OCCAM-V2

Combining Static and Dynamic Analysis for Effective and Efficient Whole-program Specialization

LEVERAGING SCALABLE POINTER ANALYSIS, VALUE ANALYSIS, AND DYNAMIC ANALYSIS

JORGE A. NAVAS AND ASHISH GEHANI

Software has evolved to support diverse sets of features. Agile software engineering practices, such as code designed for reusability, introduce redundant code. In a common motif, entire libraries are linked where only a small number of functions are needed. The accumulation of extraneous code can negatively impact application users who need to access only a subset of these features.

At one end of the spectrum of concerns, embedded systems are typically provisioned with a limited amount of memory. The presence of code that unnecessarily takes space can increase cost and adversely affect performance. At the other end, cloud computing platforms provisioned with bloated code may suffer from a commensurately increased internal attack surface, through mechanisms such as return-oriented,[18] jump-oriented,[3] call-oriented,[17] and data-oriented[10] programming.

As the number of configuration parameters increases, the space of application traces grows exponentially. This makes comprehensive testing proportionately

more expensive, motivating the need for *whole-program specialization,* which can focus program analysis on the specific variant that will be deployed.

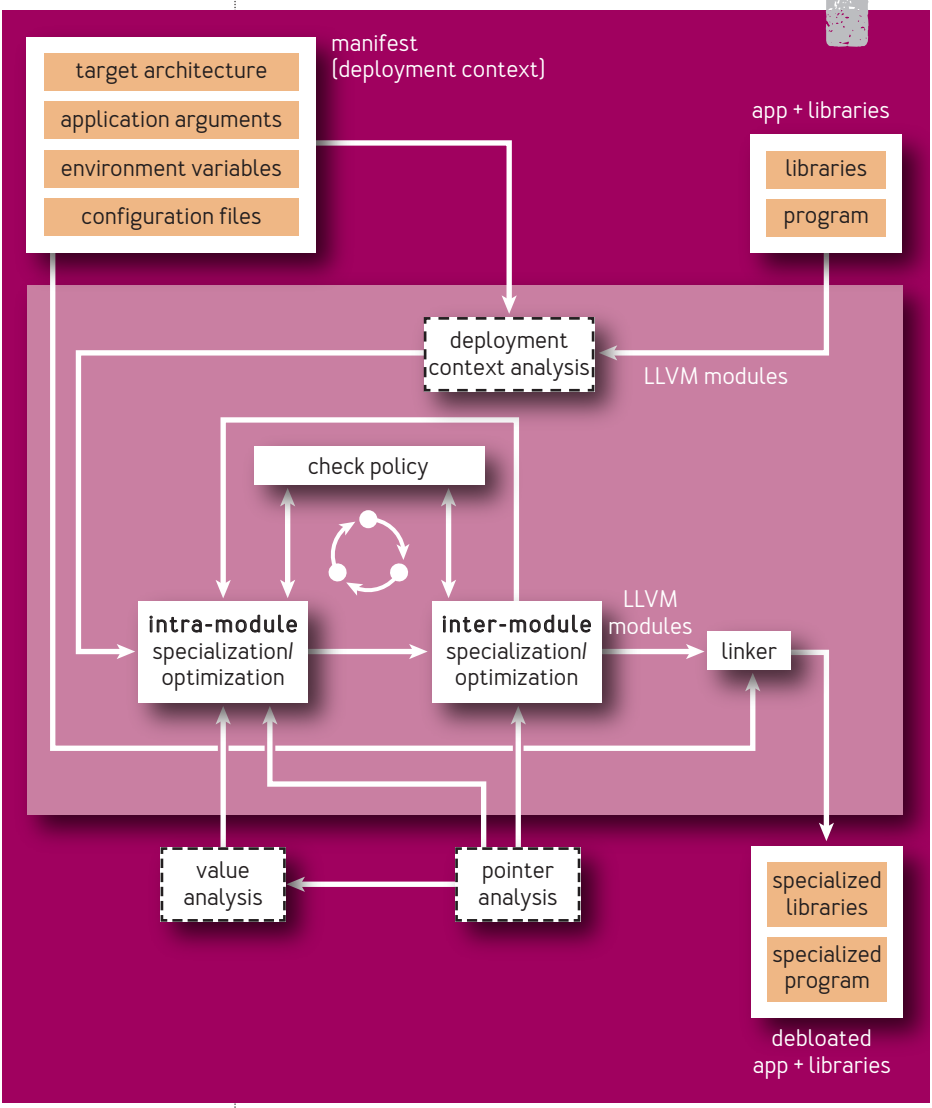
Since a range of source languages can be compiled to LLVM's[12] intermediate representation, several tools have been developed to specialize LLVM bitcode. These include OCCAM,[13] LLPE,[20] and Trimmer.[19] All of them, however, use only compiler transformations for the task. In contrast, our tool, OCCAM-v2, incorporates deeper static analysis using an abstract interpretation framework, as well as dynamic analysis. The combination provides noticeably better results.

OCCAM-v2[16] has been developed as an extension of our earlier tool, OCCAM.[13] Functionality to automate complex bitcode builds was factored out into a separate tool, `gllvm`.[8] This article begins by providing relevant background about OCCAM, then describes its static-analysis extensions, which provide the same soundness guarantees as OCCAM. Next it explains optional aggressive specialization using dynamic analysis, and finally reports the results of evaluating the tool on a suite of applications.

BACKGROUND

Automated software winnowing[13] applies partial evaluation to prune large applications and their dependencies, as illustrated in figure 1. (The components enclosed in dashed boxes are new in OCCAM-v2.) In particular, OCCAM allows static information to be passed to the entry point of a target program using a specialization manifest that contains specific environment details. The tool operates on LLVM bitcode.

FIGURE 1: **OVERVIEW OF OCCAM'S WORKFLOW**



The functions and call sites in each module are calculated and stored in an external interface definition. Using this, OCCAM then searches for cross-module calls with concrete arguments. If any are found, a policy is consulted to determine whether to construct a specialized version of the function. When this occurs, the code for the new function is constructed and added to the relevant LLVM bitcode module; the corresponding interface is updated to reflect the availability of the new function; other call sites with the same signature are updated to use the new specialized version of the function.

OCCAM starts at the program entry point and uses any static information that may be provided in the accompanying manifest for an application. The modules containing functions identified in the call sites of the `main()` module are then operated upon. This process continues until all reachable modules have been identified. At each step, LLVM's internal optimization passes are invoked. In particular, the global optimizer is leveraged for constant folding and propagation, and dead-code elimination prunes unused branches and unreachable code. The latter steps effect OCCAM's intra-modular specialization.

Since this process may uncover new opportunities for specialization, both within a single module as well as across multiple modules, the entire sequence is repeated until a global fixed point is reached. In this manner, OCCAM is able to scale whole-program specialization. In a final step, OCCAM invokes LLVM's compiler to transform the bitcode into native objects and the linker to produce a specialized native application.

EXTENDED STATIC ANALYSIS

OCCAM-v2's static analysis improvements derive from its incorporation of a custom pointer analysis and a value analysis. Both are based on abstract interpretation.[5]

Precise call-graph construction

LLVM's `PoolAlloc` DSA (data structure analysis) can be used to resolve indirect function calls, as well as other pointers. OCCAM-v2 uses the call graph computed by an improved variant of DSA: the open-source type-sensitive pointer analysis, SeaDSA.[11]

Improved cross-module interfaces

To support scaling to large codebases, OCCAM operates on one module at a time (except for post-fixed-point processing, including final linking). To facilitate whole-program analysis, however, changes being made to a module must be cognizant of interactions with other code. This is realized through the construction of an interface for each module, which can be used in the process of updating the others. In particular, the interface provides a description of the external symbols and calls that are used within a module. In the past, interface generation has used the call graph computed by LLVM.

OCCAM-v2 uses the call graph computed by SeaDSA. This call graph is more precise than the one from LLVM. This in turn improves the specificity of module interfaces, which better optimizes cross-module specialization. When the interface of a module is computed, it initially includes all call sites that have noninternal linkage (and can thus be called from another module). Further, all call sites in

functions that can be reached by calls from other modules are identified. In cases where such calls are indirect, the target can be any function (in the absence of further information). Using the resolved call graph, OCCAM-v2 reduces the set of possibilities.

Handling function pointer arguments

When a call is made to a function that is external to a module, its signature is reported in the interface of the module that contains the caller. When a call site contains a constant argument, the invocation can be specialized in principle. If an argument is a function pointer, OCCAM was previously limited to the case where a null value was used. This does not suffice for handling the case where a function is passed by reference as an argument. OCCAM-v2 supports specialization of external calls with function pointer parameters. If the function takes one or more constant arguments in the context of the callee, the function can now also be specialized. This is of particular note since it covers common programming patterns, such as the use of callback functions.

Signature-based candidate call elimination

The use of DSA results in an over-approximation of the actual program call graph since DSA ignores the types of a function call's arguments. Consequently, functions with signatures that do not match the call site's signature may be treated incorrectly as valid candidates. To address this, SeaDSA's pointer analysis was leveraged to replace indirect calls with a switch statement where each case corresponds to one of the candidate addresses. After this,

the resulting call graph will not have indirect calls. In some cases, however, the function signatures at the call site and candidate addresses may not match. The analysis correctly filters out such cases based on parameter types.

Class-hierarchy-analysis-based candidate call elimination

When an indirect call is resolved to a set of candidate addresses, the resulting call graph is less precise than that of the original program. The precision of subsequently performed static analyses can be improved by reducing the set of candidates. If an instance of LLVM bitcode originates from a program that had been written in C++, the CHG (class hierarchy graph) can be leveraged to improve the analysis precision.

OCCAM-v2 builds a CHG where nodes are C++ classes and an edge between classes c1 and c2 denotes that c1 is a subclass of c2. The CHG is used to resolve indirect calls that originate from C++ virtual functions. (Our pointer analysis supports C++ virtual calls. CHGs, however, can often refine the information inferred by SeaDSA through analysis of the virtual tables that can be extracted from LLVM bitcode.)

Given a call site that originates from a virtual call in a C++ program, the following steps are performed:

1. Identify the class C associated with the object containing the called method.
2. Compute all directly and indirectly derived subclasses of C. (Once the CHG is available, this can be computed at low cost.)
3. Use the virtual table of each subclass to collect all

methods with a type signature that matches the called method mentioned in step 1. (The virtual table of a class can be constructed by inspecting the bitcode.)

At this point, the set of all possible callees for a given indirect call that originated from a virtual call are known. This allows an indirect call to be resolved using this set of callees.

Alias-analysis-based optimization

The LLVM framework provides an `AliasAnalysis` infrastructure. Clients of this API can query whether two LLVM values may alias. This is useful to transformations that can leverage the knowledge that two values cannot point to the same location when optimizing target code. Implementations must model memory and provide support to answer these queries. An `AliasAnalysis` API implementation was developed that uses SeaDSA. This allows the entire suite of LLVM passes used by OCCAM-v2 to leverage the increased precision of the *points-to* information. The improved alias analysis enables other relevant optimizations, such as dead-store elimination, which helps identify variables that are otherwise static.

Use of value analysis

OCCAM uses dead-code elimination when winnowing software. The core of the pruning algorithm relies on constructing a mapping from variables to constants. This means that the variable will have the identified constant value for all possible program executions. To create further specialization opportunities, OCCAM-v2 constructs mappings from variables to intervals. This

allows reasoning about branches that depend on variables for which an interval can be identified.

Abstract interpretation[5] is a common technique to reason formally about programs. It is used as the mathematical foundation in the design of static analyses. One classical use of the technique is to infer for each program variable the possible values that the variables can take. These possible values can be approximated in different ways (called *abstract domains*): intervals to represent the smallest and largest values; differences between two variables;[14] octagon constraints;[15] and linear inequalities between variables.[6]

Although powerful, this kind of analysis is not available in LLVM. One possible reason is that most of the existing implementations assume mathematical integers, ignoring the fact that in LLVM integers must be modeled using the machine representation of arithmetic. Adapting this kind of analysis for machine arithmetic is challenging because it is hard to find a good balance between precision and efficiency.

Inferring invariants

The C++ library Crab[7] supports static analysis of programs using abstract interpretation. It is structured as a library so tools can use its functionality programmatically. Crab operates on a language-independent form of a target program's CFG (control flow graph). The Clam[4] frontend translates LLVM bitcode into Crab's representation. After this, Crab's analyses are used to infer invariants in the target program. Constant propagation and dead-code elimination is then effected (using the same LLVM

optimizer passes that were used previously). The presence of the identified invariants allows these passes to make stronger assumptions. In some cases, this enables more optimization to be performed.

Field-sensitive constant propagation

OCCAM's intra-module specialization relies on LLVM's SCCP (sparse conditional constant propagation). This pass is limited in the analysis it can perform since it is designed for efficiency and use in a compiler's routine workflow. In particular, if a constant is stored in a C array or the field of a struct, SCCP cannot identify and use this static data.

Clam relies on the field-sensitive reasoning of SeaDSA to maintain an abstract representation of memory. SeaDSA partitions memory into a finite number of regions, each of which is mapped to a logical array. In this form, a region is modeled by a logical expression that describes the locations that are affected by memory-access operations. This allows efficient reasoning about memory with SeaDSA handling the alias analysis.

When the LLVM bitcode is translated into Crab's CFG representation, memory stores are modeled as weak updates (where the region affected is known but the exact position is unknown). Clam defines an abstract domain for weak updates. Crab uses it for making inferences about the CFG. While reasoning in this domain is efficient, it is imprecise. To improve precision, particularly in the case of fields of a struct, a new abstract domain was developed.[9] It provides a hybrid abstraction, where a memory store can instead be modeled as a strong update (which replaces the old value with a new one) in specific cases.

Optimization of intrafunction analysis

Knowledge of invariants facilitates compiler optimizations that improves OCCAM-v2's ability to specialize and eliminate code fragments in a target application and its dependencies. A context-sensitive interprocedural analysis was developed for this, providing increased precision over the baseline. To improve performance, support was added for memoization of intrafunction analysis. This can lead to memory exhaustion in practice, however. This is addressed by limiting the memoization to loop headers.

INCORPORATING DYNAMIC ANALYSIS

As an online partial evaluator, OCCAM relies on known concrete values for code simplification and elimination. A significant source of such information is the set of arguments specified by the user. In practice, these are processed by `getopt()` or similar functionality close to the program entry point. Such code typically performs complex string operations in a loop. Additionally, the loop bound may not be static. This combination poses a challenge for static-analysis techniques.

OCCAM-v2's DCA (deployment context analysis) splits specialization into two phases. Conceptually, the execution trace of a target program is decomposed into a single prefix that is computed using (conditional) dynamic analysis in the first phase, and a set of suffixes that represent any remaining portion of the program that was not executed in the first phase. The dynamic analysis starts at the program entry point and proceeds as long as the key hybridization condition holds. Specifically, any branch encountered must depend on only known values. The

memory snapshot is the state of the program at the end of the prefix's execution. It is used to simplify the suffixes using static analysis.

This functionality is implemented by modifying LLVM's interpreter `lli`. The new pass differs from `lli` in three respects:

- The value of a virtual register or a memory location that has been allocated on the stack or heap may not be known, even during execution. For example, the program may use the value of the first argument (referred to as `argv[1]` in C), but at runtime the user may not have provided any arguments when invoking the program. In such cases, the LLVM interpreter will abort execution. In contrast, the new pass will move to the next bitcode instruction and continue interpreting.
- When a register or memory location is encountered for which the value is unknown, the new pass will keep track of this fact. When a subsequent instruction depends on a value that is unknown, its result will be tracked as unknown as well. In this manner, the new functionality provides a lightweight binding time analysis that propagates forward the taint from dynamic values.
- In the special case that the bitcode instruction being interpreted is a condition that depends on an unknown value, the execution will be halted. This allows a program to continue to run even in the presence of branches, as long as they can be evaluated. The condition for halting execution was defined to avoid having to explore a potentially exponential state space, which would result if branches conditioned on unknown values had to be followed.

Option processing

The Posix interface to the operating system provides a standard mechanism for applications to process command-line arguments. The interaction between this and DCA requires careful handling. To implement the dynamic analysis needed, OCCAM-v2 adapted LLVM's `ExecutionEngine` class, which forms the core of `lli`. When a program invokes `getopt()` during DCA, some arguments may be unknown. This can arise since OCCAM supports specification (via the manifest for a target) of residual arguments that will not be concretized during specialization.

To accommodate this case, custom handling of `getopt()` has been incorporated in DCA. Since the manifest requires concrete arguments to occur first, it is possible to infer the position at which residual ones will start. When the interpreter reaches a dynamic argument, DCA transitions to the phase where it uses static analysis to specialize the remaining bitcode using the program state at that point. This allows the specialized program to retain support for handling the residual arguments.

Multimodule dynamic analysis

Dynamic analysis of the target application proceeds from the program entry point. When a branch predicated on an unknown value is encountered, a specialized version of the application is generated using a subset of the program state at that point. To be able to determine whether a value is known, the implementation tracks memory as it is allocated in the LLVM module that contains the program's `main()` function. Initially, only these memory regions were

treated as known. If a value was read from unallocated memory, it was treated as unknown. This approach suffices for the case when the entire program is in a single module. In practice, applications may depend on code in multiple modules (libraries, for example). DCA can handle this using the LLVM `ExecutionEngine`'s FFI (foreign function interface) support for external calls. In this case, memory may be allocated by code that is not being analyzed. To accommodate this, a complementary approach was developed. By focusing on whole-program analysis, it can be assumed that memory is known by default. Instead, values derived from (dynamic) program-entry-point arguments are treated as unknown during specialization.

Conservative exclusions

➡ *Callbacks.* When interpreting an instruction that is a call to an external function, an argument may be a function pointer used to provide a callback. `ExecutionEngine`, however, does not provide a mechanism to expose a bitcode function that is being interpreted to the native code being invoked through FFI. Consequently, a check is performed. If an argument of a call that uses FFI is a function pointer, DCA halts and transfers the program's state to the specialized version. Support has been added for checking whether the called function can have side effects. If it cannot, DCA can continue.

➡ *Shared resources.* Similarly, specific external calls warrant special handling. One category where this arises is when a resource is shared between the target program and the specialization harness. Consider the case where the program being specialized closes the file

descriptor associated with standard output. Since LLVM's `ExecutionEngine` assumes that only the interpreted code is executing, no special handling is provided. As a result, executing such an instruction would also close the descriptor for the partial evaluator. Other such categories include calls to `exit()`, `vfprintf()`, and the `pthread_()` family. When such cases arise, DCA conservatively halts and proceeds to create a specialized program using its current state.

➡ *External globals.* The scope of the tracking of globals performed by `ExecutionEngine` is limited to variables in the program. If a global variable is allocated in a library dependency, this can result in a value that cannot be resolved. To accommodate this, DCA proceeds in this instance.

Variadic functions

LLVM's `ExecutionEngine` does not handle variadic functions. This was addressed by extending the implementation to treat such functions correctly if they are present in the target application bitcode.

The body of a variadic function can access its arguments through `va_start` and `va_arg` macros. The expansion of these may be represented in bitcode using corresponding LLVM intrinsics, `VAStartInst` and `VAArgInst`. If this is the case, `ExecutionEngine` can interpret these instructions. For many applications, however, `va_arg` is implemented with LLVM's `GetElementPtr` instruction (used to access an element in an array). Support was developed for handling the bitcode corresponding to `va_start` in such cases.

Handling arithmetic intrinsics

The intermediate representation of LLVM includes arithmetic intrinsics. This class of instructions can be compiled into performant native code by leveraging knowledge of the target hardware architecture's instruction set. The generated code, however, is opaque to the interpreter's analysis. Instead, these instructions are now translated into corresponding bitcode versions. Since this can then be evaluated in the modified `ExecutionEngine`, side effects are tracked directly.

Exposing wrapped libc calls

`glibc` (GNU C library) is the interface most widely used to interact with the Linux kernel. Internal functions (with names that start with '__') are accessed indirectly through wrappers. In such cases, `dlopen()` will not resolve the name of the wrapper to the underlying function. When the interpreter's FFI is used, however, the address of the called function must be passed in. By including the header file that declares particular wrapper functions, the address of a backing function can be identified. This approach is used to allow `stat()`, `fstat()`, and `lstat()` to be resolved when encountered (during a run of the modified interpreter).

EVALUATION

To evaluate the effectiveness of OCCAM-v2 for software winnowing, we ran it on a collection of 20 applications. The programs used for this purpose are the ones selected to study Trimmer,[1] another LLVM partial evaluator. In their work with Trimmer, the authors provide an explanation for why each of these applications was selected. Further, they

specify the set of program arguments used to specialize each of the applications. OCCAM-v2 was run on these programs with the same set of arguments on Ubuntu 18.04 using LLVM 10.

Effectiveness

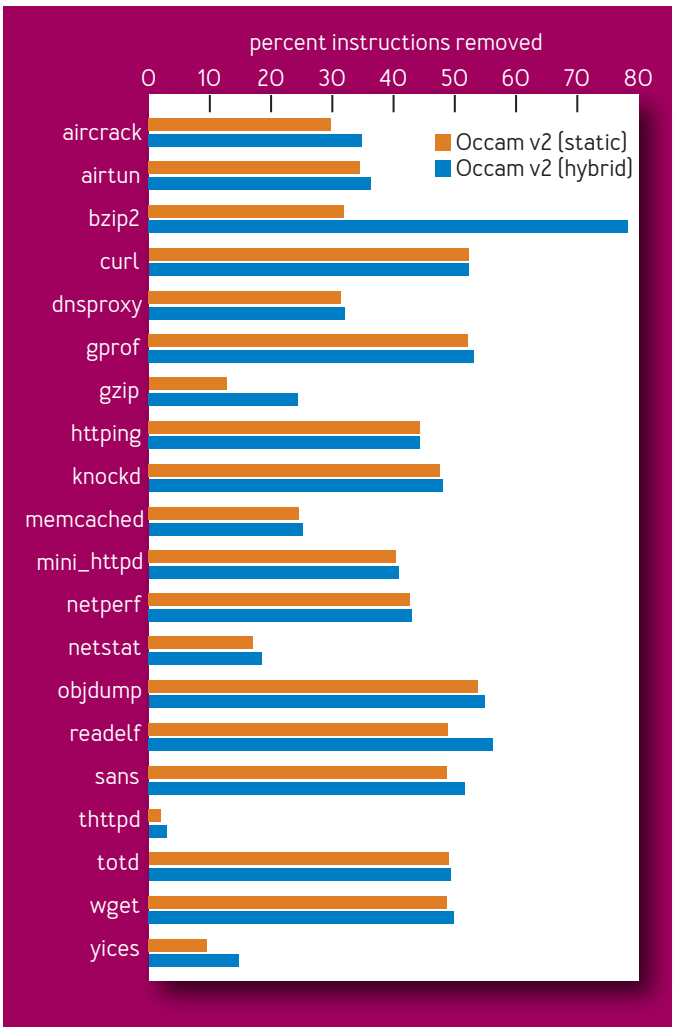
Figure 2 shows the results of applying (1) OCCAM-v2 with just the enhanced static-analysis functionality enabled; and (2) OCCAM-v2 using the dynamic analysis on the deterministic prefix of each program and the enhanced static analysis applied to the remaining suffix of each program. Each program is first compiled into LLVM's intermediate representation. The number of instructions in the bitcode is counted. Then, (1) OCCAM-v2 using only static analysis; and (2) OCCAM-v2 with both dynamic and static analysis applied. The percentage of instructions removed in each case is calculated and reported in the figure. In each case, the specialization context is the same as that used in the Trimmer evaluation.[1] As can be seen from the data, OCCAM-v2's combination of dynamic analysis followed by enhanced static analysis always results in more instructions being removed than in the case where just static analysis is used. On average, OCCAM-v2's hybrid analysis is able to remove 40.6 percent of the original program's LLVM IR (intermediate representation) instructions.

Efficiency

OCCAM-v2's static analyses have been selected to maintain scalability and efficiency. When necessary, precision is sacrificed to ensure performance. More

2

FIGURE 2: **% INSTRUCTIONS REMOVED**



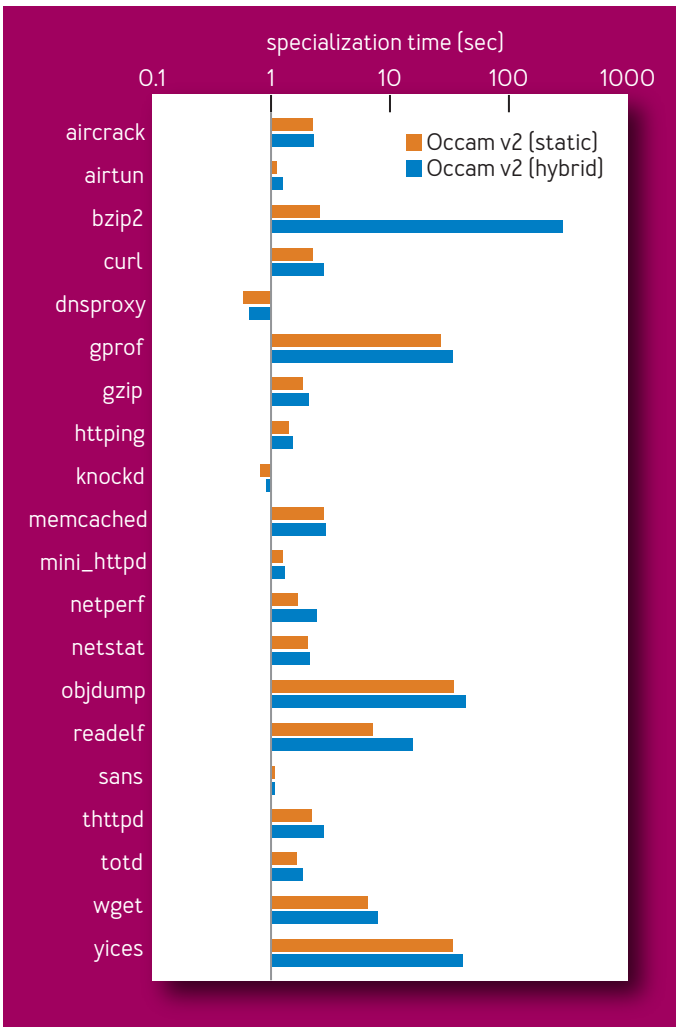
specifically, the pointer analysis used is based on Bjarne Steensgard's algorithm.[21] Further, the abstract interpretation framework is configured to use only computationally efficient reasoning domains, including Booleans, intervals, and strides. As a result, the average specialization time when using the static-analysis mode is 6.7 seconds.

For perspective on the efficiency of OCCAM-v2's approach, we can compare it to the one used by Trimmer. The latter opts for precision over scalability by using a pointer analysis based on Lars Andersen's algorithm,[2] as implemented in the SVF (static value-flow) framework.[22] The effect can be seen in the cost of the configuration annotation pass[1]. The three programs that take the longest for Trimmer's analysis are `objdump`, `yices`, and `gprof`, using 41.4, 23.6, and 16.3 minutes, respectively. In contrast, OCCAM-v2's entire static-analysis-based specialization for these three programs completes in 34, 34, and 27 *seconds*, respectively.

Figure 3 reports the time taken to specialize each of the 20 programs in the evaluation. The end-to-end time to run OCCAM-v2 on each target program was measured two ways: using only static analysis and combining dynamic and static analysis. Note that the time is reported on a logarithmic scale. The times needed for both the static- and hybrid-analysis approaches are provided. For most programs, the times are similar. `bzip2` is an exception since the specialization is configured to leverage the content of a target file, which requires the interpreter to step through the entire run. Note that this high specialization time yields a 78 percent reduction in the instruction count. Even with

3

FIGURE 3: **SPECIALIZATION TIME**



this outlier, the average hybrid analysis specialization time is 22.4 seconds.

LIMITATIONS AND MITIGATIONS

This section describes limitations encountered in practice and approaches for mitigating them.

Whole program assumption

OCCAM makes the assumption that it has access to the "whole program"—that is, the target application and all the code that it depends on—at specialization time. This allows it to reason that any code that cannot be invoked from the whole program is a candidate for elimination. This approach works for many, but not all, programs.

This concern arises for multiple reasons. An important case occurs when the source of an object, which must be linked into the program, is available only in assembly. OCCAM's analyses will not be aware of any function calls or accesses to external global variables in such an object since assembly cannot be built into LLVM bitcode. OCCAM may therefore prune symbols and functions that are needed. We have encountered this with both `musl libc` and the Linux kernel.

A similar situation may occur in the absence of assembly as well—for example, Apache loads modules dynamically based on its configuration file. These modules may introduce reverse dependencies—that is, the assumption that the main program contains particular functions, which may have previously been pruned based on the absence of calls to them. To address these cases, we have added support to OCCAM-v2 for specifying a set of functions

and global variables that should not be internalized. This prevents the dead-code elimination from pruning these elements.

Link-time symbol collision resolution

OCCAM iteratively performs constant propagation and dead-code elimination within modules, as well as function specialization across modules. When a fixed point is reached, the specialized modules are linked together. At this stage, symbols in modules may collide.

A symbol collision may occur if the same symbol was used in the application, as well as one of its libraries, or if it was defined in multiple libraries. The naming scheme for specialized functions minimizes the chance of new collisions arising. OCCAM-v2 addresses this with an intermediate step. Linking is internally staged to allow the symbols in the bitcode file specified in an argument to override subsequent duplicates. An intermediate linked bitcode file is constructed with `llvm-link`. This is then linked with the native libraries specified in the manifest using `clang++`.

FUTURE DIRECTIONS

➡ *Suffix simplification.* After the first phase completes, the memory snapshot contains state of two types. The first consists of values in registers that can be safely used to simplify suffixes. Such simplification includes use of LLVM's internal optimizations, as well as ones made possible by Crab's abstract interpretation over configured domains. The second consists of values in memory. To use these, there are two options. One approach identifies

special cases where this is likely to be safe in general. The second approach consists of employing a pointer analysis to ensure the soundness of suffix simplifications. Since this incurs significant computation cost, it had not been incorporated. In the future, support for this could be added. Users will need to explicitly activate it in cases where they are willing to incur the overhead during specialization.

➡ *Reducing overspecialization.* DCA relies on the fact that the prefix is a path that will always be executed by the target application, given a specific set of inputs. The attraction of this approach is that it promises a general mechanism for capturing external inputs. The current implementation explores a strategy that assumes by default that such values are independent of ones obtained from external input in the suffixes. It then adds constraints as needed to handle specific cases that violate the assumption.

For example, consider a variable in the snapshot that is a pointer to a location in memory. Assume it has been assigned a concrete value during dynamic analysis. If it occurs in the program in one of the suffixes, it will not be replaced with a constant. This is because at runtime that pointer may take on a different value. In the absence of such exclusions, the emitted binary will contain instances of overspecialization—that is, false-positive concretizations. An alternative strategy would be to implement the complementary approach: Assume by default that values cannot be concretized unless they derive from specified inputs.

CONCLUSION

OCCAM-v2 leverages scalable pointer analysis, value analysis, and dynamic analysis to create an effective and efficient tool for specializing LLVM bitcode. The extent of the code-size reduction achieved depends on the specific deployment configuration. Each application that is to be specialized is accompanied by a manifest that specifies concrete arguments that are known *a priori,* as well as a count of residual arguments that will be provided at runtime. The best case for partial evaluation occurs when the arguments are completely concretely specified.

OCCAM-v2 uses a pointer analysis to devirtualize calls, allowing it to eliminate the entire body of functions that are not reachable by any direct calls. The hybrid analysis feature can handle cases that are challenging for static analysis, such as input loops, string processing, and external data (in files, for example). On the suite of evaluated programs, OCCAM-v2 was able to reduce the instruction count by 40.6 percent on average, taking a median of 2.4 seconds.

Acknowledgments

This work was supported in part by the NSF (National Science Foundation) under Grant ACI-1440800 and in part by the ONR (Office of Naval Research) under Contract N68335-17-C-0558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or ONR.

References

1. Ahmad, A., Noor, R., Sharif, H., Hameed, U., Asif, S., Anwar, M., Gehani, A., Zaffar, F., Siddiqui, J. 2022. Trimmer: an automated system for configuration-based software debloating. *IEEE Transactions on Software Engineering* 48(9), 3485-3505; https://ieeexplore.ieee.org/document/9478582.
2. Andersen, L. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, Department of Computer Science, University of Copenhagen.
3. Bletsch, T., Jiang, X., Freeh, V., Liang, Z. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security,* 30-40; https://dl.acm.org/doi/10.1145/1966913.1966919.
4. Clam: LLVM front end for Crab. GitHub; https://github.com/seahorn/clam.
5. Cousot, P., Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, 238-252; https://dl.acm.org/doi/10.1145/512950.512973.
6. Cousot, P., Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, 84-96; https://dl.acm.org/doi/10.1145/512760.512770.
7. Crab: A C++ library for building program static analyses. GitHub; https://github.com/seahorn/crab.

8. Gelle, L., Saidi, H., Gehani, A. 2018. Wholly!: a build system for the modern software stack. *23rd International Conference on Formal Methods for Industrial Critical Systems*, 242-257; https://link.springer.com/chapter/10.1007/978-3-030-00244-2_16.
9. Gurfinkel, A., Navas, J. 2021. Abstract interpretation of LLVM with a region-based memory model. In *13th International Conference on Verified Software: Theories, Tools, and Experiments*, 122-144; https://dl.acm.org/doi/abs/10.1007/978-3-030-95561-8_8.
10. Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., Liang, Z. 2016. Data-oriented programming: on the expressiveness of non-control data attacks. In *37th IEEE Symposium on Security and Privacy*, 969-986; https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7546545.
11. Kuderski, J., Navas, J., Gurfinkel, A. 2019. Unification-based pointer analysis without oversharing. In *Proceedings of the 19th Conference on Formal Methods in Computer Aided Design*; https://ieeexplore.ieee.org/document/8894275.
12. LLVM compiler infrastructure; https://llvm.org.
13. Malecha, G., Gehani, A., Shankar, N. 2015. Automated software winnowing. In *Proceedings of the 30th ACM Symposium on Applied Computing*, 1504-1511; https://dl.acm.org/doi/10.1145/2695664.2695751.
14. Miné, A. 2001. A new numerical abstract domain based on difference-bound matrices. In *Proceedings of the Second Symposium on Programs as Data Objects*, 155-172; https://dl.acm.org/doi/10.5555/645774.668110.
15. Miné, A. 2006. The octagon abstract domain. *Higher*

Order Symbolic Computation 19(1), 31-100; https://link.springer.com/article/10.1007/s10990-006-8609-1.
16. OCCAM: Object Culling and Concretization for Assurance Maximization. GitHub; https://github.com/SRI-CSL/OCCAM.
17. Sadeghi, A., Niksefat, S., Rostamipour, M. 2018. Pure-call oriented programming: chaining the gadgets using call instructions. *Journal of Computer Virology and Hacking Techniques* 14(2), 139-156; https://link.springer.com/article/10.1007/s11416-017-0299-1.
18. Shacham, H. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 552-561; https://dl.acm.org/doi/10.1145/1315245.1315313.
19. Sharif, H., Abubakar, M., Gehani, A., Zaffar, F. 2018. Trimmer: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 329-339; https://dl.acm.org/doi/10.1145/3238147.3238160.
20. Smowton, C. 2015. I/O optimisation and elimination via partial evaluation. Ph.D. thesis, University of Cambridge.
21. Steensgaard, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, 32-41; https://dl.acm.org/doi/10.1145/237721.237727.
22. Sui, Y., Xue, J. 2016. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th ACM Conference on Compiler Construction*, 265-266; https://dl.acm.org/doi/10.1145/2892208.2892235.

Jorge Navas *is a static-analysis researcher at Certora. His focus is the design and implementation of automatic tools that can boost programmer productivity to make code more reliable and secure. He holds a Ph.D. in computer science from the University of New Mexico and a B.S. in computer science from the Technical University of Madrid.*

Ashish Gehani *is a senior principal computer scientist at SRI in Menlo Park, California. His research interests are data provenance, reproducibility, and security. He holds a Ph.D. in computer science from Duke University and a B.S. in mathematics from the University of Chicago.*

CONTENTS