
Carving Parameterized Unit Tests

Alexander Kampmann and Andreas Zeller

{kampmann, zeller}@cispa.saarland

CISPA / Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

arXiv:1812.07932v1 [cs.SE] 19 Dec 2018



Carving Parameterized Unit Tests

Alexander Kampmann and Andreas Zeller

{kammann, zeller}@cispa.saarland

CISPA / Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

(Dated December 20, 2018)

Abstract

We present a method to automatically extract (“carve”) parameterized unit tests from system executions. The unit tests execute the same functions as the system tests they are carved from, but can do so much faster as they call functions directly; furthermore, being parameterized, they can execute the functions with a large variety of randomly selected input values. If a unit-level test fails, we lift it to the system level to ensure the failure can be reproduced there. Our method thus allows to focus testing efforts on selected modules while still avoiding false alarms: In our experiments, running parameterized unit tests for individual functions was, on average, 30 times faster than running the system tests they were carved from.

1 Introduction

Tools and methods for software test generation can be distinguished by the *level* at which they feed generated data into a program. At the *unit level*, test generation operates by invoking individual functions, allowing for effectively narrowing down the scope of analysis and execution, while at the same time making internals directly available for testing. The downside, however, is that synthesized function calls may *violate implicit preconditions*: If a test generator finds that `sqrt(-1)` crashes, this does not help developers who never intended `sqrt()` to work with negative numbers anyway. When generating tests at the *system level*, this problem of false failures does not occur, as a system is expected to reject all invalid inputs; and any failure caused by third-party system input needs to be fixed. On the other hand, system-level testing must read, decompose and process inputs, before the function of interest is finally reached. This leads to *overhead*, as compared to a unit-level test. Furthermore, effective system test generation is often hampered by scale: symbolic analysis, for instance, hardly scales to system sizes.

In this paper, we present a method that joins the benefits of both system-level and unit-level test generation, while at the same time avoiding their disadvantages. Our key idea is based on the concept of *carving unit tests* [4], observing system executions to extract unit tests that replay the previously observed function executions in context. However, we extend the concept by extracting *parameterized unit tests*, allowing to replay not only the original function invocations, but also to synthesize several more. To this end, we identify those function arguments that are directly derived

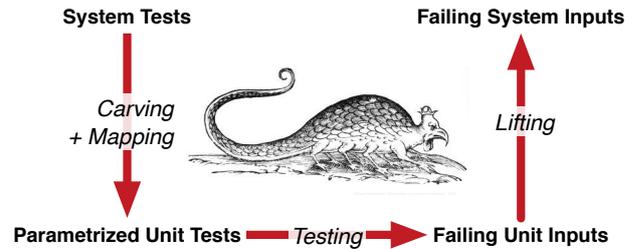


Figure 1: Overview of our approach. Starting from a set of (given or generated) system tests, our BASILISK prototype extracts (“carve”) parameterized unit tests, each representing a function with symbolic parameters in the context observed during system testing. The unit tests are then exhaustively tested. Arguments found to cause failures at the unit level are then lifted back to the system level, using a previously established mapping between function arguments and identical values occurring in the system input; the new system test is re-run to ensure it reproduces the failure. The carved parameterized unit tests can be obtained from arbitrary system test generators, and be explored with arbitrary unit test generators.

from system input. These arguments then become unit test parameters, allowing for extensive fuzzing with random values. We can thus random test individual functions with hundreds of values, with all invocations in the context of the original run, but without requiring the overhead of starting the program anew for each system test run. Furthermore, we can generate tests for any subset of unit tests as we like, spending testing time on error-prone or recently changed functions.

As an example of a carved parameterized unit test, consider the function `bc_add()` from the `bc` calculator program. `bc_add()` accepts two numbers, `n1` and `n2`, encoded in `bc`’s internal representation for numbers with arbitrary floating-point precision, and writes the sum of those two numbers to the number pointed to by `result`. `scale_min` gives the minimal number of floating-point positions to be used by `result`.

```
1 void bc_add (bc_num n1, bc_num n2, bc_num *
      result ,
2 int scale_min);
```

From an execution of `bc` with a concrete input (say, “1 + 2”), our BASILISK implementation observes the call `bc_add(1, 2, &result, 0)`. It identifies 1

and 2 as coming from system input, and thus makes them parameters of the carved parameterized unit test `test_bc_add()`:

```
1 void test_bc_add(int p1, int p2) {
2 // set up the context
3 bc_num n1;
4 bc_int2num(&n1, p1);
5 bc_num n2;
6 bc_int2num(&n2, p2);
7
8 // call the function under test
9 bc_num result;
10 bc_add(n1, n2, &result, 0);
11 }
```

We can now call `test_bc_add()` with random values for `p1` and `p2`, thus testing it quickly without having to start the `bc` program again and again:

```
1 test_bc_add(337747944, 352295539);
2 test_bc_add(535612873, 790525737);
3 // ... and more
```

Feeding random values into a function call brings the risk of violating implicit preconditions, even in the context of a concrete run. In the case of a function failing, we thus *lift* the failing unit test back to the system level; we can do so because we know where the original function argument came from in the system input. Only if the failure can also be repeated at the system level do we report the failure. Lifting is also useful if one is interested in *coverage*: If a unit test achieves new coverage, we lift it to the system level, and verify whether the new coverage also applies there.

In our example, let us assume that `test_bc_add(10, 20)` fails. From the original run, we know that the arguments `p1` and `p2` correspond to the values 1 and 2 in the system input. In the input, we would thus replace the values 1 and 2 by the failure-inducing values of `p1` and `p2`, resulting in the input `10 + 20`. Only if `bc` fails on this input would we report the failure. The lifting process thus gives us (1) system inputs that fail (or obtain coverage), (2) the same zero false positive rate as system tests, yet (3) the speed and convenience of unit-level testing.

The overall interplay of carving, testing, and lifting is sketched in Figure 1. As our approach is independent from a specific system-level or unit-level test generator, it allows to combine arbitrary system-level test generators (random, mutation-based, grammar-based, ...) with arbitrary unit-level generators (random, symbolic, concolic, ...). In this paper, we explore one such combination, paving the way for many more that bring the best of both worlds.

The remainder of this paper is organized as follows. After discussing the background (Section 2), this paper details its individual contributions:

1. We present our approach for carving unit tests out of C programs (Section 3).
2. We show how to identify *parameters* from system input, thus carving *parameterized unit tests* (Section 4).
3. We demonstrate how to fuzz C programs with random values in a carved context (Section 5).

4. We show how to *lift* failed unit tests to the system level for validation (Section 6).

In Section 7, we evaluate BASILISK on a series of C programs. We find that carving parameterized unit tests can yield more coverage in less time when compared against system-level testing; these savings are multiplied when focusing the testing effort on a subset of the program. Section 8 closes with conclusion and future work.

2 Background

2.1 Carving Unit Tests

Carving unit tests was introduced by Elbaum et al. [4] as a means to speed up repeated system tests, for instance in the context of regression testing. Elbaum’s work used the Java infrastructure, serializing and deserializing objects to enable faithful reproduction of units in context.

In contrast to this, our implementation carves unit tests from C programs. Instead of serialization, we use a heap traversal to record heap structures, and generate code to recreate a similar heap. To the best of our knowledge, BASILISK is the first tool to implement any kind of unit test carving for C.

Also we extended the carving mechanism to carve *parameterized unit tests* [17], where function parameters whose values can be mapped to system input are left as parametric and thus open for unit test generators to explore.

2.2 Extracting C Memory Snapshots

Interpreting and recovering C data structures at runtime is notoriously difficult, since every programmer can implement not only her own data structures, but also her individual memory management. The work of Zimmermann and Zeller [21] on extracting and visualizing C runtime data structures (“memory graphs”) as well as their later application in debugging [20, 3, 13] is related to ours in that all these works attempt to obtain a reliable and reproducible snapshot of C data structures. Yet, all these works use these data structures for the purposes of debugging and program understanding rather than carving.

2.3 Generating System Tests

The idea of *generating software tests* is an old one: To test a program *S*, a producer *P* that will generate inputs for *S* with the intent to cause it to fail. To find bugs, a producer need not be very sophisticated; as shown in the famous “fuzzing” paper of 1989, simple random strings can quickly crash programs [11].

To get deeper than scanning and parsing routines, though, one requires syntactically correct inputs. To this end, one can use formal specifications of the input language to generate inputs—for instance, leveraging *context-free grammars* as producers [14]. The LANGFUZZ test generator [9] uses its grammar for *parsing* existing inputs as well and can thus combine existing with newly generated input fragments.

Today’s most popular test generators take *input samples* which they mutate in various ways to generate further inputs. *American Fuzzy Lop*, or AFLFuzz, combines mutation with search-based testing and thus systematically

maximizes code coverage [19]. More sophisticated fuzzers rely on *symbolic analysis* to automatically determine inputs that maximize coverage of control or data paths [7]. The KLEE tool [2] is a popular symbolic tester for C programs.

In our experiments, we use RADAMSA [8] which applies a number of mutation patterns to systematically widen the exploration space from a single input. Instead of RADAMSA, any other system-level test generator could also do the job.

2.4 Generating Unit Tests

The second important class of testing techniques works at the *unit level*, synthesizing calls of individual functions. These techniques separate in two branches: *random* and *symbolic*.

Random tools operate by generating random function calls, which are then executed. A typical representative of this class is the popular RANDOOP [12] tool. Random calls can be systematically refined towards a given goal: EvoSuite [5] uses a search-based approach to evolve generated call sequences towards maximizing code coverage.

Symbolic techniques symbolically solve *path conditions* to generate inputs that reach as much code as possible. PEX [16] fulfills a similar role for .NET programs, working on *parameterized unit tests* in which individual function parameters are treated symbolically.

Compared to the system level, test generation at the unit level is very efficient, as a function call takes less time than a system invocation or interaction; furthermore, exhaustive and symbolic techniques are easier to deploy due to the smaller scale. The downside is that generated function calls may lack realistic context, which makes exploration harder; and function failures may be false alarms because of violated implicit preconditions. Our parameterized unit tests supply a realistic context for unit-level testing; also, validating all unit-level failures at the system level means that we can recover from false alarms and any remaining failures are true failures.

2.5 Generating Parameterized Unit Tests

A number of related works has focused on obtaining parameterized unit tests by starting from existing or generated unit tests. Retrofitting of unit tests [15] is an approach where existing unit tests are converted to parameterized unit tests, by identifying inputs and converting them to parameters. The technique of Fraser and Zeller [6] starts from concrete inputs and results, using test generation and mutation to systematically generalize the pre- and postconditions of existing unit tests. The recently presented *AutoPUT* tool [18] generalizes over a set of related unit tests to extract common procedures and unique parameters to obtain parametrized unit tests. In contrast to all these works, our technique carves parameterized unit tests directly out of a given run, identifying those values as parameters that are present in system input.

3 Carving Unit Tests

3.1 How it Works

The concept of *carving* was introduced by Elbaum et al. as a general way to generate unit-tests out of system tests [4]. Conceptually, a system test can be represented as the execution path through the program under test that is taken when the program processes the test inputs. In carving, we select one function invocation, that is, a subpath which starts with the invocation of a function f , and ends when f returns. We call f the *function (or unit) under test*. We also record the values of all global variables, as well as all parameters for this invocation of f . We call the set C of recorded variable and parameter values the context. For the context C , it is necessary to record heap structures. Variables or function parameters may be pointers into the heap; the heap itself may again contain pointers, forming a (potentially large) structure of heap objects.

Just like regular unit tests, carved unit tests consists of three parts:

- **Setup.** The setup code populates all variables with the values we recorded in C . It reconstructs all heap structures recorded in C .
- **Test.** The test part invokes the function f , using the values that were constructed in the previous step as parameters.
- **Tear down.** The tear down step releases all resources that were acquired by the setup.

If the function f uses global variables and parameters only, the test is deterministic.

3.2 Example

Let us again take a look at the `bc` calculator. When `bc` parses the input `"1 + 2"`, each number is then stored in `bc`'s internal representation, as a `bc_num` structure. Then, `bc_add()` is invoked with two `bc_num`'s, one representing 1, and the other representing 2. This leads to the carved (non-parameterized) unit test as follows:

```
1 void test_bc_add() {
2 // set up the context
3 bc_num n1;
4 bc_int2num(&n1, 1);
5 bc_num n2;
6 bc_int2num(&n2, 2);
7
8 // call the function under test
9 bc_num result;
10 bc_add(n1, n2, &result, 0);
11 }
```

3.3 Implementation

In theory, building a carving tool is easy. Just take all variables and their values and write them out. The practice of carving, and especially the practice of carving for C programs is a challenge.

Our BASILISK prototype implements carving based on the *low-level virtual machine* (LLVM)[10]. LLVM provides a compiler, *clang*, which compiles C code to an inter-

mediate representation (LLVM IR). Clang also compiles LLVM IR to machine-code.

The LLVM IR intermediate representation is designed to be used by compiler optimizations, that is, static analysis of the code. It removes all the syntactic sugar that was added to C for the sake of human developers, so it is hard to read for humans, but better suited for automatic analysis. At the same time, it pertains type information, which makes an analysis simpler than analyzing machine code directly.

BASILISK works in two phases. It statically instruments the LLVM IR code, inserting probes which report all method invocations including the parameters, as well as all writes to global variables. During execution, those probes write the observed values at those points to a trace file.

For primitive types, like ints or floats, observed values can be written directly. However, there are more challenging situations for other data types, as listed below.

3.3.1 Pointers

In LLVM IR, as in C, a pointer is no more than a memory address. There is no information about the length of the memory segment that a pointer points to. But then, what should we dump out?

Dumping only the byte that the pointer points to would mean we do not see the remainder of the memory area. Keep in mind that pointers are often used as arrays, e.g. a `i8*` pointer, LLVM IR for a `char *` pointer, is often used to point to a string, an array of characters in memory. In this case, dumping just one byte would give only the first character of the string.

At the same time, we can not just dump arbitrary amounts of data. We would dump data that does not belong to this variable, and in some cases, accessing memory past the end of an allocated memory segment could even trigger a segmentation fault, and thereby crash the program under test.

To solve this problem, we maintain a map that records, for each pointer, the length of the memory segment it points to. Keep in mind that pointers can be calculated from other pointers. If a pointer `a` points to a memory segment of length 10, `b = a + 5` gives a pointer `b` which points to a memory segment of length 5, the second half of the segment pointed to by `a`. The map needs to be able to identify `b` as pointing into the memory segment at `a`, and calculating the remaining length¹. This allows us to find out how much memory should be dumped for a pointer.

In the following, we explain how to populate the map. There are two kinds of pointers:

Stack pointers point to local variables or function arguments. The memory areas for those objects are part of the stack, they are allocated (and deallocated) by the runtime when a function is called (or returns). As the compiler needs to be able to generate code to allocate a new stack frame, the compiler is capable of calculating the size of those memory areas. Our instrumentation gets the size at instrumentation time, the

¹Just in case you thought you could use an associative map for this lookup

concrete pointer is captured at runtime, and written to the map.

Heap pointers point to memory on the program heap. In C (and in LLVM IR, if the C standard library is used), memory is allocated with the `malloc()` function. `malloc()` receives the required length for a memory segment, and returns a memory segment of this length. Tracking all calls to `malloc()`, we can update the map on each call to `malloc`.

3.3.2 Strings

For strings, one might assume that we can rely on the fact that they are zero-terminated, that is, the last character will be zero. Unfortunately we can not. Zero-termination is a convention, not a rule. Programmers may also decide to accompany their `char*` variables with an integer variable, holding the length of the string. Also, we encountered several cases where a bitset was stored in a `char *`. In a bitset, there may be relevant data behind a zero byte. So assuming zero-termination in those cases means that we would loose data.

We thereby decided to handle `char *` in the same way as any other pointer type.

We saw that strings have uninitialized data at the end quite often, so if a `char *` was dumped, we would, in constructing the context, also try to construct the context with the assumption that the string was zero-terminated. In many cases, this removed uninitialized segments at the end of the string.

3.3.3 Structs and Unions

Struct types and union types are derived types. For a struct, that means that the struct consists of several values of different types, written to memory one after another. A union describes several options on how to interpret the same memory segment, e.g. 4-bytes of data may either be a single `int32` value, or a four-letter string.

We dump structs by handling each field recursively. Unions are compiled to structs and bitcasts (reinterpreting bytes in memory as some other type) in LLVM, so we don't need to think about how to handle them, LLVM already did.

3.3.4 Extensive lengths

For some subjects, we encountered large heap structures. Those might be large arrays, either as a LLVM array type or with a pointer that points to a lot of memory, or large connected structures, e.g. hash tables with lots of entries and long bucket lists. This is a problem, because the runtime overhead of dumping such large structures is extreme and a unit test which has to reconstruct such a structure will be large and slow. As our evaluation will show, slow unit tests are a problem for our approach.

We solved the problem by introducing a size limit for the heap structures we dump. If a structures is too large, we simply abort dumping it. This means that the carver has to deal with incomplete structures, and some unit test may not

build a complete context. However, as the lifting step filters false alarms, this does not pose a big problem.

3.3.5 External Resources

If the program under test had a file open when f was invoked, reconstructing variable values and heap structures will not be sufficient. Calls to `read()` or `write()` on the file will fail. Similar situations may occur with other resources, such as locked mutexes or open network connections.

A perfect solution would have to deeply interact with the operating system (and the system environment) to perfectly preserve and reconstruct states, which is out of our scope. We thereby just ignored the issue. It may lead to false alarms in the unit tests, but our lifting step will filter those.

3.3.6 Writing to Global Variables

We also want to dump the values of global variables. However, when a function is called we can not know which global variables it uses. Thereby we need to dump new values for global variables whenever a global variable is written to.

This is rather easy. LLVM IR uses the `store` instruction to write to memory. If `store` writes to a global variable, we dump the new value for this variable.

Unfortunately, this is not enough. If the global variable is a struct or array, individual members may be written. In this case, LLVM IR uses the `getelementptr` instruction to calculate the address of this member first. For an array of structs, or struct members in a struct, the result of `getelementptr` may be used in an address calculation again. If the global variable is a pointer to a pointer, the `load` instruction may be used to retrieve the underlying pointer.

We solved this as follows: When our instrumentation encounters a `store` instruction, it traces back the pointer operand, until it hits a value which is not a `getelementptr` and not a `load`. If this value is a global variable v , the `store` is considered to be a write to v and the value is dumped.

3.3.7 Program-Specific Extensions

Even with all the above heuristics, programmers may still choose internal data representations that make it hard for BASILISK to recognize data. As an example, `bc` uses char arrays (strings) to represent numbers as a string of digits; However, they do not use ASCII-encoded digits, which our heuristics for strings could handle, but the integer values of the characters directly.

To address this issue, we implemented special handling for the arbitrary-precision numeric types in `bc`. This special handling is just 76 lines of kotlin code. More careful engineering of our prototype could make this even easier. This special handling allowed us to use the `bc_int2num()` function from the subject to set up the environment.

Other than that, BASILISK was able to precisely identify, extract and reproduce almost all data structures for all of our subjects. Still, depending on how creative programmers

are as it comes to their own memory management, similar extensions may be required.

4 Parameterizing Carved Unit Tests

4.1 How it Works

We want to use our extracted unit tests for test generation. Therefore, they need to be *parameterized*—that is, there need to be parameters whose values can be set by the fuzzer.

In principle, the fuzzer may change any value in C . However, this bears the problem of generating *false positives*. A failure of a function under test is only relevant if it can be triggered with system-level inputs. A failure due to invalid unit-level inputs is irrelevant if those inputs will never be provided to the unit in a system-level invocation. We therefore restrict ourselves to *values that are derived directly from system-level input*.

System-level input consist of command-line arguments, input files, and other inputs. For ease of presentation, we will assume that all inputs are just one string S . In order to identify parameters that would be directly derived from S , we need the *data flow* from the system-level input S to the unit-level input C , such that we can identify the *origin* of each and every variable in C . To establish such a data flow, we could use *dynamic tainting*, tracking all input characters throughout the execution as well as their derived values, and eventually checking which of these reach the variables in C . However, we are interested only in *direct* flows that can be easily inverted—that is after a change in C , we want to be able to easily generate a system input that represents the change as well. Furthermore, dynamic tainting can be very slow, in particular considering that one may be interested only in a small set of functions and their arguments.

To match variables and their origins in the system input, we thus use a simple, yet efficient approximation. We traverse the variable values and heap structures in C . For each $v \in C$, we check whether v occurs in S :

- If v is a string, we check whether it is a substring of S .
- If v is numeric, an integer or a floating-point number, we check whether the decimal representation is a substring of S .

If we find a match, we mark v as a parameter. Instead of using the recorded value v , we now allow the fuzzer to insert a new value v' into the unit test, as a replacement for v .

4.2 Example

In our running example, BASILISK identifies the value 1 of the first method parameter as being related to characters 0 to 1 of the system-level input "`1 + 2`". The second argument 2 is related to characters 4 to 5 of the same input. Thereby it turns those two values into parameters. The fuzzer may now choose new values for those parameters.

4.3 Implementation

As described before, we check whether a method parameter or the value of a global variable is a substring of one of the system-level inputs. We handle S as a set of values per

input source, and we do the string comparisons for each input source individually.

For `char *` variables, we interpret them as strings and check them for substrings. This may not always be the correct approach, because as described above, `char *` variables are not always strings; yet, this brought sufficient results.

For integer variables and variables with a floating-point type, we used the usual decimal encoding to convert them to strings and again applied a substring comparison. This may limit the applicability to other subjects. If a subject accepts numeric input in, e.g. hexadecimal encoding, our prototype would not detect that and thereby miss opportunities to symbolize parameters.

Global variables frequently contain some system input. However, if this global variable is not used in the function under test, the unit test does not need to be parameterized. The values from the fuzzer would never be used. Thereby, we only consider reachable globals for parameterization.

We consider a global variable as *reachable* if there is a load from or a store to this variable in a reachable function. We consider a function as reachable if it is the function under test, called in a reachable function, if a function pointer to this function is created somewhere in a reachable function, or if a function pointer to the function exists in *C*.

We generate LLVM IR code which sets all global and local variables as recorded in *C*, and calls the function *f*.

5 Fuzzing Function Calls

5.1 How it Works

Once we have a parameterized unit test, we can use a fuzzer to choose new values for those parameters. A fuzzer basically provides random values.

Instead of simple random fuzzing, we could also use more sophisticated test generators; A tool like PEX [16], for instance (if it were available for C), could apply symbolic constraint solving to systematically explore paths in the carved parameterized unit test. Since the scope of the symbolic analysis would be constrained to only the carved parameterized test and the function under test, it would not suffer from the problems of scaling one would have when applying it at the system level. Hence, our approach effectively enables a fusion of system-level and unit-level test generation.

5.2 Example

For the running example, the fuzzer generates invocations such as:

```
1 test_bc_add(337747944, 352295539);
2 test_bc_add(535612873, 790525737);
3 // ... and more
```

5.3 Implementation

We wrote a simple unit-level fuzzer to generate new values for all parameters. We execute the unit tests with those new parameters and report all cases where the unit test fails or covers previously uncovered code.

For integer and double types, our fuzzer uses bitflips, random values and the values 0, `INT_MAX` and `INT_MIN`. For strings, our fuzzer uses bitflips, sequences of random bytes, sequences of random ASCII characters, all 0 strings and all `0xFF` strings. Also, we implemented a mutator which takes the original string and repeats sequences thereof.

Carving from a program run generates one (parameterized) unit test for each function that was called in the execution. The system maintains a list of all unit tests. When it is time to execute the fuzzer on a unit test, it orders the unit tests by the number of inputs that were already given to the function under test in other unit tests, and by the coverage all system tests known so far achieved on the function under test. The function with the lowest number of invocations, and among those with the same number of invocations the function with the lowest coverage so far will be parameterized and fuzzed first.

6 Lifting Failure-Inducing Values

6.1 How it Works

As stated in the introduction, invoking functions with generated values runs the risk of *false alarms*, with the values violating implicit preconditions for the usage of the functions. Typical examples include numerical values that are out of range, strings with invalid contents, and more. From the perspective of the function alone, one cannot distinguish whether the function failed because of a bug, or whether it failed because of an invalid input. Only in the context of the whole system can one decide whether a failure is a true failure, because a system is not supposed to fail with an internal error in the presence of invalid input.

A similar problem occurs with *coverage*. If we are interested in maximizing coverage, and we can generate a unit-level test that covers a new program structure, we want to validate this at the system level—because, again, the coverage achieved locally may not be easily feasible at the system level.

To address these problems, we thus *lift* input values from the function level back to the system level. Unit-level input values are selected for lifting, if they trigger a failure on unit-level, or reach new coverage on unit-level.

For each parameterized unit test, the fuzzer generates new values v' for all parameters v . Now, each v was a substring of the system-level input S . We recorded which interval in S v corresponds to. Thereby, we can derive a new system-level input S' by replacing v with v' in S .

BASILISK then invokes the program under test with the input S and observes the outcome of the execution. Ideally, the system test executes the same execution path as the unit test did. However, there are three possible outcomes

- If the invocation of the program under test produces the same failure as the unit test, we have a *true positive*, and generated a new, valid, and bug-revealing system test. We also provide the information which unit test and which unit-level input triggered the problem, which tells the developer where she should start debugging.

- If the invocation of the program under test covers the same code as the unit test, we have a *true positive*. We generated a new, valid system test which achieves additional coverage.
- If the system test does *not* fail and does *not* achieve the new coverage, as predicted by the unit test, we have a false positive. False positives occur because in the system context, the unit-level values do not reach the function unchanged, or not at all. Also, the failure may not occur because the context C is not completely reconstructed, or because the substring relation between v and S was circumstantial. Whatever the reason, there is no need for the human software developer to look into the failure, as it cannot be reproduced at the system level.

For all reported failures, the developer gets a system test as well as a unit test that both faithfully reproduce the failure (or achieve the new coverage). In case of failure, the developer can use the system test to assess which external circumstances lead to the error, and also to demonstrate that the failure is real; and she can use the unit test to debug the program in context, without having to step through the entire system test.

6.2 Example

In our running example, the first parameter was related to characters 0 to 1 of "1 + 2", and the second parameter was related to characters 4 to 5. Assuming that the invocation

```
1 test_bc_add(337747944, 352295539);
```

reveals a bug or at least provides additional coverage, we generate the system-level input 337747944 + 352295539.

7 Evaluation

In our evaluation, we attempt to answer the following research questions:

1. How do the *unit tests* generated by BASILISK compare against system tests from RADAMSA?
2. How do the *system tests* generated by BASILISK compare against system tests from RADAMSA?
3. Which *time savings* are possible if one wants to focus on a subset of functions?

7.1 Subjects

We applied our prototype implementation on seven subjects.

Four of the subjects are part of *GNU coreutils*, a collection of standard command line tools which is used, e.g. on Linux. The `cut` program reads text from a file and outputs substrings, as specified by the user. The `paste` program, also part of *GNU coreutils*, can be used to merge lines from different text files. The `tac` command reads a file and outputs it in reversed order. `b2sum` computes a message digest, some kind of checksum, from an input file.

Table 1: Evaluation subjects

| Subject | LoC | Functions | |
|---------|------|-----------|--------------------------------|
| b2sum | 1228 | 115 | checksum calculation |
| paste | 662 | 79 | text processor |
| tac | 987 | 111 | text processor |
| bc | 3456 | 151 | arbitrary-precision calculator |
| dc | 1997 | 136 | arbitrary-precision calculator |
| cut | 1346 | 127 | text processor |
| sed | 2715 | 215 | text processor |

`sed` is a stream editor that applies a list of user-specified commands on its input and outputs the resulting text.

The remaining two subjects are the `bc` and `dc` programs. Both of them are programming languages with arbitrary-precision floating-point arithmetic. `bc` uses a C-like syntax, meaning that the input "1 + 2" prints 3, as one would expect. `bc` also allows for variables and functions. `dc` uses reverse polish notation, where the operator follows its operands: "1 2 +" yields the output 3. `dc` has registers, which can be used as variables. `bc` and `dc` share their arithmetic code.

The subjects are listed in Table 1. We also report the lines of code for each subject. Especially for the programs from *GNU coreutils*, the source code repositories do contain more code. We counted only lines of code that are in functions that are reachable from the main method of the respective program. We used the reachability analysis we described in Section 4.3.

7.2 Running BASILISK and RADAMSA

Our BASILISK prototype starts with a set of *seed tests*. First of all, it uses RADAMSA to generate 10 additional system tests per seed test. Then, it carves unit tests from all system tests. Afterwards a unit test is selected as described in Section 5.3, and the process of parameterizing, unit-level fuzzing and lifting is applied to this test. Once this is done, the next unit test will be selected and processed. Generated system tests are executed immediately, and the coverage they generate may already be used in the next selection step.

In our experiments, we used a time limit. Our prototype runs each system test directly after creating it, so the output is coverage information. RADAMSA only generates test inputs. That means that comparing directly is not fair. While, for our tool, the time limit includes test executions, RADAMSA needs additional time to execute the generated system tests.

In order to mitigate this difference, we had RADAMSA generate system tests in batches of 10 tests each and executed each batch before generating the next one, until the time limit was exceeded. This means that for RADAMSA, as for BASILISK, system test execution time is included in the time limit.

Table 2: Execution times for system tests vs. unit tests

| Subject | Unit Tests | System Tests | Speed up |
|---------|------------|--------------|----------|
| b2sum | 22.77ms | 1172.94ms | 51.50× |
| paste | 33.74ms | 1836.20ms | 54.42× |
| tac | 19.29ms | 3520.12ms | 182.44× |
| bc | 87.37ms | 1023.46ms | 11.71× |
| dc | 122.95ms | 1708.41ms | 13.90× |
| cut | 41.38ms | 264.85ms | 6.40× |
| sed | 6.82ms | 729.00ms | 106.85× |
| total | 47.76ms | 1465.00ms | 30.76× |

7.3 Unit Tests vs. System Tests

In our first experiment, we compared the unit tests generated by BASILISK with system tests produced by RADAMSA. Experiments were ran on an Intel i7-2600 processor at 3.40GHz with 16 GB RAM on Linux. We measured real time elapsed for the programs instrumented to obtain coverage information; this instrumentation is responsible for the relatively long executions.² The time limit was 15 minutes for both tools.

We used one hand-written seed test, which was identical for both tools. In designing the seed tests, we used input examples that we found in online tutorials or the bug tracker of the respective programs. We ran each experiment 5 times with different seeds; the results are means over all five runs.

Table 2 lists the mean execution time for the BASILISK unit tests as well as the RADAMSA system tests. We see that, as expected, a single execution of a carved parameterized unit test with arguments runs much faster than a system test, up to a factor of 180 and with a mean factor of 30.

Carved unit tests execute much faster than system tests.

The finding that unit tests execute much faster confirms the experiments of Elbaum et al., who reported that their carved tests “reduce average test suite execution time to a tenth of our best system selection technique” [4]. In contrast to Elbaum et al., though, we are not limited to invocations seen during system testing, but can (and do!) generate additional ones.

7.4 Overall Coverage

Let us now see whether the unit test speedup brings benefits during testing. To this end, we identify those unit tests that result in an increase in branch coverage. We lift those tests to become system tests. Table 3 shows that although BASILISK runs hundreds of thousands of unit tests, only a small fraction of these results in new branch coverage and yields new system tests. While some of the unit tests fail,

²We implemented our own coverage tool, as for our experiments, we would require the ability (a) to fully trace executions and (b) to maintain that trace (and the coverage) even in the presence of failures; the tool is thus optimized for reliability rather than performance. An industrial strength implementation for coverage only would require only an overhead proportional to the code size, as it would replace instrumented with uninstrumented blocks once covered.

Table 3: Lifted unit tests

| Subject | #Unit Tests | #Lifted Tests | % lifted |
|---------|-------------|---------------|----------|
| b2sum | 545110.4 | 329.8 | 0.06% |
| paste | 219379.6 | 273.0 | 0.12% |
| tac | 872961.2 | 79.2 | 0.01% |
| bc | 8181.4 | 159.1 | 1.94% |
| dc | 396664.8 | 125.6 | 0.03% |
| cut | 909140.4 | 383.0 | 0.04% |
| sed | 25095.8 | 167.7 | 0.67% |
| total | 2976533.6 | 1517.4 | 0.05% |

none of these failures still occur after lifting the generated arguments back to system tests.

Lifting unit test failures to system tests is effective in preventing false alarms.

If we measure the branch coverage of the system tests thus lifted, we can directly compare the coverage of BASILISK and RADAMSA. (At this point, both tools have spent the same test budget of 15 minutes, which for BASILISK includes carving, parameterization, and lifting.) Table 4 contrasts the number of tests produced and branch coverage achieved.

It is worthwhile to note that BASILISK achieves its coverage through *fewer* tests than RADAMSA. Such a lower number of system tests is helpful, as it (re-)executes faster. This is because BASILISK only generates system tests where the originating unit tests have achieved *new* coverage (and where the coverage gain is confirmed for the system test after lifting), while RADAMSA uses no such feedback from the program.

Comparing the first column in Table 4 and the second column in Table 3, it can be seen that the number of tests that were generated in BASILISK’s fuzzing stage, the difference between the columns, is small. So BASILISK mostly relies on lifting. `cut` is an exception here. For `cut`, many unit tests were selected for lifting (thus counted in table 3), but could not be lifted (the coverage gain was not confirmed), so the number of system tests is lower than the number of lifting attempts.

We see that on five out of seven subjects, BASILISK reaches a higher coverage than RADAMSA. The two subjects where BASILISK does not reach a higher coverage are `bc` and `dc`; for `cut`, the gain is small as well. Looking at the rightmost column of Table 2, these subjects have the smallest speed ups of unit tests in comparison to system tests. The performance gained by executing a unit rather than the entire program is offset by the analysis time of carving and parameterization. On the other hand, the subjects with a high speedup of unit tests vs. system tests can very much profit from the carved parameterized unit tests.

Table 4: Branch coverage achieved by BASILISK and RADAMSA

| Subject | #System Tests | | Coverage | |
|---------|---------------|---------|----------|---------|
| | BASILISK | RADAMSA | BASILISK | RADAMSA |
| b2sum | 358.0 | 629.0 | 37.93% | 19.49% |
| paste | 280.0 | 346.3 | 33.33% | 31.08% |
| tac | 89.6 | 212.6 | 34.66% | 30.71% |
| bc | 169.0 | 577.2 | 26.47% | 28.46% |
| dc | 135.4 | 434.6 | 18.39% | 41.06% |
| cut | 339.2 | 3117.1 | 21.00% | 20.50% |
| sed | 175.7 | 1058.3 | 21.19% | 15.73% |

Table 5: Number of functions reached by BASILISK and RADAMSA

| Subject | BASILISK | RADAMSA |
|---------|----------|---------|
| b2sum | 42 | 21 |
| paste | 21 | 22 |
| tac | 23 | 24 |
| bc | 69 | 71 |
| dc | 49 | 73 |
| cut | 50 | 49 |
| sed | 81 | 69 |

If the unit tests are sufficiently faster than the system tests, parameterized carved unit tests yield a higher coverage.

To paint a complete picture, let us also take a look at the functions invoked. Table 5 compares BASILISK and RADAMSA in terms of covered functions.

The biggest difference between BASILISK and RADAMSA is in the `dc` subject. Investigation of the generated tests shows that RADAMSA manages to generate new operators, e.g. "`1 2 + p`" may be mutated to "`1 2 * p`". BASILISK only mutated the numbers themselves. Thereby RADAMSA discovered more functions, namely those for multiplication and other operators, while BASILISK found more paths in the already discovered functions. Of course, whether one or another strategy is more effective depends on the (typically unknown) distribution of bugs in practice.

7.5 Focusing on Single Functions

While our previous experiment focused on running *all* functions, in practice, we typically want to focus on *specific* functions. For instance, one may wish to focus testing on recently changed functions, functions that have a history of failures, functions that are critical, or other reasons that demand extensive testing.

System test generators like RADAMSA give the tester no means to focus test generation on specific functions. With parameterized unit tests, as carved by BASILISK, this is easy: We just execute those unit tests that test the function of interest.

If we want to test a *single* function, Table 2 already gives

us an indication of the speedup we can expect. By executing only those unit tests related to a single function, we can expect a significant speed-up; in our experiments, this is the factor 30 already mentioned. While this speedup occurs only after carving and parameterization, a high number of tests will amortize the effort for these steps.

In our experiments, after carving a parameterized unit test for a function, one can test the function on average 30 times faster than with the original system test.

This factor 30 we found in our experiments has to be taken with a grain of salt, as it will very much depend on the speed differences of individual functions vs. system execution as a whole. If the function to be tested encompasses most of the functionality of the program (e.g., the `main()` function in C), carving a parameterized unit test will not yield significant time advantages. On the other hand, if a program takes multiple seconds to start up or to process inputs before it can call a function (think of starting a Web browser or an office program), the time savings factor can easily reach several orders of magnitude. Such savings accumulate as the carved parameterized unit tests can be reused again and again.

7.6 Focusing on Sets of Functions

Let us now assume we have not one, but a *set* of “focus functions” we are especially interested in—a set that may also be reached by chance through unfocused testing. Table 6 shows the focus functions we randomly chose per subject.

Again, we ran BASILISK for 15 minutes per subject.³ In the first “unfocused” setting, we ran BASILISK as described above, executing all unit tests without further discrimination. In the second “focused” setting, we had BASILISK carve and explore parameterized unit tests *only* for the focus functions. Table 7 shows the time spent in the focus functions in both settings.

We see that having BASILISK focus on a small set of functions executes these focus functions much more often than the unfocused version. The time BASILISK spends on these functions increases by a factor of at least six. In other words, we can test a function six times as much than with an unfocused setting, or we can test six times as quickly.

³All reported values are means over 5 runs.

Table 6: Focus functions for each subject

| Subject | Functions |
|---------|---|
| b2sum | blake2b_init(), blake2b_init_param(), blake2b_update(), blake2b_final() |
| paste | xstrdup(), xmemdup(), quotearg_n_style_colon() |
| tac | quotearg_style() |
| bc | lookup(), bc_sub(), bc_multiply(), bc_out_num(), bc_add() |
| dc | bc_out_num(), dc_add(), dc_mul(), dc_multiply(), dc_binop(), bc_add() |
| cut | xstrdup(), hash_initialize(), xstrndup(), quote(), quote_n(), c_tolower() |
| sed | compile_string(), normalize_text(), compile_regex() |

Table 7: Time in milliseconds spent in focus functions

| Subject | Unfocused | Focused |
|---------|-----------|---------|
| b2sum | 128.76 | 661.00 |
| paste | 131.26 | 785.50 |
| tac | 11.75 | 640.16 |
| bc | 112.41 | 647.34 |
| dc | 151.33 | 771.94 |
| cut | 96.28 | 820.48 |
| sed | 98.26 | 842.20 |

With carved parameterized unit tests, one can focus on a subset of functions to be tested, yielding significant time savings.

The differences are even more dramatic when comparing the *number of invocations*. In Table 8, we see the number of invocations for the focus functions per configuration, now also including RADAMSA. We see that focusing on a subset of functions can yield savings of up to a factor of 18,000 (cut). Again, this factor depends on the average running time of a unit; for dc, our focus functions do not yield savings over system testing, as they take too long to carve and run.

With carved parameterized unit tests,

Table 8: Number of invocations of focus functions

| Subject | Unfocused | Focused | RADAMSA |
|---------|-----------|---------|---------|
| b2sum | 140468 | 2167102 | 1114 |
| paste | 24723 | 248898 | 1039 |
| tac | 60 | 1938 | 172 |
| bc | 498 | 96041 | 11131 |
| dc | 19551 | 19388 | 259167 |
| cut | 270514 | 2973383 | 163 |
| sed | 562 | 69327 | 1119 |

focus functions can be executed much more often.

7.7 Discussion Summary

In the end, speed gains through carved parameterized unit tests will depend on multiple factors: Even if we have no focus set at all and include the effort for carving, we may still see gains in coverage over time (and conversely, less time for the same coverage), as discussed in Section 7.4. On the other hand, the smaller the focus set, and the quicker the functions execute, the higher the gains will be—up to the dramatic speedups shown in Table 2, discussed in Section 7.5.

7.8 Limitations and Threats to Validity

Like any empirical study, our evaluation is subject to threats to validity, many of which are induced by limitations of our approach. The most important threats and limitations are listed below.

- Threats to **external validity** concern our ability to generalize the results of our study. We cannot claim that the results of our experimental evaluation are generalizable. A huge concern is that we establish mappings from inputs to unit-level values via (sub-)string equality. This approach will, most likely, fail for programs that do not process text, e.g. image analysis software. Another concern is that, if individual units need a long time to execute, the gains through carved parameterized unit tests will be small compared to system testing; if individual units are too large, the overhead of analysis, carving, and parameterization may not be offset through faster unit test execution. We counter the threat in making our research infrastructure available, allowing for replication and extension.

As we detail in Section 3.3, carving is inherently limited in its ability to reconstruct a given context, especially when including external resources. A carved parameterized unit test involving external resources runs the risk of not being able to execute a function at all, and such failures will not be reported as they will not be reproducible in the lifted system test. This can be partially amended through deeper interaction with runtime and operating system; but such extension is out of scope for the present paper.

- Threats to **internal validity** concern our ability to draw conclusions about the connections between our independent and dependent variables. RADAMSA is a random-driven test generator, yielding different results each time. Since in our evaluation, BASILISK starts with the tests coming from RADAMSA, and as it chooses random values when fuzzing individual functions, we have a second influence of randomness. Furthermore, the choice of the seed test(s) has a huge influence on the achieved coverage. A good seed test leads to more initial coverage, and thereby more carving opportunities, which benefits BASILISK. On the other hand,

a good seed test also gives RADAMSA more opportunities to mutate system-level inputs; these effects may level each other out. We counter both threats by running the experiments multiple times with different seeds, reporting mean times across all runs.

- Threats to **construct validity** concern the adequacy of our measures for capturing dependent variables. To evaluate the quality of our tests, we use standard measures such as code coverage and execution time, which are well established in the literature.

8 Conclusion and Future Work

Carved parameterized unit tests bring together the best of system-level and unit-level testing. Like unit tests, they can be quickly executed and focused on a small set of locations; from system tests, they obtain valid and realistic contexts in which test generation takes place; and when a unit fails, the failure can be lifted to the system level and validated there, either suppressing a false alarm or yielding a failing system test. As our evaluation shows, the greatest potential of carved parameterized tests is in the speedup they provide: Focusing on a small set of functions of interest allows to speed up testing by orders of magnitude when compared to system-level test generation. On a more conceptual level, carved parameterized unit tests create a bridge between system-level test generators and unit-level test generators, which can be arbitrarily combined; it thus paves the way towards new and exciting combinations of the best of two worlds.

Although our present approach came to be by exploring and refining several alternatives, it is by no means perfect or complete. Our task list for the future includes the following extensions:

- **Dynamic tainting.** Our method for associating unit-level values with system inputs works well for all our subjects; however, we would like to have a method that also works when the input undergoes a number of computation steps. To this end, we want to apply *dynamic tainting* to follow individual characters of a system input through an execution, along with their derived values; this would effectively allow to associate any function argument with the input subset that influenced it. The downside is that dynamic tainting induces a massive overhead (which may be better spent on test generation), and that the transformation steps from system input to function argument may not be easily reversible, preventing the final lifting step.
- **Advanced unit testing.** Rather than simply feeding random values into functions, we would like to apply symbolic or search-based test generators at the unit level such that we can cover functions in a guided fashion. We are currently experimenting with the KLEE tool [2] for this very purpose; first results show that its generated inputs provide higher coverage than our randomly produced inputs, but this advantage is offset by the time it takes for analysis. Another tool on

our list is *libfuzzer* [1]; this tool provides a sequence of random bytes, which it evolves according to the coverage achieved.

- **Advanced system testing.** Instead of RADAMSA, we plan to experiment with alternate system-level test generation tools to obtain a wider range of function calls and arguments. Grammar-based testing, as in the LANGFUZZ test generator [9], would allow to quickly cover input features and thus functions and values; furthermore, we could directly associate grammar elements such as strings and numbers with function arguments.
- **Better carving.** Carving is a bag of hurt, plain and simple. We are happy we made it far enough to get the experiments running, and we think we do have a nice and clean carving infrastructure at this point. There is definitely room for improvement, and we may work on this at some point; but if, in the meantime, you plan to implement a carving technique for C, please contact us: We are happy to share our carving infrastructure such that you can save a year or so of coding and another year of debugging.

To allow easy reproduction and validation of our work, we have created a replication package for download. The package includes all raw experimental data, as well as BASILISK itself:

<https://tinyurl.com/basilisk-icse19>

References

- [1] libFuzzer: a library for coverage-guided fuzz testing. <https://lvm.org/docs/LibFuzzer.html>.
- [2] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224, 2008.
- [3] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering, ICSE '05*, pages 342–351, New York, NY, USA, 2005. ACM.
- [4] Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Matthew Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Transactions on Software Engineering*, 35(1):29–45, 2009.
- [5] Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. *2011 11th International Conference on Quality Software*, pages 31–40, 2011.
- [6] Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*,

- ISSTA '11, pages 364–374, New York, NY, USA, 2011. ACM.
- [7] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.
- [8] Aki Helin. Radamsa. <https://gitlab.com/akihe/radamsa>.
- [9] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium, Security'12*, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.
- [10] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
- [11] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, December 1990.
- [12] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, volume 2 of *OOPSLA '07*, pages 815–816, New York, NY, USA, 2007. ACM.
- [13] Marina Polishchuk, Ben Liblit, and Chloë W. Schulze. Dynamic heap type inference for program understanding and debugging. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 39–46, New York, NY, USA, 2007. ACM.
- [14] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, Sep 1972.
- [15] Suresh Thummalapenta, Madhuri R. Marri, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Retrofitting unit tests for parameterized unit testing. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, FASE'11/ETAPS'11, pages 294–309, Berlin, Heidelberg, 2011. Springer-Verlag.
- [16] Nikolai Tillmann and Jonathan de Halleux. Pex–white box test generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, pages 134–153, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [17] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 253–262, New York, NY, USA, 2005. ACM.
- [18] Keita Tsukamoto, Yuta Maezawa, and Shinichi Honiden. AutoPUT: An automated technique for retrofitting closed unit tests into parameterized unit tests. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1944–1951, New York, NY, USA, 2018. ACM.
- [19] Michał Zalewski. american fuzzy lop. <http://lcamtuf.coredump.cx/afl/>.
- [20] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM.
- [21] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In *Revised Lectures on Software Visualization, International Seminar*, pages 191–204, London, UK, UK, 2002. Springer-Verlag.