None`. This lets us rewrite the program as follows.

```
 1 void f(struct list *l) {
 2   assume(_type_at_l       // consequence of
 3         == _type_list);   // function prototype
 4
 5   while (nondet()) {
 6     int i = *l->n;            // SLICED
 7     max = i > max ? i : max;  // SLICED
 8     assume(type_at(l->next) // consequence of
 9           == _type_list     // invariants on l
10       || type_at(l->next)   // and field 'next'
11           == _type_None);
12     l = l->next; // SLICED
13     _type_at_l = nondet(           // consequence of
14         _type_list, _type_None);   // assignment
15     assert(_type_at_l
16           == _type_list   // required after
17       || _type_at_l       // assignment to l
18           == _type_None);// (preserve invariant)
19   }
20 }
```

It is now apparent that lines 6, 7 and 12 do not affect the types, and can be removed. After that, the loop body of this program is trivially invariant: `_type_at_l` is either `_type_list` or `_type_None`, before and after the loop. Since this is exactly the property being asserted, both assertion and loop can also be removed prior to symbolic execution.

Clearly this is not an off-the-shelf slicer, but rather one which also performs certain elementary program abstractions, extending the slicer's usual local reasoning. In this example we abstracted a type-level assignment into nondeterministic choice (line 12; contrast line 13 in the previous listing), exploiting the bound established by a previous assumption. This is an overapproximation which, in case of amenable code, such as in this example, allows more code to be sliced away. This is analogous with existing *amorphous* slicing techniques [11]. Clearly our approach also relies on careful design of the augmentation with reified types—we make use of the modelling of null pointers as the distinct _type_None—and of the invariant protocol for adding the assertions. The protocol defines a set of rules for when an invariant must be checked, and when (as a consequence) it may be assumed. Again, the invariant and its protocol must be designed carefully—here we use a simple 'assume on read, check on write' protocol for pointed-to types. This allows us to assume soundly that on reading a pointer-to-list, we always obtain null or the address of a list, in return for always asserting the equivalent property on writing.

4.2 Polymorphism

This seems like a lot of effort to reformulate a very pedestrian check over monomorphic code, which is already handled by even basic type-checkers. The value of this approach is in its potential for extensibility: by tweaking individual parts of this pipeline, we can accommodate different disciplines and their invariants. We will sketch how to extend the approach to handle Hindley-Milner-style polymorphism, revisiting our opening example. The idea is to strengthen the invariant to include a 'type-homogeneity' property across the various generic pointers in the list, making the corresponding changes to the invariant protocol. The example now uses a generic pointer (void*) and shows *only* newly added instrumentation, for brevity.

```
 1 struct list {
 2   void *n; // back to generic void*
 3   struct list* next;
 4 }
 5
 6 int g() {
 7   struct list *l = /* pseudo-syntax */ [&1, &2, &3];
 8   // .n field points to int, a subtype of void
 9   Type _type_at_l_n = _type_int; // from assignment
10
11   while (l) {
12     assume(type_at(l->n)  // homogeneity assumption
13         == _type_at_l_n); // (checked at
14                           // list construction)
15     assert(_type_at_l_n == _type_int); // required by
16     int i = *(int*)l->n;               // <- downcast
17     max = i > max ? i : max;
18
19     l = l->next;
20     assert(type_at(l) == _type_None    // preserve
21         || type_at(l) == _type_list); // invariant on l
22   }
23   return max;
24 }
```

Here the loop expects the type _type_int but the list declaration only guarantees the more general void, so our original invariant protocol's assumption is not enough to remove the assertion (and hence the loop) at slicing time. However the new assumption in line 12 tells us that the type of the pointer field n is the same for every node in the list. List construction, although not shown, is correspondingly instrumented to assert this. This in turn allows the downcast check to be eliminated, provided that the slicer can reason one iteration backward around the loop back-edge (e.g. by unrolling the loop once), to see that either the assumption on line 15, before the loop, implies the asserted property or the previous iteration's assumption does.

The above example provides a very minimalist illustration of a homogeneity assumption, and further development is needed to make it a realistic polymorphic example. First we would refactor it into a polymorphic function containing the loop, separate from the monomorphic invocation on an int list. This would require passing a compar callback, responsible for the downcast, rather than the current loop's direct use of > and (int*). In turn, this gives us an inter-procedural and higher-order example: to make the callback invocation itself sliceable requires some form of summary of its type-correctness precondition, suitable for assertion-checking in terms of _type_at_l_n. Again, such a check would occur outside the loop, at the call which *passes* the callback, much as a Hindley-Milner-style checker would check the use of an int -> int function being passed into a polymorphic 'a -> 'a context. Further work is needed, including extending our reified type information with function summaries of this kind.

4.3 Directions

We believe unpacking type-checking into this combination of *augmentation*, *invariant protocol* and *slicing algorithm*, may be fruitful in several ways.

Configurability. It opens up type-correctness properties to a spectrum of analysis methods that is potentially configurable in the sense of Beyer et al. [2], from fast-but-shallow to slow-but-deep. Ultimately this provides more means of catching errors statically, rather than deferring checks until run time (Java-style) or skipping them altogether (C-style), without redesigning the whole language. It also allows analysis methods to be chosen according to other advantages; for example, symbolic execution, although slow (on unabstracted code), provides precise counterexamples, which programmers may prefer to type-checkers' sometimes obscure error messages.

Language-Neutrality. The use of invariants allows us to explore the design space of type-checking across language boundaries. For example, what if variable declared as a pointer-to-list could no longer be relied on to be either null or point to a list? This sort of situation arises in cross-language scenarios (imagine sharing our list with Python code, say), since different languages may preserve a different invariant (in Python's case, any reference is fair game to point to any object). Given a different invariant protocol, checking may still be tractable (say, checking on loads rather than on stores), or the code in the 'more relaxed' language code may be instrumentable with assertions that effectively 'tighten' it to the other language's invariant, perhaps at a cost in flexibility.

Generalisability. Extending our invariant protocol towards Hindley-Milner-style polymorphism raises the question of whether this extends to other 'fancy' type systems. Dependent types [7, 29] and linear types [1, 18], for instance, have been shown to have a number of benefits for low-level programming, but are found in few mainstream programming languages. These require flow sensitive checking, which symbolic execution already provides.

Application-Friendliness. It suggests a path by which *application-level* invariants can enhance or accelerate static checking, by being incorporated into the protocol. Subsequent analysis steps, including slicing and symbolic execution, need not be limited to properties baked into the language, and may benefit from knowledge of user-defined disciplines. This may ultimately resemble liquid types [21], where the programmer can programmatically specify invariants which are statically checked.

5 Ongoing Work

Our work is embodied in a new compile-time analysis tool for statically checking the type-correctness of C code that uses pointer casts.[2] The tool runs on unmodified source code, and requires no additional programmer annotations. It instruments the program to represent and check types explicitly, slices the program on those checks, yielding a smaller (abstracted) program which is then analyzed with a symbolic execution engine.

This approach raises at least three key research questions: whether it can recover the same checking power as conventional type checking (on those programs amenable to it); whether it adds useful additional checking power (on those that aren't); and whether it can be implemented efficiently enough for frequent 'fast-feedback' use. The latter is especially important. Type checkers have long been designed for fast feedback, and Distefano et al. suggest that fix rate for issues discovered by static analysis is markedly higher if issues are reported close to when the code was written [9].

5.1 Implementation

Our prototype uses assertion-based instrumentation similar to that in §3, including an `assert` before every downcast. These trigger type checks during symbolic execution, but are also used by the slicer as its slicing criterion (i.e. it tries to preserve only code that these assertions depend on).

Symbolic execution is provided by KLEE. We augmented the KLEE memory abstraction to record a precise type for each allocated memory region. The type information is generated for each allocation site by a system developed in prior work [14, 15]. Upon new allocation of symbolic memory by KLEE we retrieve the type information and record it for later reference. When a type assertion is encountered, KLEE's built-in memory facilities are used to dereference the symbolic pointer to a set of memory objects and offsets. Using the layout information for the type of the memory region, we can calculate the type at each offset and check the assertion holds.

For this prototype one may think of subtyping in C as "zero-offset containment". More specifically, we follow the approach of previous work [14, §5.1 and §6].

5.2 Results

Our prototype cannot yet provide comprehensive results for our approach in general, but it supports running simple examples. These show both the strength and precision of symbolic execution, and also weaknesses of using an unmodified slicer, further motivating the domain-specific enhancements described in §4.

Slicing. Currently we use an unmodified slicer from the Frama-C [8] suite. We have discovered that it is capable of identifying unrelated sections of code, but not even able to deduce that we are only interested in addresses, not pointed-to values. Consider this type-invariant example

```
 1 int main() {
 2   int unused = 0; // Sliced
 3   int* used = malloc(sizeof(int));
 4   *used = 0;
 5   for (int j; /* arbitrary */) {
 6     *used = *used + 5;
 7     unused += j; // Sliced
 8   }
 9   void* generic = used;
10   assert(type_at(generic) == _type_int);
11   return unused + *((int*) generic); // Sliced
12 }
```

The slicer removes lines 2, 7 and 11. It correctly determined that `unused` does not influence the type assertion on line 10. However it is rather obvious that all the assignments to `*used` do not change its type. We hope to achieve better result, particularly with respect to loops, once we augment the slicing algorithm with domain specific knowledge, as described in §4.

Symbolic Execution. Our experience has already shown that we are able to detect errors (or their absence) even in gnarly code. We use the same example as §4 but additionally made the list heterogeneous, by mixing list elements of type `int` and `double`.

```
int g() {
  struct list *l = /* pseudo syntax */ [&1.5, &2];
  int max  = 0;
  int even = 0;

  while (l) {
    int i;
    if (even) {
      assert(type_at(l->n) == _type_int);
      i = *l->n;
    } else {
      assert(type_at(l->n) == _type_double);
      int i = ceil(*((double*) l->n));
```

[2]Besides pointer casts, any other language feature that currently 'bypasses' static type-checking would be in scope—in C, this would include union access, variadic function calls, `memcpy()` and some stores to memory. However, we focus on pointer casts as the most common of these.

```
    }
    max = i > max ? i : max;
    l = l->next;
    even = !even;
  }
  return max;
}
```

This program is type-correct, because even is used to determine whether we should treat the element as a pointer to int or double. The Hindley-Milner-style invariant protocol discussed earlier cannot accommodate such code, because it expects homogeneity. In this example one if branch expects _type_double and the other _type_int. This leads the slicer to pass the instrumented code unchanged to symbolic execution, which reports no error.

In contrast, imagine that we forgot to !even at the end of the loop, making the program type-incorrect. Again, the loop is not sliced away, and we verified that symbolic execution reports a type error in the second iteration. In the future we plan to explore further difficult code patterns like this, and also larger examples in general. (Currently our tool has no support for multi-file programs; this will require build-time propagation of additional metadata, but we are working to add this.)

6 Related Work

Our technique is thematically related to gradual typing [24], but is distinct: even in the case of unmodified code that is not factored for static checking, our technique performs ahead-of-time analysis that may find type errors, whereas gradual typing relies on dynamic checking.

Our technique may be seen as a form of abstract interpretation, albeit a peculiar one. Traditional applications of abstract interpretation abstract the domain of program *values*, seek a sound overapproximation of program behaviour, and must select an abstract domain carefully to allow this. By contrast, our slicing-based technique abstracts over both value domains and program structure, uses the types directly from the program, and is prepared to tolerate underapproximation on gnarlier code.

Kahrs [13] previously applied abstract interpretation to infer types for Milner-style polymorphic programs. Our technique takes this a step further and removed the constraint to polymorphism. We instead check general type assertions, allowing for dependent type style constructs as well as potentially allowing checking of application-specific properties.

Our technique also shares themes with bounded model checking [3], but differs in a key detail: our abstraction technique's potential for slicing away program structure, especially loops, means its exploration of at least some infinite-state programs need not be underapproximate. The spectrum between model-checking and abstract-interpretation approaches has been remarked before, motivating a proposal of configurable analysis [2]. Although our analysis is only as configurable as its symbolic execution engine, it shares this theme of taking a property (here type-correctness) and finding a 'broad spectrum' means of checking it, reaching diverse depth and precision trade-offs for diverse programs.

Khoo et al. [16] explored a technique for mixing symbolic execution and type checking, although very different in form: the programmer annotates the program blockwise to specify whether the deeper or shallower analysis is required, and the analysis engine handles merging of analysis results at the boundaries. The goal is to enable better precision/efficiency trade-offs while analysing novel properties (e.g. nullness, in the example) and retaining soundness, at the cost of added annotation effort. In contrast, our use of slicing focuses on standard type-correctness properties, and is happy to forgo soundness in exchange for zero annotation effort; we effectively allow the code's 'sliceability' to decide precision, and to recover efficiency to the extent possible. The use of programmer annotation is likely complementary, e.g. similar block-level annotations may provide a structured means of recovering reliable soundness under our approach.

Slicing as a 'preprocessor' for symbolic execution is also used by Slaby et al. [25], where error states are detected by instrumention that is then used as a slicing criterion. Similarly Shoshitaishvili et al. [23] created slices from locations where the program performs inputs to locations where privileged information may be accessed. In both cases the slices then pass onto symbolic execution, suggesting that slicing before symbolic execution is a more widely applicable pattern.

7 Conclusion and Future Work

In this paper we have argued that there is value in reformulating static type-checking in a more flexible form, where gnarlier code may be accommodated and more complex disciplines do not require writing code in new languages. We have identified a path to achieving this without requiring modifications to existing code, by combining a deep analysis with a cheaper program abstraction to recover performance and precision opportunistically. We described a prototype for checking cast safety in C, which shows that the deep analysis is powerful, but more work is required to craft a custom slicing abstraction that can handle real coding idioms.

Recovering efficiency for symbolic execution largely depends on eliminating as much of the code as possible beforehand. To make our approach feasible in larger codebases, the next step is to implement the slicer augmentations we described in §4. A similar tool, described in [25], reports check times of around 2 seconds, which makes us hopeful that we can achieve similar. We also expect that the local reasoning outputs of the slicer may be computable incrementally. It remains to be seen whether it is possible also to incrementalise slicing itself. Both could yield substantial improvement in edit-compile-run performance of the tool.

References

[1] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 156–161. https://doi.org/10.1145/3102980.3103006

[2] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. 2007. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Werner Damm and Holger Hermanns (Eds.). Springer, Berlin, Heidelberg, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

[3] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, W. Rance Cleaveland (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 193–207.

[4] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECTǓ2014a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*. Association for Computing Machinery, New York, NY, USA, 234Ǔ2013245. https://doi.org/10.1145/800027.808445

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, USA, 209–224.

[6] L. A. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering* 2, 3 (May 1976), 215Ǔ2013222. https://doi.org/10.1109/TSE.1976.233817

[7] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. 2007. Dependent Types for Low-Level Programming. In *Programming Languages and Systems*, Rocco De Nicola (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 520–535.

[8] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C. *Lecture Notes in Computer Science* (2012), 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

[9] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

[10] Brian Hackett and Alex Aiken. 2011. Inferring Data Polymorphism in Systems Code. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. Association for Computing Machinery, Szeged, Hungary, 332–342. https://doi.org/10.1145/2025113.2025159

[11] Mark Harman, David Binkley, and Sebastian Danicic. 2003. Amorphous Program Slicing. *Journal of Systems and Software* 68, 1 (Oct. 2003), 45–64. https://doi.org/10.1016/S0164-1212(02)00135-8

[12] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

[13] Stefan Kahrs. 1992. *Polymorphic Type Checking by Interpretation of Code*. Technical Report. University of Edinburgh, Laboratory for Foundations of Computer Science.

[14] Stephen Kell. 2015. Towards a Dynamic Object Model within Unix Processes. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. Association for Computing Machinery, New York, NY, USA, 224–239. https://doi.org/10.1145/2814228.2814238

[15] Stephen Kell. 2016. Dynamically Diagnosing Type Errors in Unsafe Code. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 800–819. https://doi.org/10.1145/2983990.2983998

[16] Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. 2010. Mixing Type Checking and Symbolic Execution. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 436–447. https://doi.org/10.1145/1806596.1806645

[17] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[18] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys '17)*. Association for Computing Machinery, New York, NY, USA, Article 1. https://doi.org/10.1145/3124680.3124717

[19] Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. 2019. Casting about in the Dark: An Empirical Study of Cast Operations in Java Programs. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 158:1–158:31. https://doi.org/10.1145/3360584

[20] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. Association for Computing Machinery, New York, NY, USA, 128–139. https://doi.org/10.1145/503272.503286

[21] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, Tucson, AZ, USA, 159–169. https://doi.org/10.1145/1375581.1375602

[22] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. 2019. Achieving Safety Incrementally with Checked c. In *Principles of Security and Trust*, Flemming Nielson and David Sands (Eds.). Springer International Publishing, Cham, 76–98.

[23] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2015.23294

[24] Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. *Lecture Notes in Computer Science* (2007), 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

[25] Jiri Slaby, Jan Strejček, and Marek Trtík. 2013. Symbiotic: Synergy of Instrumentation, Slicing, and Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 630–632.

[26] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. 2009. The Need for Flexible Object Invariants. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO '09)*. Association for Computing Machinery, New York, NY, USA, Article 6. https://doi.org/10.1145/1562154.1562160

[27] Frank Tip. 1994. *A Survey of Program Slicing Techniques*. Technical Report. CWI (Centre for Mathematics and Computer Science), NLD.

[28] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (July 1984), 352–357. https://doi.org/10.1109/tse.1984.5010248

[29] Dengping Zhu and Hongwei Xi. 2005. Safe Programming with Pointers Through Stateful Views. In *Practical Aspects of Declarative Languages (Lecture Notes in Computer Science)*, Manuel V. Hermenegildo and Daniel Cabeza (Eds.). Springer, Berlin, Heidelberg, 83–97. https://doi.org/10.1007/978-3-540-30557-6_8