Automated Software Winnowing*

Gregory Malecha
Harvard University

Ashish Gehani Natarajan Shankar
SRI International

ABSTRACT

The strong isolation guarantees of hardware virtualization have led to its widespread use. A consequence of this is that individual partitions contain much software that is designed to be used in a variety of environments and by a range of applications, while in practice only a limited subset is actually utilized. Similarly, the modular design of software has contributed greatly to the ability of application developers to quickly write sophisticated programs. However, in most instances only a small fraction of the functionality included in a particular software component is needed.

To address the resulting code bloat, we describe a tool OCCAM that combines techniques from partial evaluation and type theory with the goal of reducing the code in deployed applications. OCCAM can be used without annotating or otherwise modifying a program's source. It leverages configuration-time information to produce a version of the application that is specialized to the context in which it will be deployed. We present our algorithms, implementation, and experimental evaluation.

1. INTRODUCTION

Modular software libraries make it easier to write complex applications by providing reusable functionality for graphics, file operations, and networking. However, to be generally useful, libraries often include more functionality than any one application needs. For example, `libpng` [10] provides a sophisticated programming interface that allows image transformation, but many applications only use it for very simple tasks. In addition, complex libraries are often wrapped in simpler ones, simplifying their use while increasing runtime overhead.

The issue also spans multiple levels in the software stack, with overhead introduced by cross-layer dependencies. For example, the `miniblog` [13] application runs on top of PHP, which in turn depends on `libc`. Numerous functions in `libc` are only used by PHP code that is not used by `miniblog`. The functions would be included in statically linked binaries, using storage space on disk and needing memory during execution. The extraneous code may even be exploited in return-oriented-programming attacks [11]. In all of these instances, users get more than they want (which we argue is not good).

While most systems are not resource-constrained as they were in the past, the prevalence of implementation flaws suggests that even unused functionality in binaries and shared libraries can affect performance, reliability, and security. To counter the consequences of software bloat, we present an approach for specializing the code to the actual usage and eliminating unused functionality.

We propose *winnowing*, a static analysis and code specialization technique that uses partial evaluation. The process preserves the normal semantics of the original program – that is, any valid execution of the original program on specified inputs is preserved in its winnowed form. Invalid executions, such as those involving buffer overflows, may be executed differently. We also describe OCCAM, a tool that implements the techniques and present an experimental evaluation of its effectiveness.

At a basic level, the functionality of a program corresponds to the number of potential executions. For example, consider the socket function from the C standard library:

```
int socket(int domain, int type, int protocol);
```

Because the function takes three **int** parameters, it has a large number of potential behaviors (with one for each possible set of values for the parameters). Most combinations are unlikely to be used by any particular application. For example, web servers often open AF_INET and AF_INET6 sockets but do not require AF_APPLETALK or AF_ATMPVC sockets. The goal of winnowing is to remove unused behavior from the entire software stack, reducing the code that must be analyzed, while preserving the desired functionality.

Motivation

We discuss four reasons to winnow deployed binaries.

Specialized servers. The advent of commodity virtual machines has made highly specialized application servers the norm in many domains. For example, production web servers often host a single site, or even just a part of a site. Including overly general libraries in these virtual machines only introduces code bloat – that is, the isolation between partitions defeats the benefit of shared code.

Custom libraries. On some embedded platforms, being able to slim the libraries might mean that developers can use larger, more mature libraries and simply remove the pieces that they do not need. Many applications provide compile-time configuration for removing large chunks of code. However, these units are defined by the developer and not the consumer. This approach places a burden on the developer for maintaining different build configurations.

*This material is based upon work supported by the National Science Foundation under Grants IIS-1116414 and ACI-1440800. 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 National Science Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'15 April 13-17, 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04...$15.00.
http://dx.doi.org/10.1145/2695664.2695751

The above approach also does not provide precise control over what the application gets. For example, using a single function, even indirectly, from a large component requires the entire component to be present. In addition, this adds work for administrators, who must then keep their systems up to date with potentially irrelevant patches. This occurs because it can be difficult to determine whether a flawed function is used by an application or not, especially when it is written in a dynamic language.

Reduced analysis. In safety-critical systems, including large libraries is a liability. Smaller applications and libraries with fewer configuration options and less general functions are easier to statically analyze. In addition, partially evaluated code is often simpler [15], making static analysis more effective by making it more context aware.

Binary diversity. Specialized libraries also make it more difficult to usefully exploit buffer overflows. Platforms such as PHP contain the same set of functions in predictable places. Specializing each deployment instance to the application being run on it not only removes unnecessary functions, but also moves and changes the functions that remain. This makes it more difficult to craft attacks that use parts of these functions for attacks.

Contributions

We discuss the ideas and methodology for winnowing single applications. Our approach is completely automatic for homogeneous applications – that is, applications in which all of the code can be compiled to the LLVM [7] bitcode format. We selected LLVM since the framework includes frontends for several popular languages, including C, C++, and Java, has a well well-defined intermediate representation, and supports both static and dynamic compilation.

Our work provides:

– A tool for aggressively reducing the functionality of code (in both libraries and applications). The tool is robust enough to winnow large, industrial-strength programs that are used in practice.

– A modular way to incorporate external information into the winnowing process. This can be used to enforce policies such as "mail should never be called" or "system should only be called with the string `ls`", in a manner similar to aspect-oriented programming [6].

– An empirical analysis of the impact that winnowing has on binary size and execution performance. Our results suggest that winnowing produces binaries with no performance overhead and, depending on the application, can yield significant reduction in binary size.

– A tool that transparently augments the compilation process of large software projects, such as PHP and SQLite, to produce LLVM bitcode files as well as native object files.

Section 2 discusses our method for winnowing. This includes winnowing of both individual in Section 2.1 and multiple compilation units in Section 2.2. Our method for winnowing "open-ended" applications, such as interpreters, is outlined in Section 2.3. We discuss our implementation in Section 3. Three case studies are described in Section 4, and evaluated in Section 5. Section 6 concludes.

2. WINNOWING

We describe the process of winnowing with a simple example. Section 2.1 discusses winnowing of individual compilation units, which LLVM calls *modules*. Section 2.2 considers winnowing across multiple modules, which is useful for large libraries and necessary for dynamic loading.

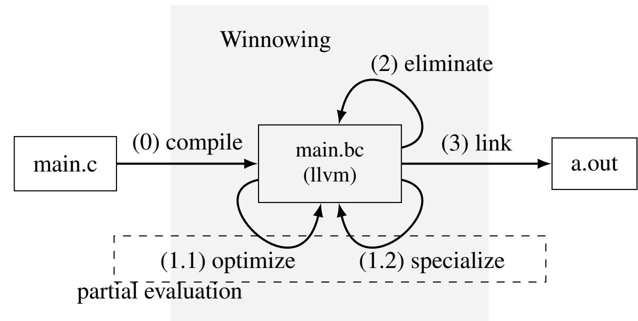


Figure 1: Intra-module winnowing operates on LLVM bitcode modules by repeatedly running partial evaluation (1.1 and 1.2) to specialize code and elimination (2) to remove unnecessary dependencies.

2.1 Intra-Module Winnowing

The basic unit of compilation in LLVM is the module. For simple programs it is often the case that they are entirely contained in a single module. In such cases, we can winnow an application using the algorithm depicted in Figure 1. For illustrative purposes, we discuss the algorithm by considering how to transform the following simple use of the C regular expression library. (We use C for expository purposes only. All analyses and transformations occur on LLVM bitcode.)

```
int regcomp(regex_t *preg, const char *regex,
            int cflags)
{ /* ... regcomp code ... */ }
void main(int argc, char* argv[]) {
  regex_t re;
  regcomp(&re, argv[1], REG_EXTENDED |
          REG_ICASE);
  ...
}
```

The algorithm proceeds in the following stages:

(0) Compilation: We begin by compiling the source code into LLVM bitcode. Our tools, discussed in Section 3, build LLVM bitcode automatically by instrumenting build scripts for Unix systems, such as those created with GNU autotools.

(1) Partial Evaluation: The core of the winnower is the partial evaluator, which we implement in two phases that the tool interleaves and iterates.

(1.1) Optimization: Partial evaluation begins by simplifying the code. The goal is to expose compile-time constants, remove dead code, and reduce known control flow. Applying the partial evaluator to our code snippet would reduce the bitwise-OR, resulting in the following:

```
void main(int argc, char* argv[]) {
  regex_t re;
  regcomp(&re, argv[1], 3);
  ...
}
```

As Fujita [2] and Smowton [12] have noted, the aggressiveness of the LLVM optimizer makes it a reasonable intra-procedural partial evaluator. In this work we leverage it exclusively for this phase. We use the LLVM `-O3` optimization profile, which combines a range of simplifications, such as global value numbering, heuristic

loop unrolling, sparse conditional constant propagation, constant folding, and known function simplification (which simplifies calls to libc functions such as strlen and memcpy). This optimization pass also performs some conservative inter-procedural optimizations on local functions such as inlining small functions and eliminating unused arguments.

(1.2) Specialization: After the optimization phase, we apply heuristics to more aggressively specialize across function boundaries. During this pass we look for function calls with compile-time known arguments. For example, the final argument to regcomp specifies options such as case-insensitive matching, and whether extended regular expressions are supported. In our example, as in most cases, the value of this option is known at compile time. To specialize, we duplicate the function, replacing occurrences of the formal parameter with the known constant, and remove the (now unnecessary) argument:

```
int regcomp_ICASE_EXTENDED(regex_t *preg,
                           const char *regex)
{ /* ... regcomp with cflags = 3 */ }
```

Applying the optimization pass to this specialized function will remove dead code by simplifying branches on cflags. It may also push constants into argument positions, revealing new opportunities for specialization.

Naively implemented, this pass has the potential to drastically increase the size of the code. However, a variety of heuristics can be used to control this. One such heuristic is to only specialize functions when the unspecialized version can be removed. This heuristic corresponds exactly to when we will be reducing the functionality of the program and is an idealized case. In practice, we have found it necessary to specialize more aggressively in order to reveal beneficial lower-level specializations. Our current heuristic is greedy, choosing to specialize whenever it sees a supported constant, while ignoring variable argument functions. Beyond integers, our specialization procedure supports floating point numbers, constant strings, null values of any type, and the addresses of global variables and functions.

Recursion: The primary difficulty of function specialization is cycles in the program's call graph. Without recursion, we can bound the number of specializations since we know that the LLVM optimizer will not infinitely unroll loops. In the presence of recursion, however, we must detect when specialization will diverge. Consider the following simple recursive function:

```
int foo(int start, int end) {
  if (start > end) { return start - end; }
  return foo(start + 1, end);
}
int bar(int x) {
  return foo(1, x);
}
```

Applying specialization and optimization leaves us with the specialized version of foo:

```
int foo_1(int end) {
  if (1 > end) { return 1 - end; }
  return foo(2, end);
}
```

which has another opportunity to specialize – namely, foo(2,?). If we naively continue this process, our specialization will proceed forever because we do not know the value of end. However, we have found recursion to be rare in the imperative code that we are specializing. Hence our current implementation is somewhat naive, choosing not to specialize recursive functions. It is worth noting that partial evaluators address this problem through the use of binding time analysis [1, 5]. In particular, techniques do exist for partial evaluation of recursive functions [4].

(2) Elimination: After partial evaluation has stabilized, or the binary is becoming too large to effectively deal with, we eliminate internal globals that are no longer necessary. Global variables may become unused when optimization removes unreachable code. Global functions will be eliminated if they were specialized and all call sites could be statically resolved to one of the specialized instances.

We implement this phase using three of LLVM's optimizers: globaldce, globalopt, strip-dead-prototypes.

(3) Linking: After iterating the winnowing process to a fixed point, we use the LLVM tools to link the winnowed code to build the binary.

This intra-module winnowing algorithm builds a semantically equivalent version of an input module while working to eliminate unnecessary functionality. The algorithm can be applied to any individual compilation unit, including static libraries and shared objects, since it does not modify the externally visible interface (at the level of abstraction of LLVM) by removing exported functions or changing their names or types. However, partial evaluation can drastically change the internal structure of the program, invalidating monitoring behavior (such as stack inspection) that is not manifest in the code.

2.2 Inter-Module Winnowing

Intra-module winnowing works well for many programs because most applications can be built as single modules by statically linking libraries prior to winnowing. However, this process breaks down with very large programs, and when programs must interact with shared libraries.

The difficulty of winnowing across module boundaries arises from the separation of code and the need to maintain compatible binary interfaces between independently winnowed modules. In addition, we would like our specializations to be reusable. For example, we may want to build a custom version of a standard library that supports several applications by including only the necessary functionality for those applications. We would then like to be able to automatically rewrite the client applications to reuse the same winnowed library.

Conceptually, we can break inter-module winnowing into three tasks. The first task enables the independent winnowing of modules by computing a module's dependencies (Section 2.2.1). The second task specializes modules (Section 2.2.2) and rewrites them based on generated specifications (Section 2.2.3). The final task "seals" the module, hiding symbols that are not used by other modules (Section 2.2.4) so that any unused functions can be eliminated during link-time optimization. The winnowing process iterates the specialization and sealing steps with intra-module specialization (Section 2.1) to produce the complete winnowed binary.

For illustrative purposes, we consider specializing the following simple code snippet, where bar is implemented in another module.

```
extern void bar(int, int);
int main(int argc, char* argv[]) {
  bar(argc, 5);
  bar(2, argc);
  return 0;
}
```

2.2.1 Computing Dependencies

The core composition mechanism is the computation and use of (function and global variable) dependencies of modules. In intra-module winnowing we had the code for foo and were able to specialize it immediately. However, now that the function is defined in another module, we can only record foo as a dependency of the client module.

To achieve meaningful specialization of functions, we will need information about the *values* of the arguments. We express this information using a singleton type system [8, 16]. Singleton type systems support a refinement of types using equality predicates. For example, the C type system can only express that the variable x is an integer (**int** x). With singleton types, we can refine this type and state not only that x is an integer, but that its *value* is 5 (**int**=5 x). Thus, the dependencies for our simple program are the following:

```
bar(int=?, int=5)
bar(int=2, int=?)
```

For uniformity we use `int=?` to specify an integer with an unknown value.

To compute the dependencies of a module, we traverse the call graph from the module entry points and look for uses of external symbols. Direct references and call sites are simple; we determine the target function and record the information as we saw above.

Indirect function calls and function pointers are more difficult. We rely on LLVM's tools for computing call graphs which employ standard program analysis techniques such as alias analysis and control and data flow analyses. These analyses improve OCCAM's ability to resolve indirect function calls.

When we cannot statically determine the target of the call, we must record the most general – that is, unspecialized – call to each possible target function. This is because we may not be able to change the binary interface without a global code rewrite. For example, if code stores the address of bar in a function pointer, we must conservatively record the most general dependence:

```
bar(int=?, int=?)
```

In some cases, the partial evaluation done during to intra-module winnowing will simplify this structure, allowing us to statically resolve the function. While this prevents further specialization, it is unavoidable in the general case – for example, when the program being specialized looks up function pointers in a table.

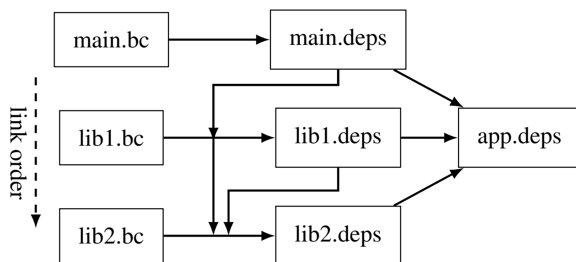


Figure 2: We compute the dependencies of a program by traversing call graphs from the `main` function. Calls to external functions are recorded in the interface and used as the roots for the traversal of containing libraries, following link order.

To compute the dependencies for an entire application, we start at the module that contains the program entry point and traverse the libraries that the code links against, as shown in Figure 2. To handle the general case of cyclic module dependencies, we iterate the process until subsequent dependency computations do not produce additional dependencies.

2.2.2 Specialization

Specialization uses each client module's dependencies to determine which functions to specialize, as depicted in Figure 3. We iterate over the list of calls in the dependency file and determine whether and how to specialize each function. Function specialization works the same way as in the intra-module case – that is, we duplicate the function body, remove the constant parameters, and substitute them in the body.

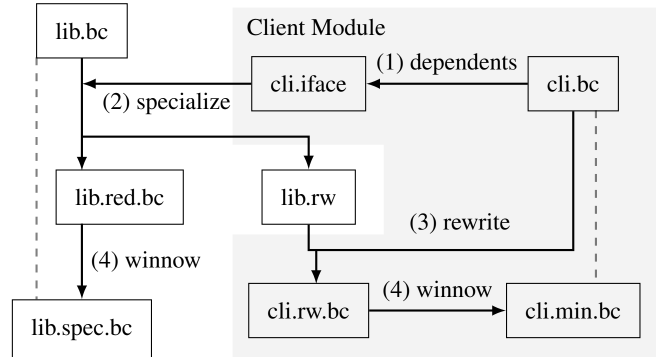


Figure 3: The specialization process uses a client dependency description (1) to generate a specialized module and a rewrite specification (2) that describes how to rewrite the client's calls to the library. The rewrite specification can then be applied to any client to make it use the new interface (3). The process concludes with applying intra-module winnowing to the specialized library (4).

The primary difference from intra-module winnowing is that we do not have access to the function call sites in order to rewrite them to call the specialized function. Therefore, we generate a *rewrite specification* that records how clients can be rewritten to use the more-specific interface. Considering the earlier specialization of bar, we would generate the following rewrites:

```
"bar"(int=?1, int=5) -> "bar_x_5"(int=?1)
"bar"(int=2, int=?2) -> "bar_2_x"(int=?2)
```

The ?1 and ?2 on the left are variable bindings that can be mentioned on the right of the arrow to refer to the specialized function's parameters. For example, the first rewrite says that calls to the function bar with the second argument of 5 can be rewritten in a meaning-preserving way with calls to the function bar_x_5, where only the first argument is passed.

2.2.3 Rewriting

After specialization, we use the rewrite specifications to update the client module. We iterate the call sites of all externally defined functions and look up the most precise rewrite that matches the call site. If a matching rewrite exists, we update the call site to use the specified function and rewrite the arguments accordingly. For example, applying the above rewrites in our simple example would result in the following code.

```
int main(int argc, char* argv[]) {
  bar_x_5(argc);
  bar_2_x(argc);
  return 0;
}
```

Note that the specialization step does not remove any of the unspecialized functions since call sites that do not match any rule will continue to invoke the unspecialized functions.

Rewriting is essentially the same as modifying the caller in intra-module specialization, except that it may be necessary to add new function prototypes to the module, and rewrites are read from the rewrite specification. Since we did not modify the generic function implementations, we can safely ignore indirect calls and storing external symbols in variables.

2.2.4 Sealing

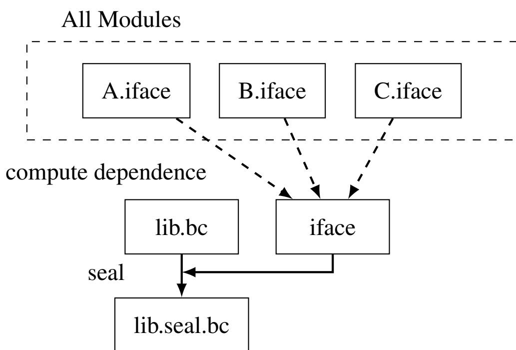


Figure 4: Sealing hides exported functions that are not referenced from outside the module. This enables more LLVM optimizations and analysis since the calling context of internal functions is more limited.

With our rewriting done, we can perform the inter-module equivalent of intra-module elimination. During this step, we make symbols not directly referenced by the outside world internal, effectively covering holes that others can use to interact with our modules. This allows intra-module winnowing to remove unreachable code and more aggressively optimize functions since it can statically analyze all potential call sites.

Sealing a module is easy using our interfaces, as illustrated in Figure 4. We simply iterate through all external symbols and make any that are not referenced by the interface internal. It is important to note that having accurate interfaces is essential to the correctness of this step since we will fail to link if we try to reference an internal symbol in another module.

2.3 Dynamic Checks

Winnowing is most effective when we can statically determine the interface for components. However, in some instances, such as when calls are made through function pointers, it is difficult to determine how functions are called. For example, during dynamic dispatch in object-oriented languages or when looking up functions tables in interpreters for dynamic languages.

For expository purposes, we consider a simple example:

```
extern int (*foo)(int);
extern int (*bar)(int);
void go(bool foobar) {
  int (*f)(int) = foobar ? foo : bar;
  f(2); f(3);
}
```

We know that foo and bar will be called only with arguments 2 and 3, but we cannot easily rewrite the code since we will need to use a different function in the calls to f. Here, we must maintain the same binary-level interface to foo and bar while not exposing all the functionality of these functions.

To achieve this, we statically rewrite the code to perform dynamic dispatch based on parameter values, similar to multi-methods. We maintain the same binary interface by constructing a wrapper function that will check the input arguments and dispatch to the appropriate specialized version or signal an error if the interface was violated. There are two potential places to perform this check, at call sites or at function definitions, and there are different benefits for each of them.

2.3.1 Rewriting the Client

The first choice is to rewrite the client code. In this setting, the library exports only the specialized functions (foo_2 and foo_3) but the client requires the non-specialized foo. To modify the client, we must implement the generic function by testing its arguments and dispatching to the appropriate customized function. For example, we implement foo in the client module as:

```
extern int foo_2();
extern int foo_3();
inline int foo(int x) {
  if (x == 2) return foo_2(); // foo(2)
  if (x == 3) return foo_3(); // foo(3)
  exit(1);
}
```

Applying partial evaluation to the client will result in this call being inlined and the conditions being removed in cases where x is statically known. Code that makes calls which don't conform to the interface will fall-through and signal an error (exit(1)).

2.3.2 Rewriting the Library

The alternative is to rewrite at the target function in the library. This is useful for enforcing interfaces on libraries that we could not statically validate. It can also be applied within individual modules to address the difficulties of dealing with indirect calls. To accomplish this, OCCAM duplicates the old foo implementation and rewrites it.

We replace the exported function foo with a function that checks the argument (to enforce the policy that foo should be called only with 2 or 3) and delegates appropriately or terminates the program. Since oldfoo is internal to the library (as indicated by the C **static** modifier), intra-module winnowing will specialize it and remove the general version from the final executable. After rewriting, the function is as follows:

```
int foo(int x) {
  if (x == 2) return oldfoo(2);
  if (x == 3) return oldfoo(3);
  exit(1);
}
static int oldfoo(int x) { /* foo code */ }
```

3. IMPLEMENTATION

We have implemented all the techniques that we have described. They have been applied to several examples to understand the usefulness of winnowing for real applications. Our code is implemented as a collection of LLVM compiler passes. We run it using the LLVM `opt` program through a set of Python wrappers. This means that each phase of the winnowing process produces a new program (or interface file). While less efficient than a pure C implementation that does not reify intermediate results, this approach gives us flexibility to experiment with additional transformations and specialization heuristics. The distribution of our code provides a tutorial of how our tools can be used to winnow an application.

To create bitcode versions of programs, we developed a collection of scripts to wrap extant build tools. This was necessary since the build procedures of large applications typically use a variety of tools, such as `autoconf`, `libtool`, `cmake`, and `make`. While these support a variety of platforms and configurations, they do not support LLVM bitcode as a target architecture.

To integrate with existing build scripts that check dependencies, our tool translates each command – for example, invocation of `gcc` or `ld`, into two commands. The first compiles a modified version to obtain LLVM bitcode, and the second is the standard version that generates the ELF version. Generating ELF objects confirms that the normal build succeeds and ensures that all appropriate dependencies are present. While suboptimal, especially in terms of build speed, our scripts are reasonably robust and are able to compile large programs and libraries, such as *PHP*, *SQLite*, and *uClibc*.

4. CASE STUDIES

Before diving into our examples, we recall our goals. First, we want to reduce the functionality of code. This does not necessarily mean reducing the size of code; rather, it means reducing the number of possible executions of the program. Duplicating a function will increase the code size, but it will not increase the complexity. Similarly, specializing a function with a particular argument value reduces its possible behaviors by limiting the values that can be passed to it. To measure this, we can look at the call graph of an application before and after winnowing.

Secondary goals are the reduction of code size and configuration complexity. Large libraries support flags to enable features at coarse granularity – for example, MySQL support in PHP. Winnowing provides library users an automated way to carve out more precisely the functions that they want. To measure this, we can compare winnowing to existing techniques, such as static linking with libraries (that uses only the necessary objects from an archive).

Our case studies cover two web servers, `nweb` [3] and `thttpd` [9], and the PHP interpreter used by web applications.

Callgraph after winnowing.

Figure 5: The callgraph of `nweb` shows how inter-module winnowing duplicates functions and removes parameters. Gray nodes denote specializations. Multi-edges indicate duplicated calls from inlining.

`nweb` is a static content web server that is implemented in about 200 lines of C. While its size and architecture do not make it a great candidate for winnowing (as it is a minimal application with few configuration options) they do make it easy to show exactly what winnowing does. `nweb` depends only on `libc` and is simple enough that *uClibc* satisfies its requirements.

Since `nweb` is configured from the command line, we begin winnowing by specializing the main function with respect to the arguments that we wish to supply. For this purpose, we use the command line `'nweb 8080 /root'` to tell `nweb` to listen on port 8080 and serve files from `/root`.

Figure 5 shows the effect of winnowing on `nweb`'s call graph. Bold gray boxes in the figure denote specialized functions. Note that of the 27 functions before winnowing, only 17 remain after. The remaining 10 are specialized or inlined into 11 different functions, each specialization removing at least 1 parameter. These specializations include:

– The log function has been duplicated three times, once for each type of status that can be logged, and the generic function has been removed. Partial evaluation of the specialized instances removes all of the interesting control flow from the specialized instances, making them straight-line code with direct function calls to sprintf, open, and write.

– The `libc` functions for setting up sockets (bind, listen and socket) each get specialized to handling TCP requests. The most interesting of these is socket, which is specialized for Internet (AF_NET), stream-oriented sockets (SOCK_STREAM).

– The `libc` function for handling software interrupts (signal) is specialized twice to handle SIGCHLD and SIGHUP. These specializations allow other tools to easily determine that the program only uses signals in a very limited way.

`thttpd` is the second web server we consider. It has a considerably larger number of features. `thttpd` is approximately 8500 lines of C code and supports CGI, using chroot on startup, in-memory caching, and `htpasswd` authentication. In addition to the core `libc` functionality, `thttpd` links against `libcrypt` in order to protect its `.htpasswd` file. `thttpd` can be configured both at runtime through command line arguments and at compile time with a header file that controls code generation using a set of 49 macros. These macros include many of the options that we would wish to expose to our winnower. For example, the path to `.htpasswd` is stored in the AUTH_FILE macro.

Winnowing `thttpd` results in 125 function specializations, which fall roughly into three categories.

– Specialization of timer creation (tmr_create). All of these calls can be specialized completely, including on the function pointer to run when the timer expires. This protects against data attacks in which an adversary corrupts a data value that is later invoked as a function.

– Specialization of buffer allocation (httpd_realloc_str). One call extends the size of the buffer to store headers:

```
httpd_realloc_str(&header, &maxheader,
  sizeof(headstr) + strlen(realm) + 3);
```

This call is specialized on the first two arguments since both are the addresses of global variables.

– Specialization of error codes makes up the third and largest group (with about 100 instances). Because our specialization framework does not yet support variable argument functions, we are not able to specialize the calls to snprintf in these error functions. However, the propagation of constants is performed. So it is feasible to achieve the extra specialization necessary to inject these constants.

PHP [14] is a popular programming language for web sites. It comes with an extensive standard library ranging from simple string and date manipulation to POSIX system calls like chroot. The "batteries-included" approach is useful for getting new applications up and running. At the same time, this presents a vast attack surface – especially for applications that use eval. To address this, we can winnow the PHP runtime functionality that is not required by the hosted application.

To demonstrate this, we build a PHP interpreter customized to serve miniblog [13], a small blogging framework written in PHP. As before, we use our toolchain to automatically build LLVM binaries for the PHP interpreter. After compiling PHP to an LLVM binary, we need to determine the interface that miniblog requires. Two complications arise in this task.

First, PHP programs are fed as text to the PHP interpreter. This means that we cannot use our LLVM passes to determine the interfaces necessary to run an application. Static evaluation of PHP is beyond the scope of this work. So we use regular expressions to capture textual patterns that correspond to PHP's function call syntax and compare the target function names with those supplied by PHP. Afterwards, we manually checked the results. This process could be replaced by a more semantic static analysis and/or by recording traces as the program runs. Even with these tools, we believe that manual auditing of this list is still useful for experts to see what is being depended upon by the underlying system.

From our static analysis we determined that miniblog relies on 52 functions in the PHP standard library, including string and file operations and MySQL functions. For comparison, we compiled the minimal interpreter that can be configured with PHP's compilation option (that includes the PHP standard library and the MySQL extension). This contains 1029 functions, including potentially dangerous functions such as system and mail. Some of the eliminated functions contained vulnerabilities (CVE-2011-1148 and CVE-2010-2191) in past PHP versions. These would not have been present in PHP deployments that had been winnowed.

The second problem is with the architecture of the PHP interpreter. The PHP interpreter stores library function implementations in a table that is dispatched to dynamically. Therefore, all calls through functions in this table will be overly conservative, most likely stating that any function could be the target. Even if we could pull the PHP code into the interpreter during partial evaluation using techniques like those developed by Smowton [12], the LLVM optimizer is unlikely to achieve the partial evaluation necessary to resolve calls through this table. In lieu of this, we note the structure of the table and modify the functions using the techniques described in Section 2.3. In the standard PHP interpreter, each library routine Xxx is implemented in the function zif_Xxx. Therefore, we can easily write hooks that will rewrite unused functions to errors.

For winnowing miniblog, we generate 997 rewrites of the form:

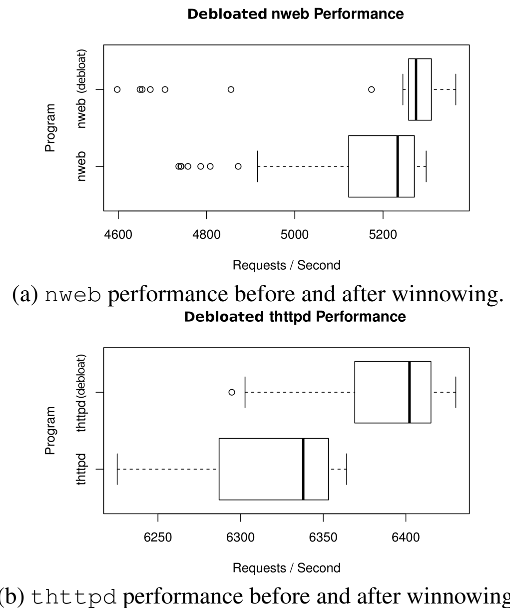
```
zif_system(?) => fail
zif_mail(?) => fail
```

For audit and debugging purposes, we have the implementation of fail log the call of the function that produced the violation.

5. EVALUATION

Performance. We compare the rate of requests that each of the servers can sustain when built with and without winnowing, in order to understand the runtime impact. Performance is measured in requests per second when serving a single, small static page. To control for network bandwidth, we performed our tests on `localhost:8080`. We use the Apache benchmarking tool `ab` to generate the requests. We ran 40 trials, each one making 5000 requests to the server. Tests were run on an otherwise lightly loaded Intel Core 2 quad-core desktop running at 2.4 GHz with 4 GB of RAM and Linux 2.6.38 with the TuxOnIce patchset.

To achieve a fair comparison, the baseline is optimized with LLVM's `-O3` before linking. Both versions were statically linked against uClibc-0.9.32, built with OCCAM and optimized using the standard configuration options from the uClibc build (using `-O2`).



(a) `nweb` performance before and after winnowing.

(b) `thttpd` performance before and after winnowing.

Figure 6: Performance comparison between basic web servers and their winnowed versions. At the 95% confidence level, winnowing of `nweb` does not impact its performance; however, winnowing of `thttpd` does improve performance.

Figure 6 gives the performance results for `nweb` (a) and `thttpd` (b). In the case of `nweb`, the difference is not statistically significant due to the large variation in the samples. The winnowed version of `thttpd` is statistically faster than the non-winnowed binary at the 95% confidence level. This performance improvement is most likely due to two factors. First, winnowing uses `-O3` as its partial evaluator while the standard compilation process uses `-O2`. Second, the duplication of functions exposes constants and creates more opportunities for compile-time optimization.

Size. The aggressive specialization policy we use for winnowing causes a 45% increase in the size of the `thttpd` module, as seen in Figure 7. This is a known problem with partial evaluation techniques. Much of this is due to duplication of the error-reporting functions that are actually parametric with respect to the arguments that they specialize. This issue can be addressed through a more conservative heuristic. It would only keep a specialization if it makes a significant difference or uncovers others that do.

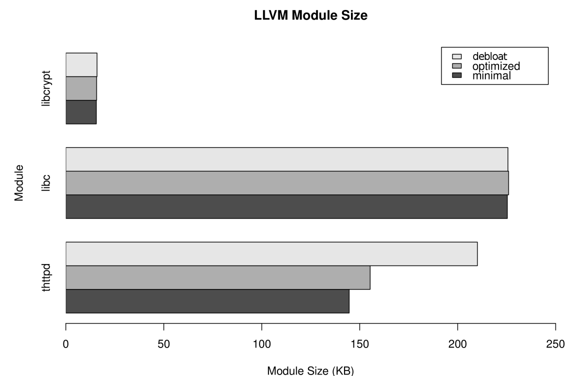


Figure 7: "minimal" is the size of the original. "optimized" is after it has gone through LLVM's `-O3`. "debloat" is the winnowed version.

Unlike the thttpd module, both libcrypt and libc have a negligible increase in binary size from winnowing. This is because most functions are leaf routines (such as string manipulation) or they immediately call into the operating system. Thus, within these individual modules there are fewer opportunities for inlining or specialization of functions. The additional size is due to specializable calls into the library minus the additional functions and constants that are pulled in by the static linker but not used by the thttpd module. These are not significant in highly optimized, low-level libraries that are statically linked since each object in the archive usually defines only a minimal number of functions.

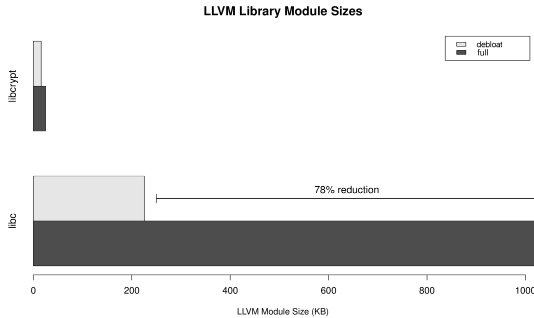


Figure 8: "full" is the size of the shared libraries used by `thttpd`, while "minimal" is their winnowed size.

Many common libraries, such as libc, are compiled to shared objects (or the system equivalent) rather than being statically linked. Normal shared objects contain the entire library, as shown with dark bars in Figure 7. Any application that dynamically links with such a library pulls in a considerable amount of extraneous functionality. By applying winnowing before building the shared object, we can reduce this considerably.

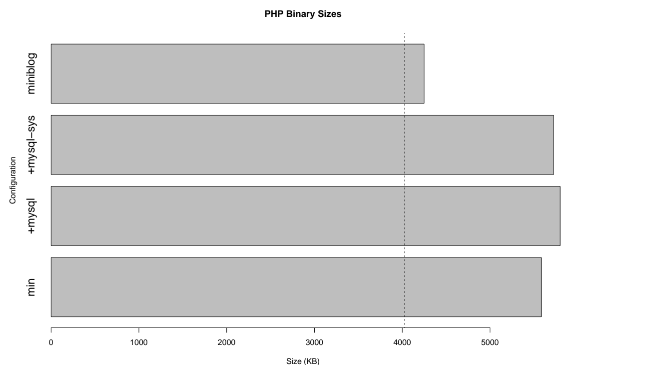


Figure 9: Size of PHP LLVM module with various winnowing configurations.

Figure 9 shows the results of applying the Section 4 technique for PHP. min denotes the minimal PHP module that can be built using PHP compilation configuration options. It is 5.5 MB in size, of which 4.0 MB (marked with the dashed line) is the interpreter (without any exported PHP functions). The remaining 1.5 MB is the standard library. +mysql represents the binary with MySQL support added. It requires an additional 4% storage (without including the MySQL library). +mysql-sys is the binary that results from adding a policy that prevents calls to a blacklist of 11 dangerous PHP functions. miniblog denotes a version of the interpreter that results after winnowing with respect to miniblog's interface. After intra-module winnowing, the LLVM bitcode shrinks from 5.8 MB to 4.2 MB in size, a 27% reduction; this is only 5.5% larger than the core interpreter.

The reduction of the PHP interpreter binary size from winnowing is similar to the minimization of shared objects that we saw for libc in the thttpd case (as illustrated in Figure 8).

6. CONCLUSION

We developed the OCCAM tool to specialize applications to their deployment context. We explained how our tools can be used to precisely remove functionality from an application in a completely automated way. We also argued that a small amount of manual effort can be used to address difficulties that arise from incomplete information, such as not knowing the program that an interpreter is running, or imprecise alias analysis.

7. REFERENCES

[1] L. Andersen. Binding-time analysis and the taming of C pointers. In *ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1993.

[2] S. Fujita. Partial evaluation with LLVM. http://llvm.org/devmtg/2008-08-23/llvm_partial.pdf.

[3] N. Griffiths. nweb: A tiny, safe web server. http://www.ibm.com/developerworks/systems/library/es-nweb/index.html.

[4] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1997.

[5] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*. Springer, 1997.

[7] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Symposium on Code Generation and Optimization*. IEEE Computer Society, 2004.

[8] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2), 1995.

[9] J. Poskanzer. thttpd. http://acme.com/software/thttpd/.

[10] G. Roelofs. Portable Network Graphics library. http://www.libpng.org/pub/png/libpng.html.

[11] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security*, 2007.

[12] C. Smowton and S. Hand. make world. In *13th USENIX Workshop on Hot Topics in Operating Systems*, 2011.

[13] spyka Web Group. Miniblog. http://www.spyka.net/scripts/php/miniblog.

[14] The PHP Group. PHP: Hypertext Preprocessor. http://www.php.net/.

[15] S. Weeks. Whole-program compilation in MLton. In *Workshop on ML*, 2006.

[16] H. Xi and F. Pfenning. Dependent types in practical programming. In *26th ACM Symposium on Principles of Programming Languages*, 1999.