REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
12/31/2016	Final Technical Report (Phase I - Base Period)	30-06-2014 - 31-12-2016

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

5a. CONTRACT NUMBER
N00014-14-C-0037

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Evan Driscoll
Tom Johnson

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
Arlington, Virginia 22203-1995

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

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

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Statement A.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Complex software is usually assembled from a number of third-party or in-house components and libraries. This development style makes writing software more tractable than starting from scratch, but this process has drawbacks. Very often the components are included in whole, but only used in part (increasing attack surfaces and bloat), or may include redundant error checks and other tests (increasing overhead). LACI (Layer Collapsing Infrastructure) uses binary-to-binary transformations to optimize compiled program executables to improve security and runtime performance, as well as reduce executable size. LACI allows its users to optimize, harden, and specialize existing binaries.

15. SUBJECT TERMS
binary analysis, binary optimization, binary rewriting, partial evaluation, program hardening

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Tom Johnson
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	30	(607) 273-7340 x.134

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

Final Report

(Report Period: 06/30/2014 to 12/31/2016)

Date of Publication: December 31, 2016

Sponsored by Office of Naval Research (ONR)

Contract No. N00014-14-C-0037
Effective Date of Contract: 06/30/2014

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

Submitted by:

Principal Investigator: Thomas Johnson
531 Esty Street
Ithaca, NY 14850-4201
(607) 273-7340 x. 134
tjohnson@grammatech.com

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

Contents

1 Summary

In this project, we investigated using binary-to-binary transformations to optimize compiled executables to improve security, improve runtime performance, and reduce executable size. The primary focus of our effort was on improving a research prototype of a partial evaluation engine original developed by researchers and colleagues at the University of Wisconsin (UW). We also carried out additional studies to determine how much room for improvement there is. We call this project LACI (Layer Collapsing Infrastructure).

Background

Our goal is to improve performance and reduce size of compiled binaries. These are useful goals themselves, but a significant benefit is improved security properties.

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 in-house components and libraries. This development style makes writing complex software much more tractable than if everything had to be written from scratch, but there are drawbacks:

- Modern software uses only a fraction of the code of its integrated components. Generic components include code that is "dead" or irrelevant for the particular client in question.
- Modern software must penetrate multiple layers of libraries to get to the required low-level functionality. Each layer implements its own, potentially redundant, sanity checks and environment customizations.
- Often, source code for libraries and components is not available, which prevents compilers and linkers from being able to optimize across library levels.
- In a nutshell, heavy use of libraries leads to code bloat. Code bloat slows application loading, reduces available memory, and makes software less robust and more vulnerable.

In addition to performance and size, "software bloat" is a security concern. At a low level, bloat means a larger code base for attackers to draw from for attacks such as return-to-library and return-oriented-programming attacks; this is true even if the extra code can never be called by the program. At a higher level, libraries are written to be generically useful, and usually include functionality unneeded by specific applications. For example, one historically-common exploit is based on abusing the `%n` format specifier for functions such as `printf`, despite few programs actually using that feature. If a program were rewritten to use a version of `printf` that does not support `%n`, that would close off an attack vector.

The opportunity

We believe that this problem will best be addressed by optimizing the software executable at or

immediately prior to deployment of software. Machine-code analysis and binary-rewriting techniques have reached a sufficient maturity level to make whole-program, machine-code optimization feasible. Thus, we believe there is now a great opportunity to build tools that will revolutionize the software development industry. In this project, we have worked on a tool that aggressively optimizes software by leveraging a global view of all the code used in the program, including the integrated components and libraries. The tool attempts to optimize software in the following ways:

- Specializing integrated components and libraries to eliminate library functionality that is not required by the software. Collapsing interface layers of nested components and libraries.
- Specializing software (libraries and included components) for the host platform to make the code leaner and more efficient for the host platform.
- Performing traditional whole-program optimizations to improve software efficiency.

We worked on a tool that operates for three classes of users:

- Developers, dedicated security analysists, and reverse engineers can use it to operate at the level of procedures and instructions: optimizing, removing, and/or customizing the behavior at a function level.
- End users can pick out parts of a program (e.g., certain command line options) that either they are interested in using, and rewrite programs to remove uninteresting features.
- System administrators can specialize programs for a particular configuration and setting.

Really these three modes of operation are different sides of the same coin, but we illustrate each of these modes with experiments. The tool operates directly on an executable, using binary rewriting to eliminate dead code and perform optimization. This will allow the tool to effectively handle commercial libraries and components for which the source code is not available. Also, it will allow users to apply the tool immediately prior to deployment, when details of the target platform are available, letting the tool tailor the software specifically to the target platform.

2 Introduction

In this section, we give some background information on the following items:

- §2.1 covers partial evaluation, the technique that forms the backbone of our investigations. An integral part of partial evaluation is an analysis called *binding-time analysis*, which has several design axes discussed in §2.1.2. For a specific partial evaluation implementation, we are building off of a research prototype developed by the University of Wisconsin, discussed in §2.1.4.
- For limit studies of how much benefit we might see from partial evaluation, we carried out investigation using an analysis called value-set analysis (VSA) with a *trace seeding* feature we developed under DARPA's VET program. VSA and trace seeding is described in §2.2.
- Finally, we review some work done during Phase I of LACI (§2.3).

2.1 Partial Evaluation, Binding-Time Analysis, and Program Specialization

Partial evaluation is a set of program transformation techniques that trim and rewrite a program, or portion of a program, according to the specific context(s) it is used in. As a simple example,

consider an application in which a particular function is always called with one of its arguments having the same constant value; the function can be specialized given that knowledge. The common compiler optimizations of constant propagation and constant folding can be seen as a very limited form of partial evaluation. At the level of a whole program, one might know *some* of the inputs to the program (e.g., configuration files) but not know other inputs; a partial evaluator can create a version of the program specialized to the known inputs.

The process of partial evaluation works in two steps, detailed below. First, the program determines what portions of the program depend on unknown input in a process called *binding-time analysis* (BTA). Second, the program is rewritten to take advantage of the known information.

Parts of the target's (the program's or program portion's) starting state and inputs are known and some are unknown. The known parts are called *static*, and the unknown parts are called *dynamic*. Either before or during the specialization process proper (see §2.1.2.1), information about what parts of the program are static and dynamic need to be propagated through the program. If "something" depends on dynamic input, then that something must be marked dynamic as well. That something might be another variable, an expression, or syntax such as control flow constructs. For example:

- If the program has `x := y` and `y` is dynamic, then `x` must be dynamic as well (at least for the live range starting with that definition).
- If `y` is dynamic, then the expression `x + y` is dynamic.
- If the program has `if (e) ...` and `e` is dynamic, then the `if` itself is dynamic. We may also mark program points in both branches of the `if` as dynamic, because the dynamic condition controls whether they are executed; see §2.1.4.

The process of determining which parts of the program are dynamic is called "binding-time analysis" (BTA), and there are several design choices that will affect the precision of results and performance of the analysis (see the following sections). The actual result of BTA is called a *division*; a division annotates each program element with "static" or "dynamic."

Once the division is known, the partial evaluator can perform *program specialization*. In its simplest form, a program specializer operates similarly to a fancy interpreter. A normal interpreter tracks a program state (a map of variables to values), applies that state to the current statement, and the result of the application is the next state, including the next statement. The specializer in a partial evaluator includes the following changes:

- The program state only tracks values of *static* variables.
- When the specializer is interpreting a program element marked *static*, it acts as a normal interpreter. (Something static can only depend on static variables, so the restricted program state is sufficient.)
- When the specializer is "interpreting" a program element marked *dynamic*, instead of doing a normal interpretation, it instead outputs that program element to the final, specialized program. For example, if an `if` statement is marked dynamic, the specializer will output that `if` statement.
- The specializer may have *multiple* next statements. For example, if an `if` statement is marked dynamic, the specializer needs to interpret both branches.

The outputting of dynamic elements in the third step above is called *residuation*, and the result of

specialization is the *residual program*.

2.1.1 Example of partial evaluation

In this section, we will provide a couple examples of partial evaluation. We will start with a simple example, adapted from a tutorial by Hatcliff .

```
int pow(int base, int exp)
{
    int result = 1;
    while (exp > 1) {
     result = result * base;
     exp--;
    }
    return result;
}
```

Suppose we want a version of the `pow` function shown above that is specialized to taking the cube of `base`. We can partially evaluate `pow` specifying `base` as dynamic and `exp` as static, equal to 3. We get the following result:

```
int pow_exp3(int base)
{
    int result = 1 * base;
    result = result * base;
    result = result * base;
    return result;
}
```

Effectively, what the partial evaluator has done in this case is unroll the loop three times. Note that the partial evaluator did not simplify `1*base`; this is traditionally left to the compiler that will build the residual program.

However, one must be careful when designing and using partial evaluators, because it is possible to create a specializer that will not terminate on some programs. (See §2.1.3 for a more detailed discussion.) With a naïve partial evaluator, we could not ask the opposite question – specialize `pow` with a static `base` and dynamic `exp`. This would lead to non-termination, as the partial evaluator would attempt to produce the following infinite program (with `base` = 5):

```
int pow_base5(int exp)
{
    int result = 1 * 5;
    if (exp == 1)
     return result;

    result *= 5;
    exp--;
    if (exp == 1)
     return result;

    result *= 5;
    exp--;
    if (exp == 1)
```

```
      return result;

    result *= 5;
    exp--;
    if (exp == 1)
      return result;
    ...
}
```

The partial evaluator continues unrolling the loop, but it does not know when to stop. Using either the manually-added `gen` annotation suggested by Hatcliff or the control-dependence-based technique that is used by our partial evaluator, the result would be:

```
int pow_base5(int exp)
{
    int result = 1;
    while (exp > 1) {
      result = result * 5;
      exp--;
    }
    return result;
}
```

This is only a small improvement over the original, but it *is* an improvement and we could hardly expect to do better without duplicating the loop-unrolling heuristics that compilers use to determine *how much* to unroll a loop.

2.1.2 Binding time analyses

Critical to the success of partial evaluation is the precision of the static/dynamic division; the division is computed by *binding-time analysis* (BTA). BTA determines the division, and the division determines what code is evaluated statically versus what code is residuated; thus, BTA directly affects how the original program is transformed.

There are several variations on BTA. This section will describe some of the important choices and their tradeoffs.

2.1.2.1 Online vs offline

The first choice is whether the binding-time analysis should be *online* or *offline*.[1] Online BTA occurs intertwined with specialization. Offline BTA is a ahead-of-time analysis that annotates the program with static/dynamic annotations; specialization proceeds obeying these annotations.

Another way of looking at the distinction is to ask whether the BTA has access to the *values* of the static inputs; or whether it knows just *which* inputs are static, and only the specialization process will know their values. Online BTA has access to the values, while offline BTA does

[1] Formally, "online BTA" is an abuse of terminology, and "BTA" is, by definition, a separate analysis; we should instead be referring to online and offline specialization, where offline *specialization* is paired with a BTA and online specialization has no need for a BTA. However, "online BTA" emphasizes that it is the placement of the computation of the static/dynamic division that is being changed; like a "dynamically-typed language" (despite a type system being, by strict definition, a static thing), it is a useful abuse.

not.

As a simple example of where this can make a difference, consider the following example:

```
int foo(bool cond /*static*/, int if_false /*dynamic*/) {
    return (cond ? 0 : if_false) * 5;
}
```

An offline BTA cannot make much progress with this program directly. Because `if_false` is dynamic, the whole conditional expression must be marked dynamic; as a result, the multiplication by five will be dynamic as well.

Suppose at specialization time, the value of `cond` is provided to be true. In this case, the value of the conditional expression *is* known statically – 0 – but the specializer still has to residuate `return 0*5` because the multiplication was marked as dynamic by (offline) BTA and the specializer is just following BTA's annotations. (In this case, further compiler-style optimization may well constant fold the multiplication, but that is not true in general.)

An *online* BTA can "do the right thing" in this case. Because the BTA is woven into specialization:

1. The "BTA" would determine that the condition is static.
2. The specializer would start to interpret the conditional and reduce it to the true branch.
3. The BTA would determine that the true branch's expression is static.
4. The specializer would evaluate the true branch to 0, which becomes the result of the conditional expression.
5. The BTA would determine that both arguments to the `*` are static.
6. The specializer would evaluate the multiplication to get 0.

The final result for an online BTA would be simply `return 0`, because the BTA isn't pre-determining the static/dynamic division before it even executes.

There are a variety of techniques, collectively known as *binding-time improvements* [1], for trying to overcome this problem. In our example above, we could distribute multiplication over the conditional to get `return (cond ? 0 * 5 : if_false * 5)`. The `0*5` portion is entirely static and would be determined as such by offline BTA, allowing it to achieve the same result as online BTA. However, binding-time improvements can be seen in part as a compensation for not running online BTA, and they are not general.

2.1.2.2 Offline: uniform vs pointwise

The first refinement of offline BTA is whether the set of variables marked dynamic is allowed to vary by program point. If the static and dynamic variables are fixed everywhere, then the BTA is *uniform*; if each point can have a separate set, then the BTA is *pointwise*. (Online BTAs are pointwise by their nature.)

For example, consider a typical `temp=b; b=a; a=temp;` sequence to swap two values. With a pointwise BTA, this sequence will swap `a`'s and `b`'s static/dynamic bindings just like it swaps their values. However, with a uniform BTA, if either variable is dynamic, then both variables must be dynamic for the entire function or program.

2.1.2.3 Offline: monovariant vs polyvariant

The final refinement we will examine is whether the BTA allows only one static/dynamic division at each program point, or multiple. If the BTA only allows one, then it is *monovariant*; if it allows multiple, it is *polyvariant*. (Again, online BTAs can be considered polyvariant.)

One context the distinction commonly arises (though not the only one) is when there are multiple call sites to the same function with different sets of static and dynamic parameters at each site. As a trivial example, consider the code fragment `foo(0, x); foo(x, 0);` with `x` dynamic. A monovariant BTA is only allowed to produce one annotation for `foo`, and so must conservatively annotate both parameters as dynamic. As a result, presumably most of the function `foo` will be dynamic. However, a polyvariant BTA would be able to produce two separate annotations for `foo` – one annotation would be (static, dynamic) and be associated with the call site `foo(0, x)`, and a second (dynamic, static) annotation would be associated with `foo(x, 0)`.

For an example of where polyvariant BTAs can be useful aside from function calls, consider the following function:

```
int foo(int a, int b)
{
    if (b > a) {
     a = b;
    }
    return bar(a, a);
}
```

Suppose we want to specialize `max` with `a` dynamic and `b` static, equaling 5. There is one path through `max` (if `a` is the larger) where `a` has its original, dynamic value at the return. In the other path, `a` has the value 5 at the return. The specializer cannot tell which path will be taken, but a polyvariant BTA and accompanying specializer could produce two copies of the return statement, one for each path. That would lead to the following specialized version:

```
int foo_5_poly(int a)
{
    if (5 > a) {
     return bar(5);
    }
    else {
     return bar(a);
    }
}
```

(The specializer would likely include a dead `a = 5` assignment in the conditional, but it would be eliminated by the compiler.) Note that the specializer may be able to simplify `bar(5)` even more, making the true case even faster; this contrasts with the following version, where both cases need the general version of `bar`.

However, a monovariant BTA and specializer would only be able to produce one return statement. Because `a` is dynamic at the return on *some* paths, it would need to be dynamic at that one point. In this example, we would just replace the uses of `b` in the original procedure by `5`.

```
int max_5_mono(int a)
{
    if (5 > a) {
     a = 5;
    }
    return a;
}
```

In particular, the assignment to `a` inside the `if` is still there. (And in contrast to the prior case, it is not dead, so it cannot be trivially removed by the compiler.)

In both examples, the polyvariant BTA is more precise, but at the cost of (i) (perhaps) many more annotations and (ii) (perhaps) many more *specializations* as the specializer will need to create a separate residual program point for each annotation.

2.1.2.4 Tradeoffs between BTA choices

There are clear performance tradeoffs between the different options for binding-time analyses. In a rough order, one can expect that online BTA will be the most expensive, followed by polyvariant BTAs, followed by monovariant pointwise BTAs, with monovariant uniform BTAs the fastest.

What is likely less evident is that the more expensive and precise BTAs are not *necessarily* better.

Consider the following examples, adapted from [11].

		Partial evaluation result	
Source	*Inputs*	*Uniform BTA*	*Polyvariant BTA*

`if (upd)`
`  val = curval;`
 	
`output = val *`
`2;`
`return output;`
	`upd = false;`
	`val = 100;`
	`outval = 0;`
	
	`curval dynamic`
		`val = 100;`
		`outval = val * 2;`
		`return outval;`
			`return 200;`

`while (a > 0)`
`    p(x)`
 	
`procedure p(x) {`
`    a -= 1;`
`    b += y;`
`    count += 1;`
`}`
	`a = 2;`
	`b = 5;`
	
	`x dynamic`
	`count dynamic`
		`b = 5;`
		`p(x);`
		`p(x);`
		
		`procedure p(x) {`
		`    b += y;`
		`    count += 1;`
		`}`
			`p1(x);`
			`p2(x);`
			
			`procedure p1(x) {`
			`    b = 5 + y;`
			`    count += 1;`
			`}`
			
			`procedure p2(x) {`
			`    b += y;`
			`    count += 1;`
			`}`

For the first example, the version specialized by a polyvariant partial evaluator is clearly better. However, the second example is more ambiguous. The version created with a uniform BTA executes one more instruction at runtime (the initial assignment to `b`), but the polyvariant version is significantly larger. Christensen et al. argue that the uniform specialization is better in

circumstances such as this; in real programs, the duplication can be substantial. When run on real machines, smaller programs are likely to have better cache behavior and may run faster even if they execute more instructions. It is also possible for switching from a less-precise to a more-precise BTA to cause a specializer to cease terminating entirely (see the next section for more on termination of the partial evaluator.)

2.1.3 Termination of partial evaluators

Another place that care must be taken is in ensuring termination of the specializer. Just as a normal interpreter can run forever, so can a naïve specializer. (As a quick illustration, consider providing a program you want to run, a specification that *all* inputs are static, and the inputs you want to run it with. By the description in §2.1, the specializer should do exactly what the original program does on those inputs, which could be running forever.)

Even if the specializer is not in a true infinite loop, it may run for a very long time in a loop with a high static bound. What is perhaps even worse is that, if there are dynamic elements inside that loop, the partial evaluator will residuate those elements each time through the loop. Effectively, when a loop has a static bound, the specializer will blindly unroll that loop for the appropriate number of iterations. (Contrast this behavior with a compiler-style loop unrolling optimization, which uses models of the cost of the control structure, loop body size, number of unrollings, and other factors to determine whether and how much to unroll a loop.)

Various attempts at partial specialization take different approaches to address this issue.

The dead-simple solution is to ignore the problem, and to put the responsibility for dealing with this on the programmer. It is possible, the argument goes, for a developer to write a program that runs too long or infinitely, and the compiler will not go out of its way to prevent this; why shouldn't the developer be able to write a program that specializes forever? Why should a partial evaluator have to jump through hoops to catch this case? To *allow* the developer to have control, such partial evaluators usually provide a means to explicitly mark an expression or variable as dynamic. For example, the developer could mark the variable tracking loop iterations as dynamic, and then the specializer would not do unrolling. This approach is what is taken by Shali and Cook [3], for example.

A more sophisticated approach [1] tries to ensure that each loop has a bounded number of iterations once you fix the static inputs. If the BTA can find a loop induction variable that it proves always decreases by some measure that cannot decrease indefinitely, then it assumes that loop is well-behaved. If it can *not* find such a variable, than any other variable changed in the loop (at least in a way that increases its "size") is marked dynamic. This ensures termination, though could still lead to large residual programs if the loop has a high bound.

Neither of these is particularly applicable to a partial evaluator that is supposed to work on arbitrary, stripped binaries. The first approach requires manual annotation – likely, a *lot* of manual annotation – and the second requires analyses that are extremely difficult on binary programs. The Wisconsin binary partial evaluator WIPER, the evaluator that we investigated in Phase II (see §2.1.4) uses a different, coarser approximation to ensure termination.

WIPER's BTA performs a forward slice from the dynamic inputs; anything included in that slice is considered dynamic. The forward slice follows control-dependence edges; this means that if a dynamic instruction is control dependent on a particular branch *B*, then *B* will also be marked dynamic. Effectively, this marks a loop test as dynamic if anything in the body is dynamic.

Implications of this decision will be discussed later (§4.4.1).

2.1.4 WIPER: Binary partial evaluator from the University of Wisconsin

During Phase II, we investigated applying a binary partial evaluation tool called WIPER to layer collapsing. WIPER was developed by Venkatesh Srinivasan and Thomas Reps at the University of Wisconsin, and is the only partial evaluator for binary code that we are collectively aware of.

WIPER operates in a typical offline manner. As stated above, its binding-time analysis is the result of a forward slice from the variables and instructions that are designated as dynamic (the "input"). It is a pointwise, monovariant BTA. The forward slice leverages additional recent work from UW on *specialization slicing* [4].

The specialization portion of WIPER is unique. Many x86 instructions (indeed, for other architectures as well) conflate updates to several different locations, for example two different registers. To maintain precision, it is important to decouple these effects; then "part" of an instruction can be marked static, part of an instruction can be marked dynamic, and the specializer can interpret the static part and residuate the dynamic part. However, there is no such thing as emitting "part of an instruction," and thus WIPER must find a sequence of instructions that has the desired effect and *just* the desired effect.

For example, suppose the input instruction is `push eax`. This instruction performs two actions:

- Stores the value of register `eax` into memory at the stack pointer (`esp`). In C-like notation, this action is `(*esp) = eax`.
- Decrements the stack pointer (the stack grows toward lower addresses). In C-like notation, this action is `esp -= 4`.

The second action writes `esp`. Suppose there is a later instruction `mov ebx, [esp + 16]`. (In C-like notation, this is `ebx = *(esp + 16)`.) At surface level, because `push eax` writes to `esp` and `mov ebx, [esp + 16]` reads `esp`, the `mov` instruction is data-dependent on the `push`; this would mean that if `eax` is dynamic at the `push`, it would mean the `mov` must be dynamic too. This is highly undesirable, and is also conceptually unnecessary – considering the push's two actions separately, it is clear that the `mov` does *not*, in fact, depend on `ebx` and should thus not be made dynamic.

Because `push` and `pop` instructions are extremely common, and because local variable accesses all take the form of offsets from `esp` (or `ebp`, copied from `esp`), conflating the two actions will result in marking basically the whole program as dynamic. BTA needs to be able to separate the two actions, so that the specializer can residuate the first action (assuming a dynamic `eax`) and statically interpret the second action.

To determine the instructions that need to be residuated, WIPER uses *instruction synthesis* [13], the process of generating instructions that meet a logical formula describing the desired behavior. To use the above example, WIPER proceeds as follows:

- Produce a logical formula for the semantics of the instruction in question. For `push eax`, the formula will be `(mem' = mem[esp - 4 ↦ eax]) ∧ (esp′ = esp - 4) ∧ …`. Note that the two conjuncts of this formula correspond to the two actions of `push eax`. (The "primed" variables indicate the value following the instruction's execution. `mem` is a logical function symbol representing the values in memory. There are also many conjuncts such as `eax' =`

`eax` and `ebx' = ebx` that indicate that other parts of the state do not change.)

- Separate the updates along interesting boundaries; WIPER considers an interesting boundary to be the updates to the stack pointer, versus everything else. We pull out the `esp' = esp - 4` conjunct from the formula, replacing it with just `esp' = esp`.
- Use instruction synthesis to generate an instruction or instruction sequence with the new formula's semantics. In our example, that will be an instruction that performs just the first action, corresponding to `mem' = mem[esp - 4 ↦ eax]` and leaving the rest of the state unchanged – `mov [esp-4], eax`.
- Consider the synthesized instruction(s) along with another instruction that just updates the stack pointer when performing BTA and specialization. In our example, the stack pointer is updated by `lea esp, [esp-4]`; we use `lea` instead of `sub esp, 4` to avoid updating flags. If `eax` is dynamic and `esp` is static, WIPER will residuate `mov [esp-4], eax` and statically interpret `lea esp, [esp-4]`.

The stack pointer, `esp`, is treated specially to allow access to local variables to be resolved statically. It is considered static at program entry, and will become dynamic only in rare cases (while inside functions that use `alloca` or similar), so we expect that nearly all places there is an instruction that conflates actions that update the stack pointer with something else will benefit from this technique.

Experiments performed on WIPER by UW showed promising results, showing an average reduction of about 25% in runtime across a small suite of test cases, comparing a specialized version to the original. We reproduced these results for a subset of those tests; reductions in runtime ranged between 4% and 55% in our runs:

		Average test run time (sec)		
test case	*program size (kb)*	*Original*	*Rewritten*	*Reduction*
interpreter	2.35	15,616	14,159	9%
sha1	3.8	5,810	3,684	37%
filter	3.7	4,047	1,822	55%
dotproduct	2.3	2,580	1,789	31%
power	1.7	1,411	1,357	4%

For the above experiments, both by us and UW, the program points that receive input were identified and marked manually; these served as the dynamic inputs. This would not be a feasible approach in real-world use; §3.2 discusses what we investigated in the context of LACI.

2.2 Value-Set Analysis

Value-set analysis (VSA) is a combined numeric- and pointer-analysis algorithm that determines an approximation of the set of numeric values and addresses that each register and memory location can hold for each program point [5]. In particular, at each instruction that contains an indirect memory operand, VSA provides information about the contents of the registers that are used. This information permits it to determine the (abstract) addresses that are potentially accessed, which, in turn, permits it to determine the potential effects of the instruction on the program state.

A key feature of VSA is that it tracks integer-valued and address-valued quantities simultaneously. This approach is crucial for analyzing executables because numeric operations and address-dereference operations are inextricably intertwined even in the machine code generated for simple source-code operations.

VSA is a *soundy* analysis [7], which means that the approximations it computes are usually-safe overapproximations of the truth (if the approximations were always safe it would be straight-up *sound*), but in order to get useful results it is sometimes necessary for VSA to operate unsoundly. For example, suppose there is a memory write (e.g., the machine-code analogue of `*p = 42`) but VSA has lost enough precision that it has no information about what the target address (`p`) refers to. In this case, the *sound* action to take would be to clobber all of memory; but this leads to extremely imprecise analysis results that would be nearly useless. Instead, in the mode we usually run VSA in, VSA will ignore the write. This means that whatever the target actually is may become *underapproximated*.

VSA works as a pretty typical dataflow analysis. For each program point, it stores an abstract state that is a map from program variables to abstract values. (Below, the abstract state for each node is referred to as the "aggregated" abstract state.) The structure of the values is described by [6].[2] Abstract values are propagated around the CFG as follows, using a worklist of nodes:

- VSA pops a new current node from the worklist.
- VSA grabs the abstract state of the current node, and interprets the semantics of the current node under the abstract state. The result is the abstract state "after" the node executes.
- VSA propagates the "after" state to each successor of the current state, unless they appear to not be reachable:
    - VSA grabs the abstract state of the successor, and joins it to the after state of the current node.
    - If the join result is different from the successor's origial abstract state, it replaces the successor's abstract state and the successor is added to the worklist.

As part of DARPA's VET project, we introduced a new mode to VSA, which we call VSA with *trace seeding*. The idea behind this mode is to provide VSA with extra information from a real, dynamic trace of the target program. The trace includes each instruction that is executed, along with the value(s) that are read or written by that instruction. It also includes write information for some system calls, such as `read`. (Dynamic values can, of course, usually be reconstructed from the trace and information about the effects of system calls. However, we include this "redundant" information in the trace to provide some protection against any unmodeled effects.)

When running with trace seeding, VSA has two phases. The second phase is just standard VSA, except that instead of starting from `main` it starts from the last instruction of the trace (which is also where the first phase leaves off).

The first phase of VSA with trace seeding operates as follows. Compared to standard VSA:

- Instead of propagating information to every (apparently-reachable) successor of the current instruction, VSA only propagates to the next instruction *in the trace*.

[2] Briefly, VSA abstract values consist of a set of abstract memory regions the value may point to plus a "strided interval" of offsets in that region, along with a strided interval of possible numberic values. A strided interval is a range *s[l,h]*, and its concretization is *{l, l+s, l+2s, …, h}*.

- In addition to an aggregated abstract state for each instruction, VSA also tracks a *singleton* "current state." When it evaluates a node, it does so under the current state rather than the node's aggregate state. Evaluating a singleton state gives another singleton state, which is the next current state. The next current state is then joined with the node's aggregate state.
- VSA cross checks the values that it *thinks* the program should have, as represented in the singleton current state, with the values that were *actually observed* and captured in the trace. If they differ, VSA updates its representation to match the trace's reality and outputs a diagnostic message.

VSA does this for each instruction in the trace. When it reaches the end of the trace, it switches to phase 2.

The meaning of the result of running VSA with trace seeding is sort of a *conditional* static analysis: it represents a soundy approximation of all runs of the program *that follow that initial trace*. In our application, that means "runs that read the given configuration file." For something like a web server where a sysadmin can set the configuration file and it is read in before any user requests are serviced, the user cannot affect the initial trace and this is a useful question to ask.

2.3 Review of some Phase I results

During Phase I, we primarily explored two aspects of the problem of binary optimizatio. First, we implemented some hand-written, special-purpose program transformations, such as restricted versions of dead code elimination and function inlining, and measured their improvements (§2.3.1). Second, we performed some initial limit studies to see the potential for partial evaluation to help (§2.3.2). These limit studies indicated promise and inspired us to propose partial evaluation as the center of our Phase II work.

In addition to the results covered below, we also collected a large test suite to subject to our transformations. More details can be found in the Phase I final report.

2.3.1 Phase I Transformation and Results

We implemented four program transformations during Phase I. These transformations are based on GrammaTech's binary rewriting infrastructure, developed over the course of many different projects. Rewriting works by reading the binary and building an intermediate representation (IR) for it, recovering information such as procedures and variables; changing the IR; and then emitting the IR as an assembly file that can be built into the transformed program. We call this the *melt, stir, refreeze* approach. The transformations we implemented are:

- The null "transformation." This is not really a transformation; it serves as both a test of the transformation infrastructure and a performance baseline. (Though there is no transformation, the program still goes through the melt and refreeze steps of our rewriter.)
- Dead code removal.
- Procedure inlining.
- Converting dynamic libraries to static libraries.

These showed good results and promise for a binary optimizer; see the Phase I final report for further details.

2.3.2 Phase I Partial Evaluation Limit Studies

We also looked at opportunities for partial evaluation. At this point, the partial evaluator was not

in a state ready to be run, so these studies were somewhat informal.

First, we performed quantitive studies of how often functions are called with constant arguments, and how many (and what) those arguments are. These are candidates for specialization, because the constant argument could be specialized into the called function. For example, we found the function `gnu_mbswidth` was called six times by the `dir` executable, always passing 0 as the second argument. This provides a prime target for specialization, because there would only need to be one copy of `gnu_mbswidth` in the final executable.

Second, we manually looked for individual examples that would indicate the promise of partial evaluation. We identified several such examples. These are covered in more detail in the Phase I final report, but we briefly review them below:

- `fnmatch` is a function that does generic wildcard matching (e.g. `*` patterns in filenames). A flags parameter controls what style of matching is done, but this parameter is almost always constant. A program using this function could be specialized to produce an optimized version of `fnmatch` that does not check that flags parameter, or even need to be passed it.
- `fts_open` is a function that traverses a file system. Similar to `fnmatch`, it takes a flags parameter that is almost always constant.
- `divdi3` and `moddi3` implement 64-bit division and modulus on 32-bit architectures. Both functions check for zero in the divisor, but in many calling contexts it would be possible to determine that a zero divisor is impossible. In fact, in our test suite, the divisor was usually a non-zero constant. This example is a bit more complex than the others in that we are interested in a *property* of the parameter – "can this be zero" – rather than the exact value. A powerful specializer could eliminate the zero check even for a non-constant value, as long as it could determine that the divisor could not be zero.

3 Methods, Assumptions, and Procedures

This section discusses the experiments we performed under Phase II.

First, we carried out three limit studies to try to determine how much promise partial evaluation has in different situations.

Second, the bulk of our Phase II effort was spent on partial evaluation. We made many robustness and performance improvements, and extended it to be able to partially-evaluate portions of a program. (§3.2.1)

3.1 Limit studies

First, we carried out several limit studies. The first limit study are interesting from the perspective of the developer – they concern low-level implementation details of the program.

- We think that, for many programs, we could completely eliminate the chance of a historically-common vulnerability: format string vulnerabilities (e.g., for `printf`). Format string exploits are based around abusing the `%n` format specifier to write a controlled value to memory. If we could replace calls to `printf` with versions specialized to a constant format string, and/or replace `printf` entirely with a version that does not support `%n`, we could reduce or eliminate this attack vector. However, this transformation would depend on programs not actually needing support for `%n`. We think this is the case, and set out to establish this hypothesis. (§3.1.1)

The next limit study is from the perspective of a user:

- We hypothesized that, for many programs, a user is often only interested in a small portion of that program's features. If support for the unneeded features were removed, that would leave the program smaller, faster, and more secure. We wanted to determine how much of a program is actually used in typical use. (§3.1.2)

And finally, we did a limit study from the perspective of a system administrator:

- Many programs, particularly servers, are highly configurable. However, a sysadmin often knows what configuration they want to run the server in, and do not regularly change that configuration. If we could specialize the program with respect to a configuration, we could again eliminate unused code, leaving the program smaller, faster, and more secure. We used VSA as a proxy for how much room there is for partial evaluation to make these improvements. (§3.1.3)

3.1.1 Improving the security of format-string functions

Hypothesis: most programs do not need the `%n` format specifier[3] to functions such as `printf`, and the standard version could usually be replaced by a hardened version that does not support `%n`.

One specialization idea that we hypothesized would be very useful at reducing attack surface is to create specialized versions of formatted I/O functions, such as `printf` and `scanf`. (This section also applies to other functions in the family, like `snprintf` and `fscanf`, and to a lesser extent other functions that operate in a similar way.) Such functions have been the culprit behind several format-string security vulnerabilities in the past [9]; Carlini et al. even describe "printf-oriented programming," proving that `printf` is Turing complete [8].

However, we hypothesized that the unsafe feature that allow these exploits, the `%n` format specifier to both `printf` and `scanf`, is not commonly used. Furthermore, in most cases the format string is a fixed, static argument. (In fact, GCC and Clang warn by default if a non-literal format string is passed as the sole argument to `printf`, and both offer `-Wformat-nonliteral` to warn when *any* non-literal is passed as the format string.)

Because of these characteristics of use, we think that it would be possible in a large proportion of the time to provide a specialized `printf`/etc. implementation that omits the unsafe operations, or perhaps even ignores at runtime the provided format string altogether, operating on a version of the function that is completely specialized to the particular format string.

This study illustrates how LACI can operate at a low level, customizing the behavior of individual functions and function calls, in the hands of a developer, security expert, or reverse engineer.

To investigate the feasibility and utility of this specialization, we performed a limit study examining how libraries and executables use format strings and how often `%n` is used.

For our corpus, we used files appearing within `/usr/bin`, `/usr/sbin`, and `/usr/lib` (including

[3] The %n format specifier instructs printf/scanf to store to a specified address memory the number of bytes output to/input from the stream. Format string vulnerabilities usually occur when an attacker can control the address written to along with some control of the size.

subdirectories) on the machine of an employee at GrammaTech. That machine runs the Xubuntu variant of Ubuntu Linux 16.04.1. In addition to the packages that are installed by default by the distribution, many more packages used for day-to-day use were present, as well as the following server packages specifically installed for this experiment: `apache2`, `lighttpd`, `nginx`, `postgresql`, and `isc-dhcp-server`.

We used the standard `strings` utility (part of GNU Binutils) to extract strings from each binary. For configurations, we told it to look for strings of length at least two characters (`-n2`; the default is four). When it is run on a binary, it has two modes for where in the file it looks: it can either look in the whole file (`-a`) or just initialized, loaded data sections (`-d`). "Data sections" includes the `.text` section that includes the actual program code, and it does not distinguish between instruction and actual *data*. (On some RISC architectures, such as ARM, read-only data is routinely intermixed with code; this is less common on x64, but still possible.) We performed initial investigations with both modes, but report most of our results with the `-d` mode.

The first thing we did was a general survey of the output of `strings`, just counting the number of strings and their properties – where in the file the strings occurred, whether or not they were duplicates, and whether or not they were safe or dangerous (containing `%n`). We first divided strings into those in the data sections (as determined by the `strings` utility itself) vs. outside of the data sections. We then divided each group into unique strings and duplicates of another string within the same location category. If a string appears multiple times, one of those times is counted in the unique category, the others in duplicates. So the number in the "unique" category is the number of unique strings in the location category. If string appears in both a data section as well as outside data sections, it will be counted as two unique strings; we did not determine if this happens. Finally, we divided each of those groups up into whether or not the strings contain `%n`. Those that do are classified as "dangerous" (really, *potentially* dangerous) and those that do not are "safe."

We then took a more file-centric viewpoint of where the `%n` occurrences are located. Using the "data sections" mode (we also investigated "all" and got nearly identical results), we looked at the number of files that contain a `%n` and do not contain `%n`, and divided those groups up by file system location (`bin` vs. `sbin` vs. `lib`).

Finally, we performed a more detailed investigation of some of the files containing `%n`. Many of the server executables are located in `/usr/sbin`, so we chose to look at files in that directory that contain `%n`. Our initial surveys did not exclude files that are not x86/x64 binaries, so we first excluded those. For a binary containing `%n`, there are three possibilities:

- The `%n` appears in a string literal passed as a format string (a "true positive" in terms of looking for code that would break if we disallowed `%n` format strings)
- The `%n` appears in a string literal used for other purposes (a false positive)
- The `%n` appears in something that is not a string literal at all (an even more falsy false positive)

To determine which is the case, we used CodeSurfer for Binaries to generate a disassembly listing;[4] Codesurfer and IDA Pro attempt to recover the locations of string literals, and in our

[4] For one program, `mysqld`, CodeSurfer crashed while trying to analyze; we used IDA Pro on its own to generate `mysqld`'s listing.

experience usually do a good job. We checked the disassembly to determine whether there is, in fact, a string containing `%n`, and how it was used. If there was not a string literal containing `%n`, we did a second check to find out why, looking at where the `%n` bytes appear. (In particular, we checked whether they appear in the bytes of an instruction.)

We think this study provides a good indication of how programs use format strings, but it is not perfect. Two potential problems are:

- The program dynamically constructs (or reads from a configuration file, etc.) a format string containing `%n`. In this case, what we would expect is to see a dynamic argument to a format-string function; neither a partial evaulator nor a specialized format-string simplification would be able to replace that call. The general version would need to remain as-is and, unfortunately, available to return-to-library attackers. However, other calls to the function would still be able to be replaced, which would still reduce the attack surface.
- When concluding that a `%n` occurrence is a false positive, we are trusting that the CodeSurfer and IDA Pro string-literal recovery is sufficiently accurate, meaning it is not the case that `%n` appears *both* in the bytes of a random instruction *and* in a string literal. This would just be a problem with the limit study and would not affect the final transformation; though it seems likely that whatever reason led to poor string recovery might also lead to a format-string specializer concluding that the format string is dynamic. (In this case, the specializer would have to leave the generalized version in place, as described above.)

3.1.2 Measuring "excess features" in Grep

The second limit study we did was to try to measure how much of the `grep` utility is used in real-world usage. This study illustrates how LACI can be used by an end user who thinks that they likely only use a small part of the overall feature set of a program.

We wrote a script that examines a user's command-line history (e.g. via `.bash_history`) and extracts `grep` invocations, and then runs `grep` over those real-world uses under profiling using QEMU. The profiling collects information about what instructions are visited. We compared the proportion of instructions that are visited in those test runs and the total number of instructions in the binary and its libraries. The grep invocations were over a test suite of a large quantity of C source files as well as the grep executable itself (as a representative of binary files).

If a user can guarantee that a workload was representative of the use they intend, anything that was not visited (or, perhaps, augmented with some additional instructions some analysis determines "important") could be removed from the program. This would allow a user to say "I never use *such-and-such* feature; I want to remove it from the program."

This experiment will underapproximate the needed set of instructions, because it is tracing based. Things like error handling should not be cut out (and would not be with a tool such as partial evaluation), and it is possible that some pattern edges were not triggered. However, as will be seen (§4.2), the proportion of the program that is actually executed is so small that even if the underapproximation were substantial (e.g., an order of magnitude), it would still be beneficial to reduce.

3.1.3 Measuring the change in VSA coverage given a configuration

The third level of users we are aiming for is system administrators; we would like LACI to be usable by a sysadmin to specialize a program for a particular environment or configuration.

Under DARPA's VET program, we implemented a new mode of value-set analysis called "VSA with trace seeding," described in §2.2. However, we did not perform much evaluation of it under VET. At a high level, trace seeding is a bit like applying the idea of partial evaluation to a static analysis. As a result, we decided to evaluate trace seeding's effect under this project, to determine if it offers any insight on how much benefit we might see from partial evaluation.

Our evaluation consisted in comparing the proportion of nodes that VSA considers reachable when running with and without trace seeding. Our target program was rfid_reader, a program developed for one of the VET engagements; it is a daemon that monitors accesses to an RFID badge reader that should open (or not open) a door to a controlled area. The program can be configured on a number of fronts, but perhaps the most interesting front is whether or not it also requires the user enter a correct PIN before it opens the door. Whether or not it does is controlled by the configuration.

We gathered a trace of rfid_reader until the point at which it waits for input from the badge reader, by which point it has already read the configuration file and populated its internal structures. We then ran VSA twice, once using the trace and once without the trace, measuring the proportion of nodes that VSA considers reachable. The drop in nodes is attributable to the trace, which imposes a precondition ("these nodes are reachable *if* the program reads this configuration file").

3.2 Experiments with WIPER (the UW binary partial evaluator)

During the LACI project, we imported the research prototype of WIPER from UW (see §2.1.4) and worked on making it more robust, performant, and better-fitted to the goals of the project. The most interesting enhancement is the ability to specialize *parts* of a program; the idea is that we could specialize several parts and then reassemble. We call this "local partial evaluation" (see §3.2.1).

There were a few technical challenges involved in the import process. UW developed WIPER using an older version of CodeSurfer, and so we updated it to take advantage of more recent improvements. We also spent time understanding how it is structured.

The first obstacle we encountered was how to specify the static and dynamic inputs. In their experiments, UW manually specified the instructions that directly read (dynamic) input; those instructions served as the starting point of the slice that is used for BTA. However, manually specifying these instructions is not realistic for larger programs.

Instead of manually marking dynamic input, we considered the following as dynamic:

- The arguments to `main` (`argc`, `argv`, `envp`).
- Return values from system calls and undefined functions.
- Variables whose address is taken. Because it is difficult to determine when the address of a variable is passed to a system function (for output to that variable), we conservatively mark the abstract memory locations representing such variables as dynamic. This also allows us to forgo the need for pointer analysis; any variable that could be written via pointer dereference is deemed dynamic. This selection errs on the side of playing things safe, reserving partial evaluation for only parts of the code that we are *certain* are not affected by the dynamic input.

3.2.1 "Local" partial evaluation

One of the opportunities where we think partial evaluation is promising is the interface between client and library code. Assuming developers often do not need the full generality present in a given library, it is likely that a lot of the library code can be trimmed down with partial evaluation.

With this in mind, we investigated the possibility of applying the partial evaluator on subtrees of the call graph, a technique we call local partial evaluation. In particular, we looked at function calls in which one or more of the parameters passed to the function is a constant value. If we can generate a customized version of the function (and its callees) based on those fixed parameters, then we can eliminate excess code and computation for that specific calling context.

Our first approach to this was to constrain the partial evaluator to just the functions that are transitively called from a given call site. As described earlier, WIPER's binding-time analysis leverages CodeSurfer's dependence analysis, which computes the data and control relationships between different instructions in the program; the BTA operates by performing a forward slice from the dynamic seeds of the program. Instructions covered by the slice are deemed dynamic. Those not covered by the slice are static.

For local partial evalutaion, we operated at the subcomponent level, starting the slice at the non-constant parameters to a specific function call. (We also include other possible sources of dynamic input within the subcomponent as well.) The instructions not covered by the slice can be deemed static with respect to the specific calling context that we're interested in.

We specialized components of several programs from the SPEC benchmark suite. Results are described in §4.4.2.

4 Results and Discussion

This section describes the results and some conclusions from each of the experiments and implementation efforts we carried out during the LACI project.

4.1 Format string measurements and function hardening

As described in §3.1.1, first, we looked at how many strings of length at least two (`strings -n2`) contain `%n`:

<table>
<tr><td rowspan="6">Total strings: 7,146,777</td><td rowspan="3">In a data/code section: 6,778,704 (95.5%)</td><td rowspan="2">Unique: 2,038,348 (28.5% of total, 30.0% of parent)</td><td>Safe (no %n): 2,032,877 (28.4% of total, 99.7% of parent)</td></tr>
<tr><td>Dangerous: 5,471 (0.08% of total, 0.3% of parent)</td></tr>
<tr><td rowspan="2">Duplicates: 4,750,356</td><td>Safe: 4,745,562</td></tr>
<tr><td>Dangerous: 4,794</td></tr>
<tr><td rowspan="3">Outside of all data/code sections: 358,073</td><td rowspan="2">Unique: 37,289</td><td>Safe: 37,129</td></tr>
<tr><td>Dangerous: 160</td></tr>
<tr><td rowspan="2">Duplicates: 320,784</td><td>Safe: 320,092</td></tr>
<tr><td>Dangerous: 692</td></tr>
</table>

From these numbers, we can see that strings containing `%n` are relatively rare, occurring just over 10,000 times (across 20,907 files, as can be seen in the next table). Furthermore, close to half of those are duplicates of others, so there are fewer than 5,600 strings; on top of *that*, based on the

proportion of false positives from our detailed look at /usr/sbin (below), a large proportion (very likely, more than half) of these strings are false positives.

Our file-centric view of where `%n` is used:

Total files: 20,907	'`strings -a -n2`' found a %n: 1,557 (7.4%)	/usr/bin: 319
		/usr/sbin: 28
		/usr/lib: 1,210
	No %n in the file: 19,350 (92.6%)	/usr/bin: 1,962
		/usr/sbin: 209
		/usr/lib: 17,179

Here things look even better: 92.6% of files appear to not contain `%n`. It is possible that some of them might still depend on the ability to use that format specifier because of dynamically-constructed format strings, but this seems pretty unlikely. The proportion of `%n`-free strings in `/usr/bin` is a little lower (86%), but this suggests that a significant majority of programs might be able to function with no change other than replacing `printf` with a hardened version.

And finally, our detailed look at the `%n` occurrences in `/usr/sbin`:

Total files: 237	Safe on their face (no string with %n): 209 (88.25)			
	Contains %n: 21	Not a binary: 9		
		Binaries: 12	%n in instruction bytes (likely false positive): 6	
			Calls format string function with %n: 5 (2.1% of total)	`grub-*`: 4
				`tcpdump`: 1
			Other (may or may not): 1	

Of these 237 files from `/usr/sbin`, only 5 or 6 files (2% – 2.5%) make use of `%n`; four of those files are from a family of `grub` utilities and use it in a similar way. The `try-from` utility calls functions in a shared library with a string containing `%n`; we did not trace it through to find its ultimate use, so that may or may not ultimately depend on passing it to a format string.

The 6 entries "instruction bytes include %n" are likely false positives. In these cases, `%n` does not appear in the assembly listing we get from CodeSurfer for Binaries and/or IDA Pro, *and* there is an instruction that happens to include the bytes 0x25 0x6e (`%n`). (Note that `strings -d` searches the entire `.text` section.) Hypothetically, CodeSurfer and IDA Pro might be missing a string constant that appears in *addition* to the `%n` in the bytes of an instruction, but we think the chance of this is remote. Regardless, even if all of these files were actually true positives (i.e., they really do contain a format string with `%n`), the number of files in `sbin` that contain a `%n` literal format string would *still* only be about 5%.

To give further credence to our results, we also downloaded the source code for the Apache HTTPD server and the Apache Portable Runtime library (APR) and did a textual search for `%n`. (Apache HTTPD uses APR.) The only places that `%n` appears in the source tree is in comments.

Future directions and recommendations:

The results of our `printf` study show that there appears to be a lot of room to improve program security by providing a restricted version of the format string functions. As mentioned in §3.1.1, format-string vulnerabilities have been a significant source of program insecurity, and we think that removing the possibility of exploiting format string functions would be very beneficial.

There are multiple avenues for approaching this task. We identify two high-level axes on which there is a choice:

- Will just some of the existing `printf` calls be redirected, or just some? And if all will be redirected, will the original `printf` still be present in the program's address space? (It must be present if only some calls are redirected, of course.)
- Will calls to the original `printf` all call into one, reduced strength `printf`, or will they call separate functions with the format string "baked in"? (If multiple `printf` calls pass the same format string, they could share the same new function.)

This leads to the following possibilities, including leaving things the same. In the following table, the cell contents indicate possible implementation techniques, as described below.

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="3">What function will be called when the format string was a literal?</td></tr>
<tr><td>Original</td><td>safer</td><td>specialized</td></tr>
<tr><td rowspan="3">What function will be called when the format string is not a literal?</td><td>original</td><td>no change</td><td>rewriting*</td><td rowspan="2">rewriting**</td></tr>
<tr><td>safer</td><td rowspan="2">not possible</td><td>LD_PRELOAD</td></tr>
<tr><td>safer, and printf is absent</td><td>custom libc</td><td>rewriting** and custom libc (both necessary***)</td></tr>
<tr><td colspan="5">* Requires rewriting if there are non-literal calls; if all calls are literal, LD_PRELOAD suffices.

** Requires rewriting if there is more than one call. If there is only one call, LD_PRELOAD suffices.

*** Alternatively, <code>libc</code> could be rewritten, but the program would still have to use that rewritten version; or the program could be rewritten to statically link <code>libc</code>.</td></tr>
</table>

As you move further right or down in the table, the rewritten program becomes more secure:

- Moving right:
  - From first column to the second, corrupting the format string at literal call sites becomes less useful to an attacker, because the attacker will not be able to cause it to contain `%n`.
  - Moving from the second column to the third, corrupting the format string at literal call sites becomes outright impossible, because the format string becomes baked into the target function.
- Moving down:
  - Moving from the first row to the second makes any potential user control of the format string, or corruption of the format string at the non-literal call sites, less useful to an attacker (because the attacker will not be able to cause it to contain `%n`).
  - Moving from the second row to the third eliminates the possibility of a return-to-library attack with a forged, `%n`-containing format string that bypasses all calls in the program.

There are some restrictions to moving down the table. Moving from the first row to the second means that the program is no longer able to dynamically generate (or read from the user) a format string containing `%n` for the call sites that are changed. (Not all need be changed; some non-literal call sites could call the original `printf` while others call the safer `printf`.) Moving from the second row to the bottom row means that *every* call site in the program must be to the safer `printf` or to a specialized version, meaning it must be intended that *no* call site in the program ever can use `%n`. This is probably a good thing – the hypothesis that our study was trying to establish – but hypothetically could restrict the application. More realistically, it may not be possible to automatically determine whether a non-literal call site is safe to restrict to the safer version.

The table lists the *simplest* implementation technique that we think could effectively achieve the desired effects. From simplest to most complex:

- *LD_PRELOAD* – use a library with a custom `printf` implementation, and use `$LD_PRELOAD` to ensure it is loaded. The library's `printf` will interpose and superceed on `printf` calls.
- *custom libc* – build a custom version of `libc` with a safer `printf` (or without a `printf` entirely) and ensure it is loaded by the target program. If the custom library provides its safer version under the name `printf`, then this suffices; it could also provide it under a different name, but that would require rewriting the program to redirect calls.
- *rewriting* – a binary rewriter would be necessary to rewrite calls to the original `printf` so they point somewhere else.

Except for the lower-right cell of the table, later implementation options subsume earlier ones. For example, if you use a rewriter, you do not also need a custom libc (except for the lower-right cell). Also, the above options apply only for programs that dynamically link against `libc`; statically-linked programs would always need a rewriter.

The general rules are:

- If it is possible that two different calls to `printf` should call different functions (either because we are specializing and they have different format strings, or because we want non-literal calls to call the original `printf`), then a rewriter is necessary.
- To ensure that `printf` is absent from the address space of the program, a custom `libc` is necessary.

4.2 Grep: unused features and functions

When the usage patterns of our engineer were measured, we found that only 3.25% of all instructions (6,499 of 199,907) in the program and its libraries were executed.

We think that the large number of unexecuted instructions are because of libraries that `grep` includes, but uses only small portions of. This is, at some level, a typical application area that we wanted to target for this project, and this servers as a further data point that the technique is very promising. Even if this number is underapproximating the usable code by an order of magnitude (due to inadequacies in our test suite, for example, but also taking into consideration any imprecision during the rewriting), that still means there is potential to remove two thirds of the aggregated program if the user is willing to say "I only want enough to use these features that I make common use of."

4.3 VSA trace seeding: improved precision from trace seeding

We saw a fairly significant drop in the proportion of nodes that VSA determines as reachable, as a result of using the dynamic trace as a seed.

With no trace seed, there are 3,382 nodes that VSA considers to be reachable. However, if we apply the trace, the number of reachable nodes drops to 2,609, a 23% decrease. (In both cases, this is out of a total of 19,408 nodes; the large number of unreachable nodes is mostly due to the executable containing code that is used in other versions of the rfid_reader test; this is similar to part of the motivation for LACI in the first place.)

Without a fuller job of reverse engineering rfid_reader it is hard to say what *exactly* is unreachable only under the trace seed assumption, but portions that seem to be removed include the code that checks the PIN. Under normal operation, the program first checks whether or not the configuration specifies that the PIN reader is present and necessary, then, if so, reads the pin from the user. A version specialized to a configuration saying that a PIN is not required could remove both the configuration check as well as the PIN check code; VSA trace seeding appears to be reflecting the static analysis analogue.

4.4 Adapting UW's partial evaluator

We first successfully replicated the improvements from partial evaluation that UW demonstrated. However, when we applied WIPER to new programs for LACI, we ran into several difficulties. When we tried to run WIPER on whole programs, the binding-time analysis marked nearly the entire program as being dynamic, which would lead to no meaningful specialization. We think the difference in results between our attempts and UW's is that we were applying it in very different context, with less well-defined inputs (because UW's approach for specifying inputs would not be appropriate for a LACI-like tool, requiring a manual specification of all the inputs). This problem is described in more detail in §4.4.1.

We then worked on local partial evaluation, as described in 3.2.1; results from those experiments are described in §4.4.2. We hit another engineering issue, which was a failure of WIPER's instruction synthesis; that is discussed briefly in §4.4.3.

4.4.1 Binding-time analysis precision

Our initial experiments with the UW partial evaluator showed that only 3% of the programs were being marked as static. This would mean that the specializer would do effectively nothing to improve the program.

As mentioned in §2.1.4, the partial evaulator's binding-time analysis uses slicing to compute the static/dynamic division. Slicing can follow two kinds of dependence edges:

- A *data dependence* indicates that a value computed at one instruction is (directly) used at another instruction.
- A *control dependence* indicates that a control-flow operation performed at one instruction affects whether another instruction is executed.

Obviously, data dependencies indicate direct flow of dynamic information through the system from one computation to the next. Control dependences represent more subtle flow of information. Consider the following function:

```
int foo(int input_var)
{
    int x;
```

```
    int y;
    if (input_var > 10)
     x = 1;
    else
     x = 2;
    y = x;
    return y;
}
```

In this example, the assignment to `y` has data dependences on both of the two assignments to `x`. However, neither assignment to `x` has any data dependencies on any other instructions – their computation has no inputs. Thus, considering data dependence alone, the assignment to `y` does not appear to be dependent on dynamically provided input. Yet clearly that is not true, and the final value of `y` *does* depend on `input_var`. This is where control dependence comes in. Both of the assignments to `x` are control-dependent on the conditional of the `if` statement, which in turn is data-dependent on `input_var`. By following both kinds of edges, we can correctly detect that the assignment to `y` and `y`'s result are, in fact, dynamic.

However, leveraging control dependence results in a conservative overapproximation of the set of instructions that should be considered dynamic. In the above example, both assignments to x would also be labeled dynamic and the partial evaluator would residuate them. In this case, this wouldn't be too big of a problem. A more problematic case is the following kind of idiom:

```
void bar(int input_var)
{
    if (!is_valid_input(input_var) {
     report_error();
     return;
    }
    /* ... rest of the function ... */
}
```

Here we have an error check at the beginning of the function to exit early if invalid data is provided. The "rest of the function" portion of the function is entirely control-dependent on the initial error check. As a result, the slice performed by the partial evaluator will label effectively the entire function as dynamic, and likely miss any opportunities for optimization.

This problem can propagate up and effect basically the entire program – for example, if a program's `main` contains an early exit (perhaps a check for a correct number or the correct formats of command line arguments and returns if not) or puts the main portion of the program in the body of a conditional or loop that is dynamic, then all functions called under that would be marked dynamic as well – because we are talking about `main`, the dynamic portion would likely be almost all of the program's functionality.

There are a couple of ideas that we have for improving things to a production level. Using a BTA with different attributes (as discussed in §2.1.2) should lead to much better results. For example, an online evaluator would not have the control-dependence problem.

An alternative approach is to augment the way the control dependencies are used by the offline BTA analysis. One option is to relax the reliance on the control dependence by following a set of heuristics. For example, excluding control dependence edges from parameter checks may get us

a certain distance.

However, from looking at cases where this problem arises, we also think that a more principled solution is possible, using data dependence edges. Take for example the following code:

```
void bar(int input_var)
{
1:  int x = input_var;
2:  int y = 0;
3:  while (x > 0) {
4:      int z = f(); // Constant computation
5:      y = g(y, z); // Computation dependent on x
6:      x--;
7:  }
8:  return y;
}
```

Here, the call to function `f()` when initializing `z` inside the loop (line 4) is constant and always returns the same value. Ideally, we'd like to evaluate this computation statically, eliminate the variable `z`, and inline the appropriate value into the call to function `g()`. Note that we cannot do the same for `g()` and `y`, because the first parameter to `g()` is the previous value that `y` had (either its initial value, 0, or the value resulting from the previous execution of the while loop.)

Control dependence would cause BTA to mark the entire loop body as dynamic, and the specializer would do no optimization. However, note that every time line 4 executes, it performs exactly the same computation. In contrast, line 5 performs a (potentially) different computation on each execution. A key distinction between the two lines is that line 5 has a cycle in its data dependence graph – in fact it is dependent on itself.

This observation may provide a key to a more principled solution: ignore control dependences for any computation that has no cycles in its backward slice. It is not clear whether this rule would hold up in all situations, but it seems like it would be a promising direction for future work on the partial evaluator. Unfortunately, we did not reach a conclusion as to the efficacy of this analysis before the end of the project.

4.4.2 Local partial evaluation results

The following tables show results from rewriting based on constant arguments. The tests we used are the following tests from the SPEC benchmark suite:

test	# instr	# procs
Astar	8,703	124
bzip2	11,826	111
hmmer	59,638	622
libquantum	9,052	139
Sjeng	21,820	191

We determined the procedures that are called with a constant literal argument, and specialized them. Not all specialization attempts succeeded; many timed out or crashed. Results for the attempts that succeeded are show below.

The following table shows:

- "# procs orig.": the number of such procedures
- "# procs rewrit.": the number of procedures they turn into in the rewritten executable. (If a procedure is called with multiple constant arguments, it will be duplicated and specialized multiple times.)
- "# instr orig.": the number of instructions in the procedures that we rewritten, in the orignial version
- "# instr duped": the number of instructions in the original version, if each of them were duplicated the number of times that procedure is specialized in the rewritten program
- "# instr rewrit.": the number of instructions in the rewritten version. Percentage changes are given relative to "# instr duped".

test	# procs orig	# procs rewrit	# instr orig	# instr duped	# instr rewrit
Astar	4	6	123	269	275 (+2.2%)
bzip2	2	18	42	406	196 (-52%)
hmmer	3	8	86	216	157 (-27%)
libquantum	4	6	73	123	106 (-14%)
sjeng	5	18	920	1,199	772 (-36%)

There are several conclusions from this data:

- Very few procedures are called with constant arguments, in the manner we are detecting it. However, our detection is reasonably weak; there is no dataflow analysis that helps determine whether arguments are constant.
- Different calling contexts lead to a duplication of procedures, in many cases many duplicates. This generally leads to an *increase* in program size, not a decrease.
- Specialization is generally able to substantially reduce the size relative to "# instr duped" – the median percentage decrease was 27%, and the total decrease across all procedures was about 32%.

We think that these data show promise for future work on the subject, though it is also clear that we are still a fair distance away from binary partial evaluation being applicable in real-world scenarios.

When looking at the breakdown of binding times, the results for these tests are better than the overall program (the 3% figure). The following table shows the number of instructions with each binding time. "# dynamic" and "# static" should be expected; the other two binding times are:

- "# dynamic-": these are dynamic instructions that also have a static side effect that is

interpreted (such as a stack update).
- "# lifted": these are static instructions that are a data predecessor of a dynamic instruction; for each of these instructions, a residual instruction needs to be generated to load the static value to the expected place.

The need for these is somewhat technical. Lifting is a standard partial evaluation top; for more information about the "dynamic-" binding time, see the UW paper on WIPER [12].

test	# dynamic	# dynamic-	# lifted	# static	total
astar	193	0	21	58 (21%)	272
bzip2	44	34	22	62 (38%)	162
hmmer	69	30	14	14 (11%)	127
libquantum	49	6	12	24 (26%)	91
sjeng	503	42	65	52 (7.9%)	662

As can be seen from the table, most instructions are still being determined to be dynamic, but a larger proportion are static – 8% to 38% static, with a median of 21%.

4.4.3 Instruction synthesizer difficulty

As discussed in §2.1.4, WIPER uses *instruction synthesis* to generate instruction sequences in the residual program, converting the partially evaluated program state for static code back into instructions when lifting that state for dynamic instructions. The synthesizer uses an approach inspired by superoptimization [10] – it searches for the smallest sequence of instructions that can result in a specific program state.

However, the instruction synthesizer often fails to produce a valid instruction sequence. We did not have time to diagnose all of the issues with the synthesizer; some possible causes could be:

- The instruction templates the synthesizer uses could be not expressive enough to capture the necessary program state.
- The search takes too long and times out. (At its core, the synthesizer uses an SMT solver to determine whether a candidate instruction sequence has the correct semantics.
- Perhaps the most likely scenario is that the synthesizer needed more work than we had in order to mature it beyond a research prototype.

5 Conclusions

The bulk of our work on LACI was on partial evaluation for machine code, building on foundational work from UW in the form of LACI. We enhanced the UW prototype to be much more capable and experimented with applying it to real world software; however, we ran into some challenges, such as the need for a more precise binding-time analysis.

Despite the challenges that we faced, we are still optimistic about the promise of using partial evaluation to optimize and minimize programs for specific deployment scenarios. Our

experiments, as well as UW's, still show promise, and there is an extensive academic literature of partial evaluation, and techniques from partial evaluation are used in production tools. (For example, partial evaluation has parallels to, and have influenced, just-in-time compilers [14].) The specific case of partial evaluation of binary code is substantially more difficult (like binary analysis in general), and our results show that this is a useful direction to pursue in the future.

6 References

1. *Partial Evaluation*. 1993: Prentice Hall International.
2. *Partial Evaluation*. 2017: Springer.
3. Amin Shali and William R.Cook, *Hybrid partial evaluation.* In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*: ACM SIGPLAN.
4. Aung, M., Horwitz, S., Joiner, R., and Reps, T., *Specialization Slicing*. ACM Transactions on Programming Languages and Systems, 2014. **36**(2): pp. 1-67.
5. Balakrishnan, G. and Reps, T., *Analyzing memory accesses in x86 executables*. Comp.Construct., 2004: pp. 5-23.
6. Balakrishnan, G. and Reps, T., *Analyzing Memory Accesses in x86 Executables.* In *International Conference on Compiler Construction (CC)*. 2004. Barcelona, Spain: Springer Verlag. pp. 5-23.
7. Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-yuh Evan Chang, Samuel Z.Guyer, Uday P.Khedker, Anders Møller, and Dimitrios Vardoulakis, *In Defense of Soundiness: A Manifesto*. Communications of the ACM, 2015. **58**(2): pp. 44-46.
8. Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R., *Control-flow bending: On the effectiveness of control-flow integrity.* In *24th USENIX Security Symposium (USENIX Security 15)*. pp. 161-176.
9. Karl Chen and David Wagner, *Large-scale analysis of format string vulnerabilities in Debian Linux.* In *ACM workshop on programming languages and analysis for security*.
10. Massalin, H., *Superoptimizer: A look at the smallest program.* In *Architectural Support for Programming Languages and Operation Systems (ASPLOS)*.
11. Niels H.Christensen, Robert Glück, and Søren Laursen, *Binding-Time Analysis in Partial Evaluation: One Size Does Not Fit All.* In *Perspectives of System Informatics*: Springer.
12. Srinivasan, V. and Reps, T., *Partial evaluation of machine code.* In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. pp. 860-879.
13. Srinivasan, V. and Reps, T., *Synthesis of machine code from semantics.* In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 2015. pp. 596-607.
14. Stefan Marr and Stéphane Ducasse, *Tracing vs. partial evaluation: comparing meta-compilation approaches for self-optimizing interpreters.* In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.