REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
24-09-2013	Final Technical Report	(2012/09/25-2013/09/24)

4. TITLE AND SUBTITLE
Lean and Efficient Software:
Whole-Program Optimization of Executables

5a. CONTRACT NUMBER
N00014-12-C-0521

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
David Cok, Alexey Loginov, Tom Johnson, Brian Alliet, Suan Yong, David Ciarletta

Junghee Lim

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

GrammaTech, Inc.
531 Esty Street
Ithaca, NY 14850

8. PERFORMING ORGANIZATION REPORT NUMBER

- **9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
- Office of Naval Research
- 875 North Randolph Street
- Suite 1425
- Arlington, VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)
ONR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Distribution A: Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Modern software is typically produced using home-grown or third-party libraries and pre-existing components. Consequently, a finished executable often contains unneeded code, duplicate defensive checks, and extra layers of procedure calls. Such bloat contributes to excess memory footprint, slower performance, and security vulnerabilities (by hosting more return-oriented-programming gadgets an attacker can hijack).
The Layer Collapsing project is devising and prototyping techniques to substantially improve the performance, size, and robustness of binary executables. We are using static and dynamic binary program analysis techniques to perform whole-program optimization directly on compiled programs: specializing library subroutines, removing redundant argument checking and interface layers, eliminating dead code, and improving computational efficiency. A tool that successfully implements this goal will dramatically improve the way software is developed and deployed, providing new optimizations available late in the development process or even by the end user.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
					Sukarno Mertoguno
a. REPORT	b. ABSTRACT	c. THIS PAGE	SAR	21	19b. TELEPHONE NUMBER *(include area code)*
U	U	U			(703) 696-0107

"Lean and Efficient Software: Whole-Program Optimization of Executables"

Final Project Summary Report
(Report Period: 9/25/2012 to 9/24/2013)

Date of Publication: September 24, 2013
© GrammaTech, Inc. 2013
Sponsored by Office of Naval Research (ONR)

Contract No. N00014-12-C-0521
Effective Date of Contract: 09/25/2012
Requisition/Purchase Request/Project No.
12PR10102-00 / NAVRIS: 1100136

Technical Monitor: Sukarno Mertoguno (Code: 311)
Contracting Officer: Casey Ross

Submitted by:

Principal Investigator: Dr. David Cok
531 Esty Street
Ithaca, NY 14850-4201
(607) 273-7340 x. 146
dcok@grammatech.com

Contributors:

Dr. David Cok
Tom Johnson
Dr. Suan Yong
Dr. Junghee Lim
Dr. Alexey Loginov
Brian Alliet
David Ciarletta

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

Financial Data Contact:
Krisztina Nagy
T: (607) 273-7340 x.117
F: (607) 273-8752
knagy@grammatech.com

Administrative Contact:
Derek Burrows
T: (607) 273-7340 x.113
F: (607) 273-8752
dburrows@grammatech.com

1 Financial Summary

Total contract amount (1 year)	$399,984.00
Costs incurred during the final quarter (7/1/2013-9/30/2013)	$74,675.08
Costs incurred to date (to 9/23/2013)	$399,984.00
Estimated to complete	$0

2 Project Overview

Summary:
In this project, we investigated using binary-to-binary transformations to optimize raw executables (without source code) to reduce static size, improve runtime performance, and improve security; we produced a prototype system embodying our techniques, and operating on Linux executables. We improved GrammaTech's existing binary analysis infrastructure, so that we could analyze and rewrite a corpus of executables without changing their behavior (as measured by each executable's own test suite). Then we implemented and tested transformations that removed dead code, inlined procedures used only once, and converted dynamically linked procedures to statically linked. We also investigated techniques using specialization and partial evaluation. The test corpus we assembled includes over 100 openly available executables and tests for those executables.

Background:
Current requirements for critical and embedded infrastructures call for significant increases in both the performance and the energy efficiency of computer systems. Needed performance increases cannot be expected to come from Moore's Law, as the speed of a single processor core reached a practical limit at ~4GHz; recent performance advances in microprocessors have come from increasing the number of cores on a single chip. However, to take advantage of multiple cores, software must be highly parallelizable, which is rarely the case. Thus, hardware improvements alone will not provide the desired performance improvements and it is imperative to address software efficiency as well.

Existing software-engineering practices target primarily the productivity of software developers rather than the efficiency of the resulting software. As a result, modern software is rarely written entirely from scratch—rather it is assembled from a number of third-party or "home-grown" components and libraries. These components and libraries are developed to be generic to facilitate reuse by many different clients. Many components and libraries, themselves, integrate additional lower-level components and libraries. Many levels of library interfaces—where some libraries are dynamically linked and some are provided in binary form only—significantly limit opportunities for whole-program compiler optimization. As a result, modern software ends up bloated and inefficient. Code bloat slows application loading, reduces available memory, and makes software less robust and more vulnerable. At

the same time, modular architecture, dynamic loading, and the absence of source code for commercial third-party components make it hopeless to expect existing tools (compilers and linkers) to excel at optimizing software at build time.

The opportunity:
The objective of this project was to investigate the feasibility of improving the performance, size, and robustness of binary executables by using static and dynamic binary program analysis techniques to perform whole-program optimization directly on compiled programs. The scope included analyzing the effectiveness of techniques for specializing library subroutines, removing redundant argument checking and interface layers, eliminating dead code, and improving computational efficiency. Our use case is that these optimizations would be applied at or immediately prior to deployment of software, so that the optimized software is tailored to its target platform. Today, machine-code analysis and binary-rewriting techniques have reached a sufficient maturity level to make whole-program, machine-code optimization feasible. These techniques open avenues for aggressive optimization that benefit from detailed knowledge of an application's composition and its environment.

Work items:
In this project we developed algorithms and heuristics to accomplish the goals stated above. We embedded our work in a prototype tool that served as our experimental and testing platform. Because "Lean and Efficient Software: Whole-Program Optimization of Executables" is a rather long title, we refer to the project as *Layer Collapsing* and the prototype tool as ***LACI*** (for **LA**yer **C**ollapsing **I**nfrastructure).

The specific work items proposed at the beginning of the project are listed below and were all successfully addressed during the contract. Items listed as 'investigations' are ready for further work or implementation in the next phase of work on this topic.

1. The contractor will investigate techniques for specializing libraries and third-party components—i.e., techniques for deriving custom versions of libraries and components that are optimized for use in a specific context.
   1.1. The contractor will evaluate program-slicing and program-specialization technology developed independently at the referenced university.
   1.2. The contractor will investigate techniques for recovering intermediate program representation (IR) required for slicing and specialization techniques. The contractor will focus on the following tasks:
      1.2.1. Using static binary analyses for IR recovery.
      1.2.2. Using hybrid static and dynamic binary analyses for IR recovery.
      1.2.3. Studying trade-offs between the two approaches.
      1.2.4. Identifying the approach to be implemented in a prototype tool.
2. The contractor will attempt to implement a prototype optimization tool. This objective can be subdivided into the following subtasks:
   2.1. Implement IR-recovery mechanisms.
   2.2. Extend and improve the implementation of the slicing or specialization technology transferred from the university.
   2.3. Investigate the tradeoff between improved performance through specialization and the resulting increase in executable size.

   2.4. Investigate options for handling dynamically linked components and libraries.
3. The contractor will investigate techniques for further optimization of executables and for collapsing library interface layers. The contractor will consider:
   3.1. Selective inlining of library functions.
   3.2. Specialization of executables to the target platform.

   As time and resources permit, the contractor will attempt to implement these additional techniques in the prototype optimization tool.
4. The contractor will evaluate the prototype optimization tools implemented or received from the university experimentally. The contractor will use synthetic benchmarks, as well as real-world open-source software for the evaluation.
5. The contractor will maintain project documentation and produce comprehensive progress reports and a detailed final report.

3 Staffing

The following personnel participated in this project.

Dr. David Cok is the PI and is responsible for program management, infrastructure and the user-facing aspects of the resulting tool**.**

Dr. Alexey Loginov is the key architect of the binary analysis infrastructure**.**

Dr. Suan Yong and **Dr. Junghee Lim are** senior scientists having detailed knowledge of the binary analysis infrastructure and algorithms.

Brian Alliet is the principal implementation engineer.

Tom Johnson is the resident expert on the API for editing the Intermediate Representation of an analyzed binary.

David Ciarletta contributed to infrastructure development and measuring overall algorithm and tool robustness.

4 Accomplishments during the project

4.1 Overall plan

During the first six months of the project we planned the details of the project work, assessed the applicability of existing tools and algorithms, and performed some feasibility experiments. This included performing studies of possible useful transformations, implementing some of them, and implementing a test and evaluation infrastructure.

In the third project quarter, we

- continued our limit studies of possible useful transformations,
- improved our implementations of some of them,
- implemented an additional transformation, and
- collaborated with UW researchers on partial evaluation (a powerful approach to specialization).

Data Subject to Restrictions on Cover Page.

- prepared for and participated in a Program Manager-requested, ONR-sponsored workshop on improving performance of binary executables (4 June 2013). We also prepared for and participated in the ONR PI review (10-12 June 2013)

In the final project quarter we

- improved the robustness of the dynamic to static linking transformation
- re-measured our experimental results as our techniques were debugged and improved
- continued work with UW on specialization, generating a small-scale prototype
- performed experiments on additional test material: the regression suite we have in-house for testing the rewriting infrastructure and the GCC cc1 executable

The following sections provide details on these accomplishments.

4.2 Transformations

At the heart of the LACI system is a set of transformations that can be performed on an executable binary program, which may be accompanied by dynamically loaded libraries. Each transformation is expected to preserve all valid functionality of the program, but to provide some benefits. We are measuring benefits in three areas. These three goals may be differently emphasized in different contexts.

- Changes in size of the executable – reductions in size are expected to translate into better use of resources and better efficiency, e.g., due to decreasing the load on the instruction cache

- Changes in runtime performance – better runtime performance is always desired by users, and is also correlated with lower power consumption

- Changes in security vulnerabilities – transformations induce diversification, making the executable harder to exploit; additionally, removing some procedures from the executable reduces the number of return statements, and thus the number of potential ROP gadgets available to attackers.

We address each potential transformation with the following steps:

- **Limit studies**: where possible, before beginning to implement a transformation, estimate how much benefit is reasonable to expect from the transformation. Note that modern compilers implement many very sophisticated optimizations, so some transformations may turn out to yield minimal benefit. It is best to know how much benefit to expect before diving into implementation work; having an estimate also helps to evaluate the success of the implemented transformation.

- **Transformation**: implement the actual transformation on the binary executable (and libraries, if relevant), using the GrammaTech's CodeSurfer Intermediate Representation.

- **Candidate selection**: devise the decision procedure that indicates when to apply the transformation. For example, an inlining transformation is able to inline a procedure

Data Subject to Restrictions on Cover Page.

at any call site. Candidate selection will decide when to apply the transformation. For example, one reasonable approach is to inline only those procedures that are called exactly once, producing slight size, performance, and security improvements. However, it may be beneficial to apply inlining to procedures that are called more than once, increasing the application size, but providing more performance and security benefit. Candidate selection embodies algorithms for such decisions.

- **Evaluation on crafted applications**: Validate that the transformation achieves its benefits on subject applications designed especially to demonstrate the algorithm.
- **Evaluation on a test suite**: We compiled a test suite consisting of a sampling of realistic executables. This evaluation provides data on how successful the transformation is in practice.
- **Threats to validity**: The transformation or the heuristics about when to apply the transformation may intentionally or by necessity exhibit limitations. We documented the situations in which the transformation is expected to be sound (i.e., preserve all valid behaviors of the program) and when it may not be. It is possible that a transformation that is not sound on some classes of applications may still have significant benefit.

The final dimension of our study is the set of transformations themselves. We list the transformations (or variants) we investigated here and discuss them further in the subsections below.

- The NULL transform
- Dead code removal
- Procedure inlining
- Converting dynamically linked functions to statically linked ones (aiming to remove shared objects and dynamically loaded libraries)
- Specialization (including partial evaluation)

4.2.1 The NULL transform

The NULL transform does not perform any changes to the logical structure of the program. However, the program is analyzed into an Intermediate Representation and then written out again. As a result, basic blocks and procedures may be in a different order in the output executable, relocatable symbols will be in different locations, and failures to identify relocatable symbols may result in a faulty end result, among other changes. Thus the NULL transform is a test of the basic LACI infrastructure: if an application still performs correctly after the NULL transform, then the LACI infrastructure is working correctly (or at least correctly enough for that particular application).

Status: As part of our nightly testing on current development, we regularly applied the NULL transform to the whole test suite. Each rewritten test application passes all of its regression tests, that is, the application rewritten with the NULL transform has the same behavior on that application's regression tests as does the original application.

Data Subject to Restrictions on Cover Page.

4.2.2 Dead code removal

Transformation. The dead code transform removes code from the subject program that is not executed. Dead code can consist of procedures that are never called or basic blocks within a procedure to which control is never transferred. Closely related to dead code removal is the elimination of data areas that are never used.

Our first target is entire unused procedures. The transformation to eliminate these is straightforward, since it just consists of eliminating the procedure – no other code needs adjusting (besides the usual relocation of code that happens when object modules are assembled into a working executable).

The actual transformation does more than just check whether a procedure is called. Rather, this transformation removes code that is statically known not to be reachable via the control-flow graph from the entry point (usually `main`). Given an object file with three procedures, only one of which is used, linkers will nearly always link the two unused procedures, simply because they are in the same object file (object files are generally treated as an atomic unit). With the control-flow information provided by our binary analysis, we can determine that these extra procedures aren't reachable, and remove them, even if they mutually call each other. The dead code removal transformation works similarly to a garbage collection algorithm. It initially marks the entry point (e.g., `main`) as reachable, then continues traversing the control-flow graph, recursively marking procedures as reachable. When the algorithm terminates, any node left unmarked is dead and can be removed. Note that data references have to be taken into account as well. For example, `mov eax, proc1` makes proc1 reachable, even if control-flow analysis misses indirect calls to `proc1` enabled by the instruction.

Applying the transformation. Knowing when to apply the transformation is a bit trickier. One can readily determine whether there are any direct calls to a procedure from elsewhere in the program. Note that there may be groups of procedures that mutually call each other but if no procedure in the group is called from outside the group, the group as a whole is dead.

However, procedures can also be called indirectly. Knowing the values of all function pointers requires a much more complicated analysis; in general, it is undecidable. A conservative approach can note when the address of a procedure is taken, even if it cannot determine whether that address is ever used. However, even then, a particularly obfuscated program could construct function pointers from integer operations. Thus there is some soundness risk in applying this transformation.

Benefits. This transformation is expected to significantly reduce the size of an executable and improve security by reducing the number of ROP gadgets. No direct effect on runtime performance is expected; some speedup may result because there is less code to read from memory when the program starts and instruction-cache locality may be improved.

Limit study. To determine how much value may be obtained by dead code removal we performed the following study. Static and dynamic analysis data was collected to gather statistics on the frequency of calls for each procedure in the test executables. A total of 104 executables from the GNU coreutils package are regularly analyzed by the test infrastructure

Data Subject to Restrictions on Cover Page.

against completed transformations (currently the NULL transform). The analysis gathers information about procedures in each executable to identify if they have formal arguments, if there are local variables, how many places the procedure is called from, and if any of those call sites use constants as one or more of the arguments. Figure 1 shows a histogram of the number of procedures with N call sites (where N is the value on the x-axis) across all of the coreutils executables. A total of 5540 thunks are excluded from this data set. Thunks (small delegating procedures) make up about 1/3 of the total procedures across all executables analyzed and will be removed when the executable is rewritten.

Of immediate note is the number of procedures with no call sites (3,877, 38.1%). This is largely due to the use of shared objects among the executables in the coreutils package. When any procedure from one of those shared objects is referenced, the entire object is linked into the executable.

Results. The dead-code-removal transformation was applied to our test executables. The results are shown in Figure 2.

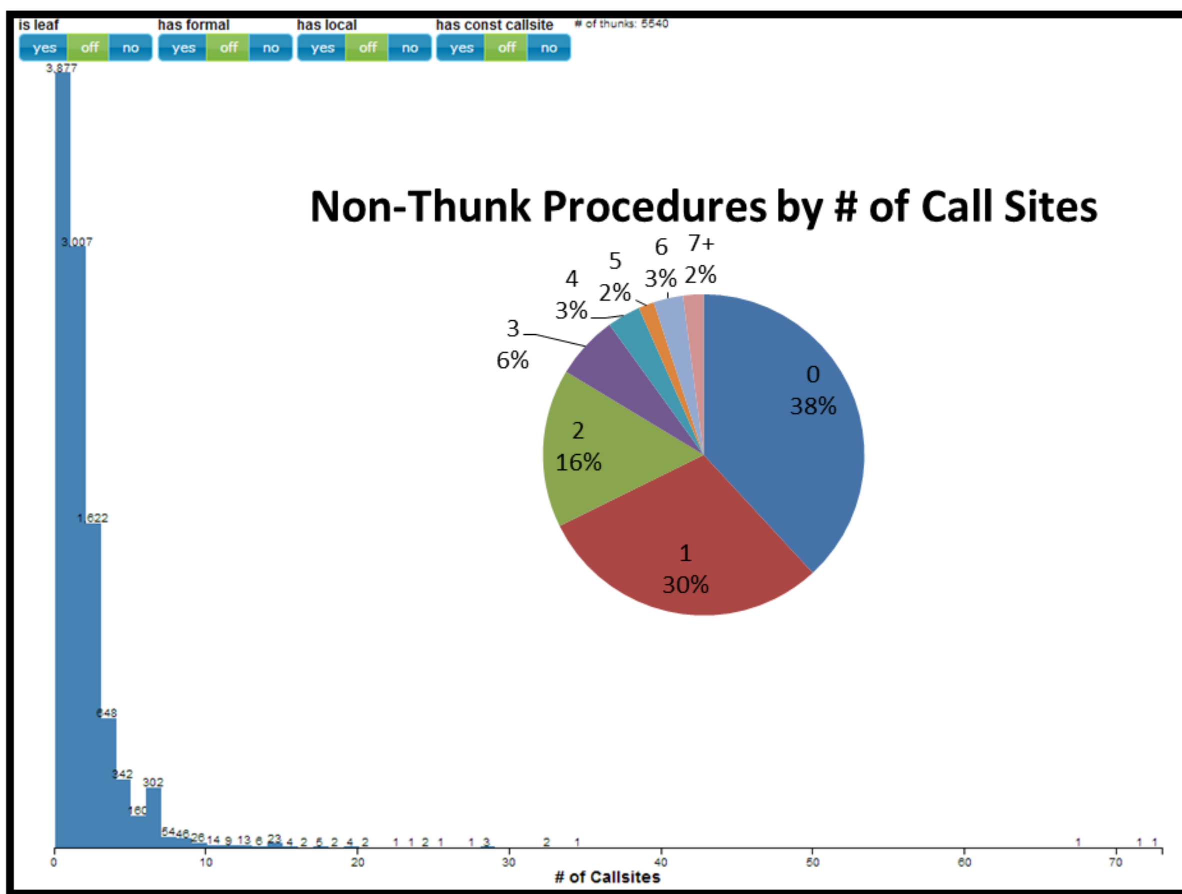


Figure 1 - Total Procedures by # of Call Sites for GNU coreutils

Data Subject to Restrictions on Cover Page.

N00014-12-C-0521
© GrammaTech, Inc. 2013

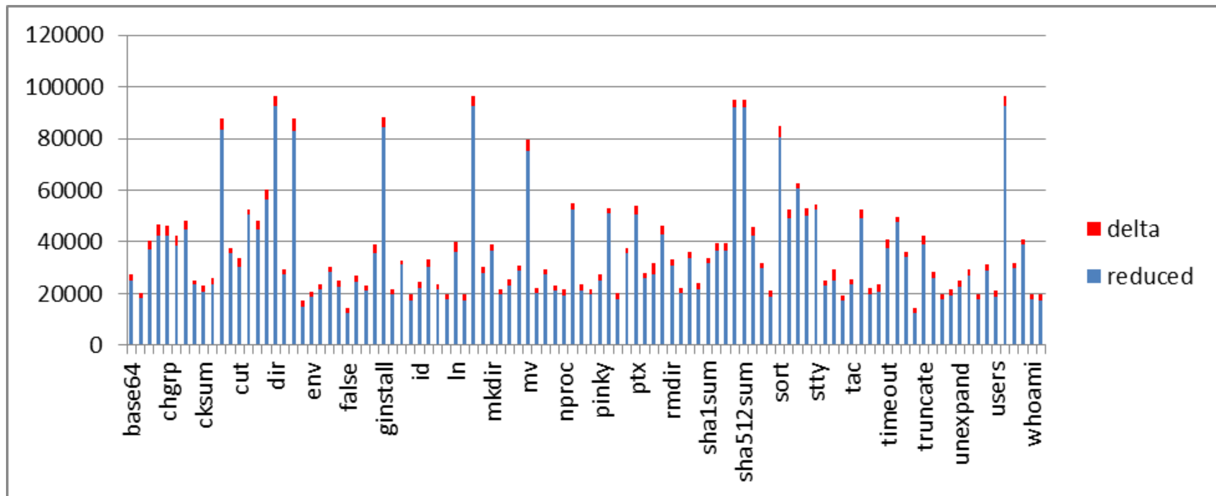


Figure 2 - Reduction in executable size from dead-code removal

The net result on this data set is that the transformation reduced the size of executables by 4-12%, with an average of 7-8%. We also counted the reduction in the number of procedures, shown in Figure 3.

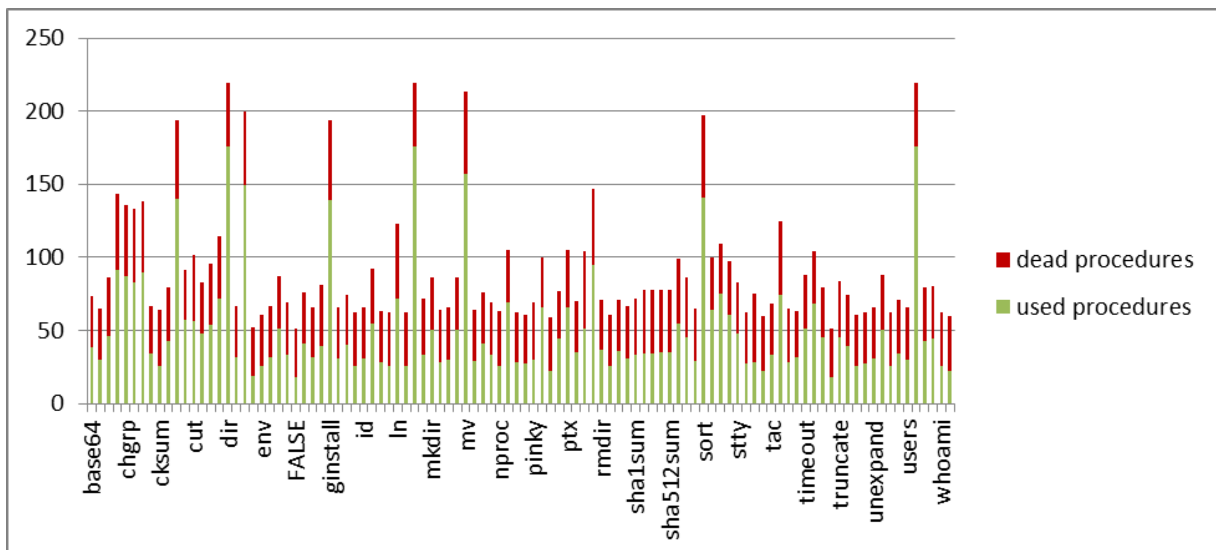


Figure 3 - Reduction in the number of procedures from dead-code removal

The reduction in number of procedures ranged from 20-65%, with an average of about 47%. The procedures removed are on average smaller than the average procedure in the applications.

4.2.3 Inlining procedures

Limit study: The data needed to assess the potential of inlining is present in the pie chart of Figure 1. We see that about 30% of procedures are called just once. If those are inlined there will be slight improvements in memory size and runtime performance and there will be a significant reduction in the number of call-return pairs, improving the security footprint. In assessing each of the executables individually, we see a reduction in the number of functions

Data Subject to Restrictions on Cover Page.

and return instructions to 62-85% of the number in the original program, with an average of 74-76% (a 25% reduction).

We measured the effect of inlining on a set of other metrics as well.

Metric	Number remaining compared to original
# of branches	93%
# of calls	84%
# of indirect calls	42%
Size on disk	94%
# of functions	52%
# of returns	53%

Transformation. The inlining transformation replaces a call to a procedure with the actual text of the procedure, removing control transfer (usually effected by `call` or `jmp` instructions). The complete application of the transformation can be carried out using the following intermediate steps (not all of these steps are necessary for improved performance, power consumption, or security):

- Replace the call to the callee and return to the caller with jump instructions.
- Remove the jump instructions and reorder the code so that the instructions of the callee are directly between their immediately preceding and following instructions in the caller.
- Combine the activation records of the two procedures so that the stack size is adjusted just once to account for the callee's local variables and saved registers. Adjust the stack to account for the removal of some operations, such as the saving of the return address.
- Remove what used to be the copies of values to stack locations that held the actual arguments of the callee. Change the callee to use the values directly from the caller's activation record (or as global memory locations). This can be accomplished by analyses such as copy propagation and constant folding.
- Adjust the stack to avoid saving space for the return address.
- Eliminate unnecessary register-save operations, e.g., registers saved by the caller that are not modified by the callee.

4.2.3.1 Inlining procedures without cloning

Applying the transformation. This transformation can be applied whenever a procedure is called just once and its address is never taken. Some of the same caveats apply as for the analysis of calls to determine dead code.

Benefits. This transformation is expected to decrease the size of an executable and increase its runtime performance only slightly. (The improvement comes from avoiding unnecessary parameter marshaling and control transfers.) However, the transformation will reduce the

number of return instructions and therefore improve security by reducing the number of ROP gadgets.

Limit study. The limit study results for dead code above also contain information about inlining possibilities. In the data set we analyzed, 3007 (29.5%) of the procedures had only one call site and are reasonable candidates for inlining.

Results. This transformation has been successfully implemented. The number of procedures removed equals the number of RET instructions removed, and is a measure of the security improvement. All test applications tested successfully after the inlining transformation.

4.2.3.2 Inlining procedures with cloning

Transformation. Inlining can be applied even if a procedure is used in more than one place: a separate copy of the procedure can be inlined in each location where it is used.

Applying the transformation. Just as for inlining without cloning, this transformation can be applied wherever the callers of a procedure are precisely known and its address is never taken and used. The challenge is to balance the size of the inlined procedure and the number of inlined clones against the resultant increase in executable size.

Benefits. This transformation is expected to increase the size of the executable and possibly increase the number of ROP gadgets. The direct effect on performance will stem from reducing parameter passing and control transfers.

Results. We did not implement the cloning operation, as we did not think we would learn anything substantially new. The implementation is expected to be straight-forward in the future.

4.2.4 Converting dynamically linked functions to statically linked

Benefits. Many library routines are made available to executables as shared objects or as dynamically loaded libraries. The advantage of this system design is that, when multiple executables use the same routines, the system need load those routines into memory only once. If each routine were linked statically into each executable, the sizes of all of the executables would increase significantly. However, the presence of shared objects means that there are also many procedures that are part of the address space of an executable but are not used by it; thus there is dead code and an increased number of possible ROP gadgets.

Transformation. The transformation is to statically link in only the needed functions from a dynamically loaded library module, removing the need to use shared objects and DLLs. This requires essentially reimplementing (or reusing) much of the functionality of the linker and loader.

Applying the transformation. The transformation can be applied whenever the analysis can be confident that it knows which of the procedures in a shared object or DLL are called and at what call site. The benefit of the transformation to the transformed application is easy to understand. However, real-world applications of the transformation need to take into account the effect on the whole system: will other applications run more slowly because the

Data Subject to Restrictions on Cover Page.

extra copies of previously shared code affect the virtual memory and the instruction cache. Distinguishing standard from application-specific libraries may provide an acceptable answer.

Limit study. The limit study above identified a large fraction (38%) of procedures that were unused; most of these are from shared objects.

Results. The essence of the implementation of this transformation was to understand the dynamic linking and loading functionality of ELF libraries and executables and implement steps to perform that functionality statically in our transformation. In order to maximize the benefits of the transformation, we implemented the ability to transform every library-function call from indirect to direct.

We applied the transformation to our collection of executables; coreutils executables were modified to be compiled using dynamic linking of a key shared library. The transformation is performed in two steps. First, we applied the transformation to convert dynamic linking to static linking. Next, we eliminate any procedures that are part of the dynamic library but are not used by the target executable (using the dead-code removal transformation described earlier). These unneeded procedures contribute to the (in)security footprint of the executable. They are not used, but they are available within the executable code of the program (whether loaded dynamically as in the input form of the program or statically as in the intermediate form); thus malware could make use of these “dead” procedures or their corresponding ROP gadgets.

As a result of the transform we observed the following changes in metrics:

Metric	Fraction of original
Size on disk	17% (range: 10%-33%)
Memory size	17% (range 10%-31%)
Number of return statements	7% (range: 0.9% - 24%)

There is a significant reduction in all of our metrics. Note particularly the significant reduction in return statements. This is a proxy for a significant reduction in ROP gadgets and therefore a significant improvement in security footprint. The changes in size are also significant, though they apply most precisely only when a single executable using the library is running (otherwise the overall reduction is smaller since the original dynamic library is shared).

During this contract, we extended our IR construction to improve the robustness of this transformation. The extension involved respecting the library-name component in library-function names that were requested to be loaded by applications. Not doing this led to problems when functions with the same name appeared in different libraries (and were requested to be loaded by the application). This situation is relatively uncommon but was encountered in one of our tests, thus prompting us to design a solution.

Data Subject to Restrictions on Cover Page.

Previously we combined all shared libraries to be statically relinked into a single global namespace. This behavior doesn't match the behavior of the runtime dynamic linker, and necessitated a workaround for symbols that were present in both the executable and the shared library. Common sources of the duplicate symbols were startup code and linker-defined symbols (`_start`, section boundaries, etc.) and could safely be dropped or renamed. But this simple solution was fragile; it would break in the presence of user-defined duplicate symbols. Our improved solution gives each shared library its own namespace and associates symbols with their origin. This allows us to handle symbols with the same name present in the executable and a shared library or in more than one shared library. References to shared libraries in the executable are not explicitly annotated with the expected shared library. To resolve these references to the appropriate definition in the set of shared libraries, we perform the same steps as the runtime dynamic linker. This allows us to perform safe transformation on new examples and eliminates the need for special cases.

4.2.5 Specialization and partial evaluation

Specialization and partial evaluation are terms that are often used to refer to similar techniques (we will adhere to the common practice of viewing partial evaluation as a kind of specialization, as will be elaborated below). The primary goal of specialization is to improve the runtime cost of a program by optimizing the program's code for the restricted context in which the program components (e.g., functions) are used. These are some example use cases:

- A program performs many computations but only some of those computations affect the program outputs (e.g., some command-line processing has no effect on the output or observable side effects). Instructions that do not affect the outputs can be removed.
- General purpose library routines may be used in just a few contexts within a program. As a result, some of the instructions within the library routine may be unused and removable. For example, perhaps a procedure tests that a given argument is non-null, issuing an error message if it is null. If analysis shows that in all of the calling contexts in a subject application the caller assures that the argument is indeed non-null, then the error-checking code within the library routine can be removed.
- Values known at load time (e.g., what kind of a platform a general library routine is being executed on) may enable the more general kind of specialization when instructions are *partially evaluated* to produce simpler or more efficient specialized code.

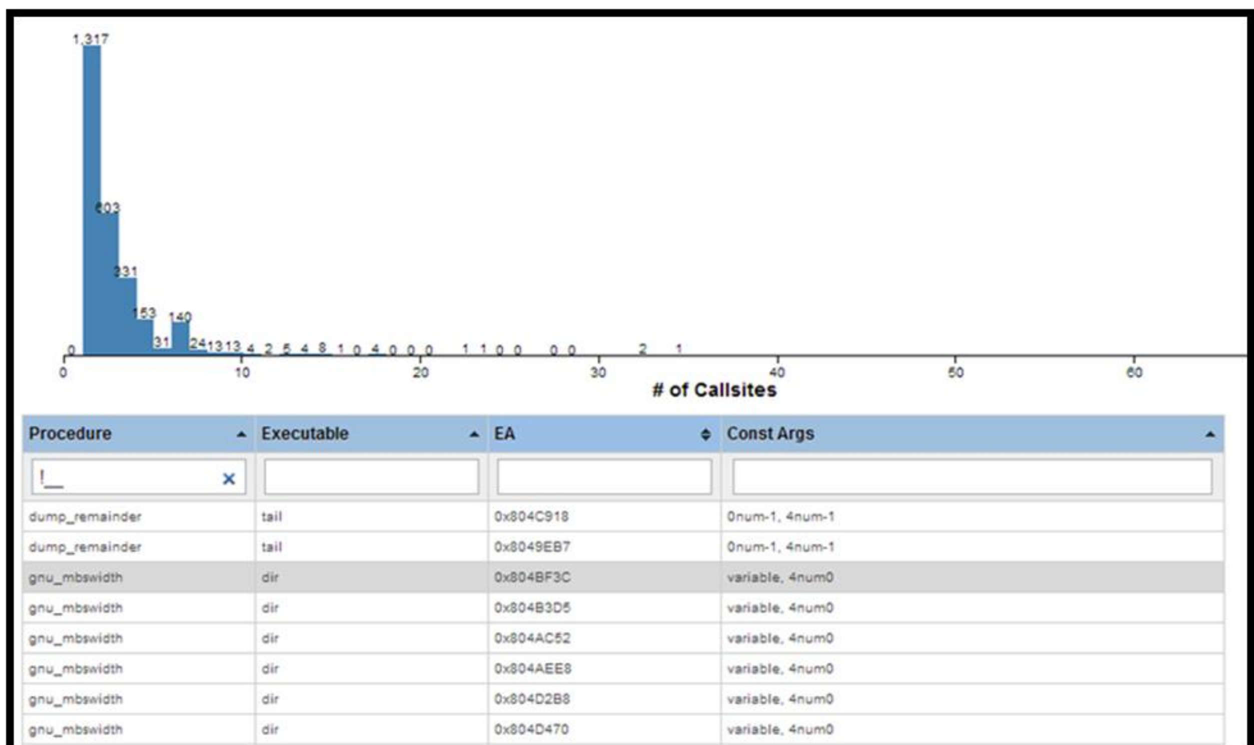
These transformations can be implemented with varying degrees of complexity.

- A dependence-based specialization considers only the data and control dependences within a program, given its limited context of use, and does not try to concretely or symbolically evaluate the program. This can provide a conservative amount of code reduction. This is implemented in the University of Wisconsin's specialization-slicing prototype.
- Partial evaluation usually refers to specialization that involves simplifying code based on information known statically. For instance, constant propagation can be used to simplify uses of a procedure's argument that is known statically. Partial evaluation begins with binding-time analysis that determines the division of arguments to a function into those known statically and those determined only at runtime.

Data Subject to Restrictions on Cover Page.

- Polyvariant specialization (or partial evaluation) is capable of creating multiple copies of a procedure specialized to multiple distinct contexts (e.g., different constant values for a given argument known statically). Transformations of copies can be aggressive, as they can be tailored to a single context. However, the gain needs to be balanced against the indirect costs of creating many similar forms of a procedure.
- Polyvariant-division specialization (or partial evaluation) is prepared to consider multiple divisions of arguments into statically known and dynamically determined. This requires the ability to create multiple copies of the procedure to be specialized. Prof. Reps's group at UW Madison has been working with GrammaTech to extend the specialization-slicing prototype into a polyvariant and polyvariant-division partial-evaluation tool (this work will continue beyond the project reported here).

Limit study. One initial study of the possibilities of specialization was to observe in our data set how frequently the arguments to procedures are constants and which constant values were used. These locations are potential targets for specialization via partial evaluation. Figure 4 displays the histogram filtered for procedures with at least one call site that uses a constant argument. By selecting one of the bars in the histogram we can then drill down into the data to see details on each call site and determine the number of specialized copies that would be required. For example, procedure `gnu_mbswidth` in the `dir` executable is called 6 times with an explicit zero passed as the second argument. This provides an example of a prime target for specialization via partial evaluation because the original procedure could be replaced by one specialized copy.



Procedure	Executable	EA	Const Args
dump_remainder	tail	0x804C918	0num-1, 4num-1
dump_remainder	tail	0x8049EB7	0num-1, 4num-1
gnu_mbswidth	dir	0x804BF3C	variable, 4num0
gnu_mbswidth	dir	0x804B3D5	variable, 4num0
gnu_mbswidth	dir	0x804AC52	variable, 4num0
gnu_mbswidth	dir	0x804AEE8	variable, 4num0
gnu_mbswidth	dir	0x804D2B8	variable, 4num0
gnu_mbswidth	dir	0x804D470	variable, 4num0

Figure 4 – Call Site Histogram and Detail Table Filtered for Procedures with a Call Site with a Constant Argument

Data Subject to Restrictions on Cover Page.

A second aspect of our limit study was to determine, at each branch in every procedure of the test corpus, whether the branch is always taken in one direction. In the entire test corpus, it appears that 3-11% of branches always follow one direction (for that branch), with an average of 5-6% over the test corpus. This is a surprisingly high percentage, as it is in addition to any dead branch removal performed by the compiler itself. On investigation, most of these cases are in startup code (which depends on fairly fixed aspects of the environment) or are actual cases of constant arguments to library functions. Similarly, about 4% of individual instructions have known operand; thus they could be evaluated statically, reducing the program size and runtime. The result of the above two analyses would result in about 3% of additional code being dead, just from computed values known to be constant by static analysis.

Examples. In addition to the limit study, we also looked for individual examples that may constitute good targets for this transformation. Below are three such examples.

1) `Fnmatch`

```
int fnmatch_ext(...) {
    /* big function, lots of code to process "extended"
patterns */
}

int fnmatch(const char *pattern, const char *string, int flags)
{
   char *p = pattern;
   /* ... */
   switch (*p++) {
   /* ... */
        case '*':
          if ((flags & FNM_EXTMATCH) && *p == '(')
            {
              res = fnmatch_ext (c, p, n, string_end,
no_leading_period, flags);
              if (res != -1)
                return res;
            }
    /* ... */
}
```

A snippet from the `fnmatch` procedure is shown above. It is a library method used in many places to do shell-style wild-card pattern matching. The `flags` argument is used to control various options of the procedure. However, `flags` is almost always a constant. Within a specific executable it is only used in a few ways. Thus the constant values used for `flags` could be constant-propagated into the `fnmatch` procedure, performing specialization and partial evaluation for those specific values of `flags`. This transformation has the potential to eliminate many of the case statements in the body of `fnmatch`, reducing the amount of code and with it, the attack surface of the executable.

We can see that this transformation effectively would perform an optimization that the programmer could have applied. However, a source-code implementation would result in duplicate or similar code in multiple places, creating a maintenance and reliability nightmare.

By performing the transformation automatically, on the binary executable, we gain the advantages of the optimization without those problems.

2) `fts_open`

A similar example is found in `fts_open`, whose code is summarized below:

```
/* Options check. */
if (options & ~FTS_OPTIONMASK) {
        __set_errno (EINVAL);
        return (NULL);
}
if ((options & FTS_NOCHDIR) && (options & FTS_CWDFD)) {
        __set_errno (EINVAL);
        return (NULL);
}
if ( ! (options & (FTS_LOGICAL | FTS_PHYSICAL))) {
        __set_errno (EINVAL);
        return (NULL);
}
```

Again, the `options` value is a parameter to the `fts_open` procedure and is nearly always a constant. Thus, entire blocks of code can be eliminated if the constant value of `options` is constant-propagated and the conditions of the `if` statements are evaluated; if the condition can be determined to be always false, the block guarded by the `if` statement can be safely elided.

3) `divdi3` and `moddi3`

The last example discusses the `divdi3` and `moddi3` 64-bit operations that implement division and modulus operations on 64-bit numbers in 32-bit architectures. Both operations perform a check that the divisor is not zero. However, in the test suite, these functions are almost always called with constant, non-zero divisors. For the calling contexts in which it can be soundly established that the input value is a non-zero constant, a specialized version of `divdi3` or `moddi3` can be constructed (automatically, by LACI) that omits this check.

This example is different from the previous two. In the other two examples we would want to determine the set of constant values that the relevant input parameter could take; those values each create a possible specialization (or perhaps groups of values might imply the same specialization). In the case of divide and modulus, the relevant categories are just two: (a) zero and (b) non-zero; the non-zero category can have many, perhaps unknown (but always non-zero) values. If we can reason about just these two domain elements and soundly establish that a calling context is always in the (b) category, without necessarily knowing the specific values of the divisor, we can perform the specialization.

Status. The implementation of this transformation is in progress. See the next section on collaboration with UW on this topic. We are taking a new approach to this problem, which did not result in a complete implementation by the end of the contract but the infrastructure and testing mechanisms that we have put in place will enable UW researchers to design and evaluate an advanced prototype.

4.3 Evaluation of specialization slicing

During this reporting period we held a number of discussions with our collaborators at the University of Wisconsin regarding work on specialization slicing[1]. Specialization slicing was focused on the use of dependences for identifying whether or not a given instruction should be included *in* the specialized output. (The term "slicing" reflects the reliance on dependences.) Such specialization cannot capture information about correlated branches and it cannot simplify the input program; it can only remove unneeded instructions.

After additional experimentation, UW researchers decided to implement deeper forms of specialization—ones that can perform *partial evaluation* and thus can simplify the input program, instead of just removing unneeded instructions. We held several discussions about this work, in order to understand and provide feedback on the proposed approach, discuss evaluation strategies, and provide infrastructure support. During this contract, we provided substantial infrastructure support to UW researchers, such as implementing and exposing new API routines that enable editing control-flow graphs (CFGs) to support the needs of partial evaluation. Near the end of the contract, UW researchers demonstrated the first steps of the end-to-end partial-evaluation prototype (operating on small programs but already capable of handling multiple procedures). While a number of limitations need to be addressed, we are happy to see our joint effort reach this milestone.

4.4 Evaluation infrastructure

An important aspect of developing a successful LACI prototype was to establish a common evaluation approach that could track the success and performance gains of applying optimizations to an executable. In January 2013, a testing and performance evaluation framework was built to establish a baseline and track progress on the development of the prototype. This framework has been continuously improved. Each test case tests the combination of a transformation applied to a target executable. The test cases are run as part of automated nightly testing and the test results are recorded using existing GrammaTech test infrastructure. The nightly tests enable us to track progress and also alert us to changes in the GrammaTech code analysis infrastructure caused by other projects; these are often improvements that aid the LACI project also, but sometimes have unintended bad side-effects – nightly testing enables us to catch such problems early.

The current tests apply a transformation to an executable, optionally performing IR validation. If the transformation is successfully applied (i.e., a new executable is written), then user-defined tests are run against each version of the executable (pre- and post-transformation). Any discrepancies in test results are reported as a failure. Additionally, system-resource utilization metrics are collected for each test run on each version of the executable and average changes in utilization profiles are reported. These metrics were

[1] Aung, M., Horwitz, S., Joiner, R., and Reps, T., Specialization slicing. TR-1776, Computer Sciences Department, University of Wisconsin, Madison, WI, October 2012. Submitted for journal publication.

Data Subject to Restrictions on Cover Page.

expanded as project objectives were refined by initial investigations, but they established a solid baseline upon which general gains in performance, size and security were tracked as transformations were developed. By the end of the contract, the test infrastructure was populated with small sample programs that test the soundness of the IR being generated and with a set of real-world executables. The test suite consists of these:

- Approximately 100 coreutils programs and their regression suites
- bzip2, libpng, a disassembler, and (in progress) cc1 from the GCC compiler
- rewriting regression suite (see below)

As part of our efforts to solidify our test infrastructure prior to the end of the contract, we also incorporated the testing of an existing regression suite for rewriting. The primary micro-benchmark regression suite for rewriting that had been developed earlier relied on using DVT (Disassembly Validation Tool), which utilized source information. This suite continues to be fragile across platforms. We generalized this benchmark suite to work without DVT, and found that by and large the tests in the suite performed better than with DVT. This stems in large part from improvements to the rewriting engine made under this contract. However, one of the tests, designed explicitly to check for the problematic "nonzero-based array access" problem, fails in the absence of DVT, as expected. This is a known shortcoming of our system, which stems from the fundamental undecidability of disassembly: absent source or debugging assistance, there is no general solution.

Additionally, we experimented with resurrecting an old test for rewriting the core of GCC (the GNU Compiler Collection). Specifically, we focused on the cc1 executable, which embodies the C frontend and the code generator). This test has not been repeated for a number of years. In prior experiments, we succeeded in rewriting an old version of the cc1 executable for cygwin by relying on some source-code hints (provided by DVT). The numerous fixes to rewriting that took place during this contract gave us optimism that we may be able to repeat the experiments without relying on DVT. We have created the infrastructure to build, rewrite, and test more modern versions of gcc on its more natural platform (Linux). gcc is a highly complex system and includes an extensive test suite that gets executed by our test infrastructure in order to confirm the successful rewriting. We have not yet succeeded in rewriting the newer cc1 executable. We hope to continue this ambitious project in the future.

Even beyond this contract, we are adding additional programs as we find suitable candidates. The key criteria for including a program are the availability of (1) openly available source so that we can build and test it in a variety of environments and (2) a reasonably strong test suite (because it would be a substantial tax on our resources to create such test suites ourselves). One attractive candidate is the Software-artifact Infrastructure Repository (SIR), maintained by a prominent group of software-engineering researchers and housed at the University of Nebraska-Lincoln. SIR contains a number of applications with extensive test suites. We expect to add this to our test suite in future work.

We considered the use of automated test-case generation via *concolic* execution as implemented in GrammaTech's Grace research tool. The application of Grace to coreutils packages is part of a different contract at GrammaTech. Although concolic testing is overall a

promising approach to test-case generation, we have decided that the current tools are not mature enough to assist us in the LACI project and that it would be a diversion from the core LACI goals to develop them ourselves.

5 Milestones

Interim results on multi-month tasks will be reported in the quarterly progress reports.

Milestone	Planned Start date	Planned Delivery/ Completion Date	Actual Delivery/ Completion Date
Kickoff meeting		As scheduled by Technical Monitor	Phone discussion in January 2013; TM declined to schedule a more in-depth discussion
Evaluation of structure and code quality of UW technology (task 1.1)	10/2012	11/30/2012	11/30/2012
First Quarterly report (task 5)		1/3/2013	1/7/2013
Investigate and implement dead-code removal of entire functions (task 3)	12/2012	3/31/2013	3/2013
Implement a testable working prototype with the null-transform option (the foundation for tasks 2 and 4)	12/2013	2/28/2013	2/2013
Continuing task: Identify failures resulting from incorrect IR; correspondingly improve or repair the IR recovery techniques. (tasks 1.2 and 2.1)	12/2012	9/24/2013, with all individual improvements noted in quarterly reports	Ongoing task with continuous improvements
Identify common coding idioms and compiler transformations that result in incorrect disassembly (task 2.1)	1/2012	2/15/2012	2/28/2012, with additional improvements as opportunities are identified
Implement a testing infrastructure (tasks 2.3 and 4)	1/2013	2/28/2012	2/2013
Design and implement the IR editing infrastructure (task 2).	1/2013	4/30/2013	5/2013; this includes additional APIs to support the new UW needs for partial evaluation
Evaluation of performance and	2/2013	3/31/2013	We completed an initial

Data Subject to Restrictions on Cover Page.

precision of UW technology (task 1.1)			review by 3/31. UW then began implementing a new approach, which is still in progress.
Develop real-world and synthetic benchmarks to evaluate performance (task 4).	2/2013	9/24/2013, with interim progress each month	2/2013 (infrastructure in place; adding additional tests is an ongoing task)
Investigate disassembly improvements such as learning-based bottom-up disassembly and all-leads disassembly (task 2.1)	3/2013	5/31/2013	Considered, but other improvements in disassembly provide adequate IR, making this task moot.
ONR Workshop on Optimization of Binaries		6/4/2013	6/4/2013
ONR Annual review		6/10-12/2013	6/10/2013
Investigate selective inlining of library functions (task 3.1)	3/2013	7/31/2013	7/31/2013
Second quarterly report (task 5)		4/3/2013	4/3/2013
Investigate finding and deleting functionally dead code, possibly using slicing and specialization (task 2.2 and 3.2).	4/2013	8/31/2013	7/31/2013 using PDG techniques. Additional analysis using specialization will be performed in follow-on work.
Investigate specialization to target platforms or target environments (task 3.2)	4/2013	8/31/2013	Postponed to follow-on work, because the UW work took a more substantive direction.
Implement aspects of the chosen disassembly extensions (task 2.1)	5/2013	8/31/2013	9/23/2013.
Evaluate hybrid analyses as a complement to static analyses for recovering IR (Task 1.2)	5/2013	8/31/2013	8/31/2013. Concolic execution was considered, but our evaluation was that it was not mature enough for this application.
Third quarterly report (task 5)		7/3/2013	7/11/2013
Measure the performance tradeoff of various optimizations and evaluate the	7/2013	9/24/2013	9/24/2013

Data Subject to Restrictions on Cover Page.

overall tool (task 2.3 and 4)			
Investigate options for handling DLLs (task 2.4)	8/2013	9/24/2013	9/24/2013
Final report (task 5)		10/24/2012 (contract end date)	9/24/2013

6 Issues requiring Government attention

None.

7 Future work

This project is a successful start on the longer-term goal of automated optimization of binary executables without benefit of source or user interaction. We are eager to follow this project with additional effort. A proposal for further work is currently under consideration by ONR. The goals of that proposal are summarized here.

- The primary goal of the next phase of work will be to further evaluate and implement program specialization as an optimization transformation for binary executables. This will include
    - Additional limit studies and studies of specific examples
    - Transferring technology from UW
    - Extending and improving the UW technology specifically for our application: optimizing binary executables
- An additional goal is to improve and extend the construction and use of GrammaTech’s Intermediate Representation of binary executables, specifically as improvement opportunities are found for the layer-collapsing application
- We will enlarge the test suite with additional test applications
- We will further investigate the security implications of the optimizations that we perform.

If an option on the proposal is funded, we will also attempt these additional tasks:

- Improving the soundness of IR construction
- Investigating tests of the soundness of transformations
- Investigating applying our techniques to cl-built Windows binaries

Data Subject to Restrictions on Cover Page.