Translingual Obfuscation

Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University
{*pxw172, szw175, jum310, yzj107, dwu*}*@ist.psu.edu*

***Abstract*—Program obfuscation is an important software protection technique that prevents attackers from revealing the programming logic and design of the software. We introduce *translingual obfuscation*, a new software obfuscation scheme which makes programs obscure by "misusing" the unique features of certain programming languages. Translingual obfuscation translates part of a program from its original language to another language which has a different programming paradigm and execution model, thus increasing program complexity and impeding reverse engineering. In this paper, we investigate the feasibility and effectiveness of translingual obfuscation with Prolog, a logic programming language. We implement translingual obfuscation in a tool called BABEL, which can selectively translate C functions into Prolog predicates. By leveraging two important features of the Prolog language, i.e., unification and backtracking, BABEL obfuscates both the data layout and control flow of C programs, making them much more difficult to reverse engineer. Our experiments show that BABEL provides effective and stealthy software obfuscation, while the cost is only modest compared to one of the most popular commercial obfuscators on the market. With BABEL, we verified the feasibility of translingual obfuscation, which we consider to be a promising new direction for software obfuscation.**

1. Introduction

Obfuscation is an important technique for software protection, especially for preventing reverse engineering from infringing software intellectual property. Generally speaking, obfuscation is a semantics-preserving program transformation that aims to make a program more difficult to understand and reverse engineer. The idea of using obfuscating transformations to prevent reverse engineering can be traced back to Collberg et al. [17], [18], [52]. Since then many obfuscation methods have been proposed [42], [51], [54], [61], [16], [73].

Currently the state-of-the-art obfuscation technique is to incorporate with *process-level virtualization*. For example, obfuscators such as VMProtect [9] and Code Virtualizer [4] replace the original binary code with new bytecode, and a custom interpreter is attached to interpret and execute the bytecode. The result is that the original binary code does not exist anymore, leaving only the bytecode and interpreter, making it difficult to directly reverse engineer [35]. However, recent work has shown that the decode-and-dispatch execution pattern of virtualization-based obfuscation can be a severe vulnerability leading to effective deobfuscation [21], [62], implying that we are in need of obfuscation techniques based on new schemes.

We propose a novel and practical obfuscation method called *translingual obfuscation*, which possesses strong security strength and good stealth, with only modest cost. The key idea is that instead of inventing brand new obfuscation techniques, we can exploit some existing programming languages for their unique design and implementation features to achieve obfuscation effects. In general, programming language features are rarely proposed or developed for obfuscation purposes; however, some of them indeed make reverse engineering much more challenging at the binary level and thus can be "misused" for software protection. In particular, some programming languages are designed with unique paradigms and have very complicated execution models. To make use of these language features, we can translate a program written in a certain language to another language which is more "confusing", in the sense that it consists of features leading to obfuscation effects.

In this paper, we obfuscate C programs by translating them into Prolog, presenting a feasible example of the translingual obfuscation scheme. C is a traditional imperative programming language while Prolog is a typical logic programming language. The Prolog language has some prominent features that provide strong obfuscation effects. Programs written in Prolog are executed in a *search-and-backtrack* computation model which is dramatically different from the execution model of C and much more complicated. Therefore, translating C code to Prolog leads to obfuscated data layouts and control flows. Especially, *the complexity of Prolog's execution model manifests mostly in the binary form of the programs,* making Prolog very suitable for software protection.

Translating one language to another is usually very difficult, especially when the target and source languages have different programming paradigms. However, we made an important observation that for obfuscation purposes, language translation could be conducted in a special manner. Instead of developing a "clean" translation from C to Prolog, we propose an "obfuscating" translation scheme which retains part of the C memory model, in some sense making

two execution models mixed together. We believe this improves the obfuscating effect in a way that no obfuscation methods have achieved before, to the best of our knowledge. Consequently in translingual obfuscation, the obfuscation does not only come from the obfuscating features of the target language, but also from the translation itself. With this new translation scheme we manage to kill two birds with one stone, i.e., solving the technical problems in implementing translingual obfuscation and strengthening the obfuscation simultaneously.

There may be of a concern that obfuscation techniques without solid theoretical foundations will not withstand reverse engineering attacks in the long run. However, research on fundamental obfuscation theories, despite promising process made recently [43], [31], [59], is still not mature enough to spawn practical protection techniques. There is a widely accepted consensus that no software protection scheme is resilient to skilled attackers if they inspect the software with intensive effort [19]. A recently proved theorem [12] partially supporting this claim states that, a "universally effective" obfuscator does not exist, i.e., for any obfuscation algorithm, there always exists an program that it cannot effectively obfuscate. Given the situation, it seems that developing an obfuscation scheme resilient to all reverse engineering threats (known or unknown) is too ambitious at this point. Hence, making reverse engineering more difficult (but not impossible) could be a more realistic goal to pursue.

We have implemented translingual obfuscation in a tool called BABEL. BABEL can selectively transform a C function into semantically equivalent Prolog code and compile code of both languages together into the executable form. Our experiment results show that translingual obfuscation is obscure and stealthy. The execution overhead of BABEL is modest compared to a commercial obfuscator. We also show that translingual obfuscation is resilient to one of the most popular reverse engineering techniques.

In summary, we make the following contributions in this research:

- We propose a new obfuscation method, namely translingual obfuscation. Translingual obfuscation is novel because it exploits exotic language features instead of ad-hoc program transformations to protect programs against reverse engineering. Our new method has a number of advantages over existing obfuscation techniques, which will be discussed in depth later.
- We implement translingual obfuscation in a tool called BABEL which translates C to Prolog at the scale of subroutines, i.e., from C functions to Prolog predicates, to obfuscate the original programs. Language translation is always a challenging problem, especially when the target language has a heterogeneous execution model.
- We evaluate BABEL with respect to all four evaluation criteria proposed by Collberg et al. [18]: potency, resilience, cost, and stealth, on a set of real-world C programs with quite a bit of complexity and diversity. Our experiments demonstrate that BABEL provides strong protection against reverse engineering with only modest cost.

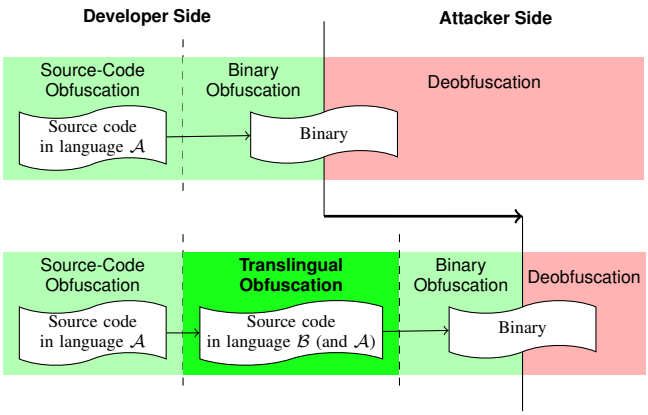

Figure 1. Translingual obfuscation is a new protection layer complementary to existing obfuscation methods, pushing the frontier forward in the battle with reverse engineering.

The remainder of this paper is organized as follows. §2 provides a high-level view on the insights and features of our translingual obfuscation technique. §3 explains in detail why the Prolog programming language can be misused for obfuscation. We summarize the technical challenges in implementing translingual obfuscation in §4. §5 and §6 present our C-to-Prolog translation method and the implementation details of BABEL, respectively. We evaluate BABEL's performance in §7. §8 has a discussion on some important topics about translingual obfuscation, followed by the summary of related work in §9. §10 concludes the paper.

2. Translingual Obfuscation

2.1. Overview

The basic idea of translingual obfuscation is that some programming languages are more difficult to reverse engineer than others. Intuitively, C is relatively easy to reverse engineer because binary code compiled from C programs shares the same imperative execution model with the source code. For some programming languages like Prolog, however, there is a much deeper gap between the source code and the resulting binaries, since these languages have fundamentally different abstractions from the imperative execution model of the underlying hardware. Starting from this insight, we analyze and evaluate the features of a foreign programming language from the perspective of software protection. We also develop the translation technique that transforms the original language to the obfuscating language. Only with these efforts devoted, translingual obfuscation can be a practical software protection scheme.

We view translingual obfuscation as a new layer of software protection in the obfuscation-deobfuscation arms race, as shown in Fig. 1. Different from previous obfuscation

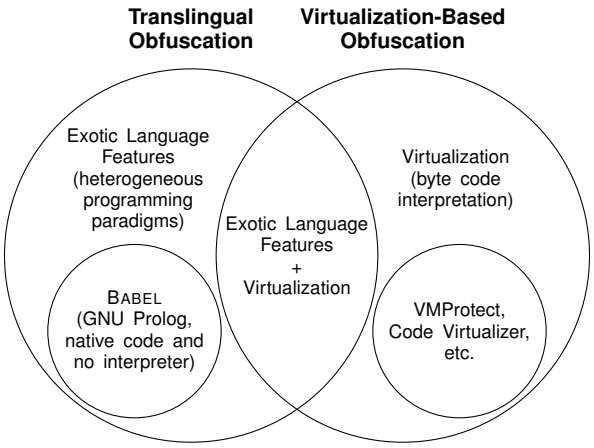

Figure 2. Comparing translingual obfuscation and virtualization-based obfuscation.

methods which either work at the binary level or perform same-language source-to-source transformations, translingual obfuscation translates one language to another. Therefore, translingual obfuscation can be applied after source-code obfuscation and before binary obfuscation without affecting the applicability of existing obfuscation methods.

2.2. Comparing with Virtualization-Based Obfuscation

The virtualization-based obfuscation is currently the state of the art in binary obfuscation. Some features of translingual obfuscation resemble the idea of virtualization-based obfuscation, but we want to emphasize a significant difference here. Currently, most implementations of virtualization-based obfuscation tend to encode original native machine code with a RISC-like virtual instruction set and interpret the encoded binary in a decode-dispatch pattern [63], [56], [33], which has been identified as a notable weakness of security and can be exploited by various attacks [62], [21], [75]. Translingual obfuscation, however, gets most of its security strength by *intentionally* relying on obfuscation-contributing language features that comes from a heterogeneous programming model. Essentially, translingual obfuscation does not have to re-encode the original binary as long as the foreign language employed supports compilation into native code. Fig. 2 shows the relationship and key differences between the two methods. Our translingual obfuscation implementation BABEL and virtualization-based obfuscation do not overlap.

2.3. Benefits

Translingual obfuscation can provide benefits that cannot be delivered by any single obfuscation method developed before, to the best of our knowledge:

- Translingual obfuscation provides strong obfuscation strength and more obfuscation variety by introducing a different programming paradigm. If there exists *a universally effective and automated method* to nullify the obfuscation effects, namely the additional program complexity, introduced by a programming language's execution model, that would mean it is possible to significantly simplify the design and implementation of that language, which is very unlikely for mature languages.
- Translingual obfuscation can be very stealthy, because programming with multiple languages is a completely legit practice. Compared with virtualization-based obfuscation which encodes native code into bytecode that has an exotic encoding format, translingual obfuscation introduces neither abnormal byte entropy nor deviant instruction distributions.
- Translingual obfuscation is not just a single obfuscation algorithm but a general framework. Although we particularly utilizes Prolog in this paper, there are other languages that can be misused for translingual obfuscation. For example, the New Jersey implementation of ML (SML/NJ) [10] does not even include a runtime stack. Instead, it allocates all frames and closures on a garbage-collected heap, potentially making program analysis much more difficult. Another example is Haskell, a pure functional language featuring lazy evaluation [41] which can be implemented with a unique execution model that greatly differs from the traditional imperative computation [46].

All these benefits make us believe that translingual obfuscation could be a new direction in software protection.

3. Misusing Prolog for Obfuscation

In this section we briefly introduce the Prolog programming language and explain why we can misuse its language features for obfuscation.

3.1. Prolog Basics

The basic building blocks of Prolog are *terms*. Both a Prolog program itself and the data it manipulates are built from terms. There are three kinds of terms: constants, variables, and structures. A constant is either a number (integer or real) or an atom. An atom is a general-purpose name, which is similar to a constant string entity in other languages. A structure term is of the form $f(t_1, \cdots, t_n)$, where f is a symbol called a *functor* and $t_1, \cdots, t_n$ are subterms. The number of subterms a functor takes is called its *arity*. It is allowed to use a symbol with different arities, so the notation '`f/n`' is used when referring to a structure term f with n subterms.

Structure terms become *clauses* when assigned semantics. A clause can be a fact, a rule, or a query. A predefined clause is a fact if it has an empty body, otherwise it is a rule. For example, "`parent(jack,bill).`" is a fact, which could mean that "jack is a parent of bill." One the other hand, a rule can be like the following line of code:

```
grandparent(G,C):-parent(G,P),parent(P,C).
```

This rule can be written as the following formula in the first-order logic:

$$\forall G, C, P.\text{grandparent}(G, C) \leftarrow \text{parent}(G, P) \wedge \text{parent}(P, C)$$

Clauses with the same name and the same number of arguments define a relation, namely a *predicate*. With facts and rules defined, programmers can issue *queries*, which are formulas for the Prolog resolution system to solve. In accordance with our previous examples, a query could be `grandparent(G,bill)` which is basically asking "who are bill's grandparents?"

A Prolog program is a set of terms. The Prolog resolution engine maintains an internal database of terms throughout program execution, trying to resolve queries with facts and rules by logical inference. Essentially, computation in Prolog is reduced to a searching problem. This is different from the commonly seen Turing machine computation model but the theoretical foundation of logic programming guarantees that Prolog is Turing complete [64].

3.2. Obfuscation-Contributing Features

3.2.1. Unification. One of the core concepts in automated logic resolution, hence in logic programming, is unification. Essentially it is a pattern-matching technique. Two first-order terms t_1 and t_2 can be unified if there exists a substitution σ making them identical, i.e., $t_1^\sigma = t_2^\sigma$. For example, two terms $k(s(g), Y)$ and $k(X, t(k))$ are unified when X is substituted by $s(g)$ and Y is substituted by $t(k)$.

Unification is one of the basises of Prolog's computation model. We show this by example. The following clause defines a simple "increment-by-one" procedure:

```
inc(Input,Output):-Output is Input+1.
```

Now for a query `inc(1,R)`, the Prolog resolution engine will first try to unify `inc(1,R)` with `inc(Input,Output)`, which means `Input` should be unified with `1` and `Output` should be unified with `R`. Once this unification succeeds, the original query is reduced to a subgoal `Output is Input+1`. Since `Input` is now unified with `1`, `Input+1` is evaluated as `2`. Finally `Output` gets unified with `2` (`is/2` is the evaluate-and-unify operator predicate), making `R` unified with `2` as well.

To support unification, Prolog implements terms as vertices in directed acyclic graphs. Each term is represented by a `<tag,content>` tuple, where `tag` indicates whether the type of the term and `content` is either the value of a constant or the address of the term the variable is unified with. Fig. 3 is an example showing how Prolog may represent a term in memory [11].

Unification makes data shapes in Prolog program memory dramatically different from C and much more obscure. The graph-like implementation of unification poses great challenges to binary data shape analyses which aim to recover high-level data structures from binary program images [36], [32], [58], [22]. Even if some of the graph

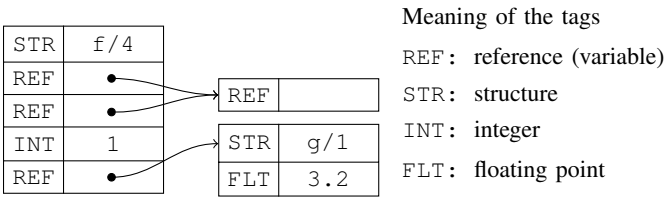

Figure 3. An example memory representation of term `f(X,Y,1,g(3.2))` in Prolog, where both `X` and `Y` are unified with another variable which itself is un-unified.

structures can be identified, there is still a gap between this low-level representation and the logical organization of original data, which harshly tests attackers' reverse engineering abilities. Unification also complicates data access. To retrieve the true value of a variable, the Prolog engine has to iterate the entire unification list. It is well known that static analysis is weak against loops and indirect memory access. Also, the tags in the term tuples are encoded as bit fields, meaning that bit-level analysis algorithms are required to reveal the semantics of a binary compiled from Prolog code. However, achieving bit-level precision is another great technical challenge for both static and dynamic program analyses, mainly because of scalability issues [60], [38], [26], [74].

3.2.2. Backtracking. Different from Prolog unification which mainly obfuscates program data, the backtracking feature obfuscates the control flow. Backtracking is part of the resolution mechanism in Prolog. As explained earlier, finding a solution for a resolvable formula is essentially searching for a proper unifier, namely a substitution, so that the substituted formula can be expanded to consist of only facts and other formulas known to be true. Since there may be more than one solution for a unification problem instance, it is possible that the resolution process will unify two terms in the way that it makes resolving the formula later unfeasible. As a consequence, Prolog needs a mechanism to roll back from an incorrect proof path, which is called backtracking.

To make backtracking possible, Prolog saves the program state before taking one of the search branches. This saved state is called a "choice-point" by Prolog and is similar to the concept of "continuation" in functional programming. When searching along one path fails, the resolution engine will restore the latest choice-point and continue to search through one of the untried branches.

This *search-and-backtrack* execution model leads to a totally different control flow scheme in Prolog programs at the low level, compared to programs in the same logic written by C. Fig. 4 is an example where a C function is transformed into a Prolog clause by our tool BABEL (with manual edits to make the code more readable), along with the program execution flows before and after BABEL transformation. The real control flow of the Prolog version of the function is much more complicated than presented, and we have greatly simplified the flow chart for readability.

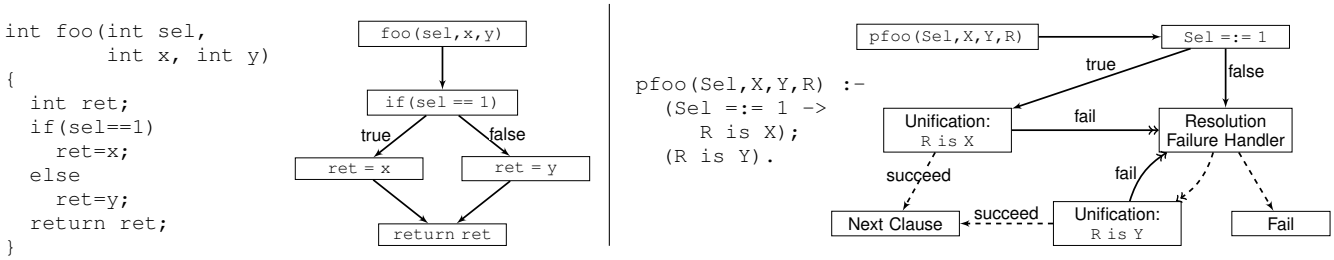

Figure 4. Different control flows of C and Prolog binaries implementing the same algorithm, due to different execution models. In the Prolog graph, dashed lines indicate indirect jumps and arrows with the same pattern indicate feasible paths through the resolution failure handler. Both control flow graphs are summarized from post-compilation binaries.

In the Prolog part of Fig. 4, a choice point is created right after the execution flow enters the predicate `pfoo` which is a disjunction of two subclauses. The Prolog resolution routine will first try to satisfy the first subclause. If it fails, the engine will backtrack to the last choice-point and try the second subclause.

Due to the complicated backtracking model, a large portion of control flow transfers in Prolog are indirect. The implementation of backtracking also involves techniques such as long jump and stack unwinding. Clearly, Prolog has a much more obscure low-level execution model compared to C, and imperative programming in general, from the perspective of static analysis. Different from some other control flow obfuscation techniques that inject fake control flows which are never feasible at run time, Prolog's backtracking actually happens during program execution, making translingual obfuscation also resilient to dynamic analysis. Most importantly, after the C-to-Prolog translation the original C control flows are reformed with a completely different programming paradigm, which is fundamentally different from existing control-flow based obfuscation techniques.

4. Technical Challenges

To make use of Prolog's execution model for obfuscating C programs, we need a translation technique to forge the Prolog counterpart of a C function. At this point, there are various challenges to resolve.

4.1. Control Flow

As an imperative programming language, C provides much flexibility of crafting program control flows almost with language key words such as continue, break, and return. Prolog programs, however, have to follow the general evaluation procedure of logical formulas, which inherently forbids some "fancy" control flows allowed by C.

4.2. Memory Model

In C programming, many low-level details are not opaque to programmers. As for memory manipulation, C programmers can access almost arbitrary memory locations via pointers. Prolog lacks the semantics to express direct memory access. Moreover, the C memory model is closely coupled with other sub-structures of the language, e.g., the type system. C types are not only logical abstractions but are also implications on low-level memory layouts of the data. For instance, logically adjacent elements in a C array and fields in a C struct are also physically adjacent in memory. Therefore, some logical operations on C data structures can be implemented as direct memory accesses which are semantically equivalent only with the C memory modeling. Below is an example.

```
struct ty {
 int a;
 int b;
} s[2];

/* Equivalent to s[0].a=s[0].b=s[1].a=0;
 * with many compilers and architectures */
memset((void*)s, 0, 3*sizeof(int));
```

Translating the code snippet above into pure Prolog could be difficult because the translator will have to infer the logic effects of the `memset` statement.

4.3. Type Casting

C type casting is of full flexibility in the sense that a C programmer can cast any type to any other type, no matter the conversion makes sense or not. This can be realized by violating the *load-store consistency*, namely storing a variable of some type into a memory location and later loading the content of the same chunk of memory into a variable of another type. The C union type is a high-level support for type castings that breaks the load-store consistency, but C programmers can choose to use pointers to directly achieve the same effect. Imitating this type casting system could be a notable challenge for other languages.

5. C-to-Prolog Translation

This section explains how we address the challenges mentioned in the previous section.

5.1. Control Flow Regularization

There has been a large amount of research on refining C program control flows, especially on eliminating goto

statements [39], [55], [70]. For now, we consider that goto elimination is a solved problem and assume the C programs to be protected do not contain goto statements. Given a C function without goto statements, there are two control flow patterns that cannot be directly adopted by Prolog programming, i.e., control flow cuts and loops. We call these patterns *irregular* control flows.

5.1.1. Control Flow Cuts. Control flow cuts refer to the termination of control flows in the middle of a C function, for example:

```
int foo (int m, int n) {
 if(m)
  return n; // Flow of if branch ends
 else
  n=n+1;
 n=n+2;
 return n;  // Flow of else branch ends
}
```

The C language grants programmers much freedom in building control flows, even without using goto statements. In Prolog, however, control flows have to be routed based on the short-circuit rules in evaluating logical expressions. With short-circuit effects, parallel statements can be connected by disjunction and sequential statements can be connected by conjunction. To show why the control flow pattern in the C code above cannot be implemented by only adopting short-circuit rules, consider a C function with body `{if(e) {a;} else {b;} c;}`. Naturally, it should be translated into a Prolog sentence `(((e->a);b),c)`, where `->` denotes implication, `;` denotes disjunction, and `,` denotes conjunction. However, this translation is not semantics-preserving when `a` is a return statement, because if the clause `a` is evaluated, at least one of `b` and `c` has to be evaluated to decide the truth of the whole logic formula.

We fix control flow cuts by replicating and/or reordering basic blocks syntactically subsequent to the cuts. For example, we rewrite the previously shown C function into the following structure:

```
int foo (int m, int n) {
 if(m)
  return n;
 else {
  n=n+1;
  { n=n+2; return n; }
 }
}
```

After the revision, the C code is naturally translated into a new Prolog clause `(e->a);(b,c)`, which is consistent with the original C semantics.

5.1.2. Loops. Most loops cannot be directly implemented in Prolog. The fundamental reason is that Prolog does not allow unifying a variable more than once. We can address this problem by transforming loops into recursive functions, but in-loop irregular control flows complicate the situation. The irregularity comes from the use of keywords "continue" and "return." A continue statement cuts the control flow in the middle of a loop, bringing up a problem similar to the aforementioned asymmetric returns in functions. As such, irregular control flows resulting from continue statements can be regularized in the same way, i.e., replicating and/or reordering basic blocks syntactically subsequent to continue statements.

Like a continue statement, a return statement also cuts the control flow in a loop, but its impact reaches outside because it cuts the control flow of the function enclosing the loop. Hence, a recursive Prolog predicate transformed from a loop needs an extra argument to carry a flag indicating whether an in-loop return has occurred.

5.2. C Memory Model Simulation

As stated in §4, the C memory model is closely coupled with other parts of the language and it is hard to separate them. However, translingual obfuscation keeps the original C memory model, making preserving semantic equivalence much easier. In our design, the Prolog runtime is embedded in the C execution environment, so it is possible for Prolog code to directly operate memories within a program's address space.

The way we handle C memory simulation illustrates the advantage of developing language translations for obfuscation purposes. Unlike tools seeking complete translation from C to other languages, translingual obfuscation does not have to mimic C memory completely with target language features (e.g., converting C pointers to Java references [23]), meaning we can reduce translation complexity and circumvent various limitations. That being said, partially imitating the C memory model in Prolog is still a non-trivial task.

5.2.1. Supporting C Memory-Access Operators. The first step to simulating the C memory model is to support pointer operations. We introduce the following new clauses into our target Prolog language:

```
rdPtrInt(+Ptr, +Size, -Content)
wrPtrInt(+Ptr, +Size, +Content)
rdPtrFloat(+Ptr, +Size, -Content)
wrPtrFloat(+Ptr, +Size, +Content)
```

These clauses are implemented in C. `rdPtrInt/3` and `rdPtrFloat/3` allow us to load the content of a memory cell (address and size indicated by `Ptr` and `Size`, respectively) into a Prolog variable `Content`. Similarly, `wrPtrInt/3` and `wrPtrFloat/3` can write the content of a Prolog variable into a memory cell. These four clauses simulate the behaviors of the "pointer dereference" operator (`*`) in C.

In addition to read-from-pointer and write-to-pointer operations, C also has the "address-of" operator which takes an lvalue, i.e., an expression that is allocated a storage location, as the operand and returns its associated storage location, namely address. There is no need to explicitly support this operator in Prolog because the address of any lvalue in C

has a static representation which is known by the compiler.[1] We can obtain the results of "address-of" operations in the C environment and pass those values into the Prolog environment as arguments.

We also handle several C syntax sugers related to memory access: "subscript" (`[]`) and "field-of" (`.` and `->`). We convert these operators into equivalent combinations of pointer arithmetic and dereference so that we do not need to coin their counterparts in Prolog. This conversion requires assumptions on compiler implementation and target architecture to calculate type sizes and field displacements.

5.2.2. Maintaining Consistency. It is a natural scheme that a C-to-Prolog translation maps every C variable to a corresponding Prolog variable. Prolog does not allow variable update, but we can overcome this restriction by transforming C code into a form close to static single assignment (SSA), in which variables are only initialized at one program location and never updated. In the strict SSA form, variables can only be statically initialized once even if the scopes are disjoint. Prolog does not require this because the language checks re-unification at run time, meaning variables can be updated in exclusively executed parts of the program, e.g., the "then" and "else" branches of the same if statement. Therefore, we do not need to implement the ϕ function in our SSA transformation.

The SSA transformation can be implemented by renaming variables. The challenging part is that simply renaming variables in the original C code could break program semantics because of side effects caused by memory operations, i.e., variable contents can be accessed without referring to variable names. This is the consistency problem we have discussed earlier. Fig. 5 shows an instance of the problem.

To address this issue, we keep the addresses of local variables and parameters if they are possibly accessed via pointers. Then we flush variable contents back to the memory before a read-from-pointer operation and reload variable contents from the memory after a write-to-pointer operation. Inter-procedural pointer dereferences are also taken into account. When callee functions accept pointers as arguments, we do variable flush before function calls and do variable reload after. The flush makes sure that changes made by Prolog code are committed to the underlying C memory before they are read again. Similarly, the reload assures that values unified with Prolog logical variables are always consistent with the content in C memory. We perform a sound points-to analysis to compute the set of variables that need to be reloaded or flushed at each program point. After inserting the flush and reload operations, the SSA variable renaming no longer breaks the original program semantics. Fig. 6 illustrates our solution based on the example in Fig. 5.

1. For example, a local variable is usually allocated on the stack and the compiler will have a static expression of its address. On x86, the expression is likely to be `$offset(%ebp)` or `$offset(%esp)`, where `$offset` is a constant. Compilers can also decide how to statically represent the addresses of global variables.

```
a=0;                          a1=0;
p=&a;// p points to a         p1=&a1;// p1 points to a1
a=1; // a gets 1              a2=1;  // a2 gets 1
b=*p;// b gets a(1)           b1=*p1;// b1 gets a1(0)

c=0;                          c1=0;
p=&c;                         p2=&c1;// p2 points to c1
c=1; // c gets 1              c2=1;  // c2 gets 1
*p=3;// c gets 3              *p2=3; // c1 gets 3
d=c; // d gets c(3)           d1=c2; // d1 gets c2(1)
```

(a) Original (b) Renamed

Figure 5. Memory operations affecting the correctness of C source code SSA renaming.

```
        pa=&a;                pa=&a1;// const pointer
        pc=&c;                pc=&c1;// const pointer

a=0;    a=0;                  a1=0;
p=&a;   p=&a;                 p1=&a1;
a=1;    a=1;                  a2=1;  // a2 gets 1
        *pa=a;// Flush        *pa=a2;// a1 gets a2(1)
b=*p;   b=*p;                 b1=*p1;// b1 gets a1(1)

c=0;    c=0;                  c1=0;
p=&c;   p=&c;                 p2=&c1;
c=1;    c=1;                  c2=1;
*p=3;   *p=3;                 *p2=3; // c1 gets 3
        c=*pc;// Reload       c3=*pc;// c3 gets c1(3)
d=c;    d=c;                  d1=c3; // d1 gets c3(3)
```

(a) Orig. (b) With flush and reload (c) Renamed with flush and reload

Figure 6. Semantic-preserving SSA renaming on C source code with the presence of pointer operations.

5.3. Supporting Other C Features

5.3.1. Struct, Union, and Array. In §4, we showed that C data types like struct and array can be manipulated via memory access. Since we have already built support for the C memory model in Prolog, the original challenge now becomes a shortcut to supporting C struct, union, and array. We simply transform the original C code and implement all operations on structs, unions, and arrays through pointers. After this transformation the primitive data types provided by Prolog are enough to represent any C data structure.

5.3.2. Type Casting. With our C memory simulation method, supporting type castings performed via pointers does not require additional effort, even if they may violate the load-store consistency. As for explicit castings, e.g., from integers to floating points, we utilize the built-in Prolog type casting clauses like `float/1`.

5.3.3. External and Indirect Function Call. Since the source code of library functions is usually unavailable, translating them into Prolog is not an option. In general, translations of translingual obfuscation can support external subroutine invocation with the help of foreign language interfaces. As for C+Prolog obfuscation, most Prolog implementations provide the interface for calling C functions

from a Prolog context. The same interface can also be used to invoke functions via pointers.

5.4. Obfuscating Translation

Our translation scheme fully exploits the obfuscation-contributing features introduced in §3.2, generally because:

- The conversion from C data structures to Prolog data structures happens by default, and every C assignment is translated to Prolog unification.
- Intra-procedural control-flow transfers originally coded in C are now implemented by Prolog's backtracking mechanism. This significantly complicates the low-level logic of the resulting binaries.

Especially, we would like to highlight the method we use to support the C memory model in Prolog. At the high level, the original C memory layout is kept after the translation. However, the behavior of the C-part memory becomes much different from the original program. To maintain the consistency between the C-side memory and Prolog-side memory, we introduce the flush-reload method which disturbs the sequence of memory access. In this way, the memory footprint of the obfuscated program is no longer what it was during program execution.

We believe our translation method is one of the factors that make translingual obfuscation resilient to both semantics-based and syntax-based binary diffing, as will be shown in §7.2.

6. Implementation of BABEL

BABEL is our translingual obfuscation prototype. The workflow of BABEL has three steps: C code preprocessing, C-to-Prolog translation, and C+Prolog compilation. The preprocessing step reforms the original C source code so that the processed program becomes suitable for line-by-line translation to Prolog. The second step translates C functions to Prolog predicates. In the last step, BABEL combines C and Prolog code together with a carefully designed interface.

We choose GNU Prolog [24] as the Prolog implementation to employ in BABEL. Like many other Prolog systems, GNU Prolog compiles Prolog source into the "standard" Warren Abstract Machine (WAM) [69] instructions. What is desirable to us is that GNU Prolog can further compile WAM code into native code. This feature makes BABEL more distinguishable from virtualization-based obfuscation tools which compile the original program to bytecode and execute it with a custom virtual machine.

6.1. Preprocessing and Translating C to Prolog

Before actually translating C to Prolog, we need to preprocess the C code first. The preprocessing includes the following steps, which is done with the help of the CIL library [53].

1) Simplify C code into the three-address form without switch statements and ternary conditional expressions.
2) Convert loops to tail-recursive functions.
3) Eliminate control flow cuts.
4) Transform operations on global, struct, union, and array variables into pointer operations.
5) Perform variable flush and reload whenever necessary.
6) Eliminate all memory operators except pointer dereferences.
7) Rename variables so that the C code is in a form close to SSA.

After preprocessing, we can translate C to Prolog line by line. The translation rules are listed in Fig. 7. Note that by the time we start translating C to Prolog, the preprocessed C code does not contain any switch and loop statements, because they are transformed into either nested if statements or recursive functions. As discussed in §5.1, we do not consider goto statements.

We take translating arithmetic and logical expressions as a trivial task, but that leads to a limitation in our translation. Due to the fact that Prolog does not subdivide integer types, integer arithmetics in Prolog are not equivalent to their C counterparts. For example, given two C variables `x` and `y` of type `int` (4 bytes long) and their addition `x+y`, the equivalent expression in Prolog should be `(X+Y)/\0xffffffff`, assuming that `X` and `Y` are the corresponding logical variables of `x` and `y`. Therefore, if a C program intentionally relies on integer overflows or underflows, there is a chance that our translation will fail. However, fully emulating C semantics incurs significant performance penalty.

Previous work on translating C to other languages faces the same issue, and many of them chose to ignore it [15], [48], [66]. The C-to-JavaScript converter Emscripten provides the option to fully emulate the C semantics [76]. It also has a set of optional heuristics to infer program points where full emulation is necessary, but that method is not guaranteed to work correctly. We do not particularly take this issue into account when implementing BABEL. However, we expect BABEL's translation to have a low failure chance thanks to the employment of write-to-pointer operations in Prolog and the variable flush-reload method. Since the write-to-pointer operation specifies data sizes, the truncation automatically takes place whenever an integer variable is flushed and reloaded. In GNU Prolog on 64-bit platforms, all integers are represented by 61-bit two's complement (3 bits are occupied by a WAM tag), which is large enough to hold most practical integer and pointer[2] values.

6.2. Combining C and Prolog

BABEL combines the C and Prolog runtime environments together, and the program starts from executing C

2. Most 64-bit CPUs only implement a 48-bit virtual address space.

Assignment	$(\texttt{foo = } e\texttt{;})^{\mathcal{T}}$	$\rightarrow$	$(\texttt{Pfoo is } e^{\mathcal{T}})$
Pointer arithmetic	$(\texttt{p2 = p1 + intVal;})^{\mathcal{T}} \mid \texttt{TypeOf(p1) = } T\star$	$\rightarrow$	$(\sigma(\texttt{p2}) \texttt{ is } \sigma(\texttt{p1}) \texttt{ + SizeOf}(T) \texttt{ * } \sigma(\texttt{intVal}))$
Pointer dereference	$(\texttt{foo = *p;})^{\mathcal{T}} \mid \texttt{TypeOf(p) = } T\star$	$\rightarrow$	$\texttt{rdPtr}(\sigma(\texttt{foo}), \texttt{ SizeOf}(T), \sigma(\texttt{p}))$
Write by pointer	$(\texttt{*p = foo;})^{\mathcal{T}} \mid \texttt{TypeOf(p) = } T\star$	$\rightarrow$	$\texttt{wrPtr}(\sigma(\texttt{p}), \texttt{ SizeOf}(T), \sigma(\texttt{foo}))$
Empty Block	$(\{\})^{\mathcal{T}}$	$\rightarrow$	$(\texttt{true})$
Non-empty Block	$(\{s_1 \cdots s_n\})^{\mathcal{T}}$	$\rightarrow$	$(s_1^{\mathcal{T}}, \cdots, s_n^{\mathcal{T}})$
Conditional	$(\texttt{if } (e) \{b_{then}\} \texttt{ else } \{b_{else}\})^{\mathcal{T}}$	$\rightarrow$	$(e^{\mathcal{T}}, \{b_{then}\}^{\mathcal{T}}; \{b_{else}\}^{\mathcal{T}})$
Function call	$(\texttt{ret = fun(a1, } \cdots\texttt{, an);})^{\mathcal{T}}$	$\rightarrow$	$\texttt{predFun}(\sigma(\texttt{a1}), \cdots, \sigma(\texttt{an}), \sigma(\texttt{ret}))$
Indirect function call	$(\texttt{ret = funptr(a1, } \cdots\texttt{, an);})^{\mathcal{T}}$	$\rightarrow$	$\texttt{predIndFun}(\sigma(\texttt{funptr}), \sigma(\texttt{a1}), \cdots, \sigma(\texttt{an}), \sigma(\texttt{ret}))$
Function return	$(\texttt{return } e\texttt{;})^{\mathcal{T}}$	$\rightarrow$	$(\mathcal{R} \texttt{ is } e^{\mathcal{T}})$
Function definition	$(\texttt{fun}(T_1 \texttt{ a1, } \cdots\texttt{, } T_n \texttt{ an) } \{b_{body}\})^{\mathcal{T}}$	$\rightarrow$	$\texttt{predFun}(\sigma(\texttt{a1}), \cdots, \sigma(\texttt{an})) \texttt{ :- } b_{body}^{\mathcal{T}}.$

Figure 7. Definition of $\mathcal{T}$, BABEL's C-to-Prolog translation. e, s, b, and T denote C expressions, statements, blocks, and types, respectively. σ is the bijective mapping from C identifiers to corresponding Prolog identifiers. $\mathcal{R}$ denotes the Prolog identifier used to hold the returned value in the translated predicate. `predFun` can be either a real Prolog predicate or a wrapper of a foreign C function, depending on whether the target function is translated or not. `predIndFun` is a wrapper for a special foreign C function which further calls into `funptr` with given arguments.

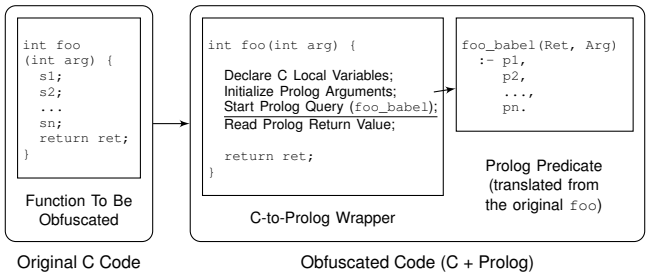

Figure 8. The context for executing obfuscated code in BABEL.

code. When the execution encounters an obfuscated function (which is now a wrapper for initiating queries to the corresponding Prolog predicate), it setups a context prior to evaluating the Prolog predicate. In the setup process the wrapper allocates local variables whose addresses are referred to in the preprocessed C function. The wrapper then passes the addresses along with function arguments to the Prolog predicate through the C-to-Prolog interface provided by GNU Prolog. Fig. 8 illustrates how the two languages are combined.

6.3. Customizing Prolog Engine

Although GNU Prolog has some nice features that make it a mostly adequate candidate for implementing BABEL, it still does not fully satisfy our requirements, thus requiring some customization. A notable issue about GNU Prolog is that its interface for calling Prolog from C is not reentrant. This is critical because by design, users of BABEL can freely choose the functions they want to obfuscate. To support this, it is in general not possible to avoid stack traces that interleave C and Prolog subroutines. We found that the non-reentrant issue results from the use of a global WAM state across the whole GNU Prolog engine. We fixed it by maintaining a stack to save the WAM state before a new C-to-Prolog interface invocation and restore the state after the call is finished.

Another issue is that GNU Prolog does not implement garbage collection; therefore memory consumption can easily explode. This problem is not as severe as it looks because we do not have to maintain a heap for Prolog runtime throughout the lifetime of the program. Because we know that the life cycles of all Prolog variables are bounded by the scope of predicates, we can safely empty the Prolog heap when there are no pending Prolog subroutines during the execution. Since GNU Prolog implements the heap as a large global array and indicates heap usage with a heap-top pointer, we can empty the heap by simply resetting the heap-top pointer to the starting point of the heap array, which is very efficient.

7. Evaluation

Collberg et al. [18] proposed to evaluate an obfuscation technique with respect to four dimensions: *potency*, *resilience*, *cost*, and *stealth*. Potency measures how obscure and complex the program has become after being obfuscated. Resilience indicates how well programs obfuscated by BABEL can withstand reverse engineering effort, especially automated deobfuscation. Cost measures the execution overhead imposed by obfuscation. Stealth measures the difficulty in detecting the existence of obfuscation, given the obfuscated binaries. We evaluate BABEL and observe to what extent it meets these four criteria.

To show that our tool can effectively protect real-world software of different categories, we apply BABEL to six open source C programs that have been widely deployed for years or even decades. Among the six programs, four are CPU-bound applications and the other two are IO-bound servers. The CPU-bound applications include algebraic transformation (bzip2), integer computation (mcf), state machine (regexp), and floating-point computation (svm_light). The two IO-bound servers cover two of the most popular network protocols, i.e., FTP (oftpd) and HTTP (mongoose). We believe that our selection is a representative evaluation set covering a wide range of real-world software. Table 1 presents the details of these programs.[3]

3. We notice that some previous work [16] on obfuscation employed the SPEC benchmarks or GNU Coreutils, which are also widely used in other research, for evaluation. Unfortunately, these two software suites use very complicated build infrastructures. Since BABEL needs to compile C and Prolog together, a specialized build procedure is required. Currently our prototypical implementation of BABEL cannot automatically hook an existing build system, so we are not able to include SPEC or GNU Coreutils into our evaluation.

TABLE 1. PROGRAMS USED FOR BABEL EVALUATION.

Program	Description	LoC	# of Func.
bzip2	Data compressor	8,117	108
mcf	Vehicle scheduler	2,685	25
regexp	Regular expression engine	1,391	22
svm_light	Support vector machine	7,101	103
oftpd	Anonymous FTP server	5,211	96
mongoose	Light-weight HTTP server	5,711	203

We define the term *obfuscation level* as the percentage of functions obfuscated in a C program. For example, an obfuscated bzip2 instance at the 20% obfuscation level is a bzip2 binary compiled from source code consisting of 80% of the original functions in C and Prolog predicates translated from the other 20% C functions by BABEL. We achieve all obfuscation levels by randomly selecting candidates from all functions that can be obfuscated by BABEL, but note that this random selection scheme is just for avoiding subjective picking in our research. In practice, BABEL users should decide which functions are critical and in need of protection. This is the same as popular commercial virtualization-based obfuscation tools [4], [9].

In the evaluation, we compare BABEL with one of the most popular commercial obfuscators, Code Virtualizer (CV) [4], which is virtualization based and has been on the market since 2006. The comparison covers all the four dimensions of evaluation, but some of the evaluation methodologies we designed for BABEL may not be suitable for evaluating Code Virtualizer. For those evaluations that we consider not suitable for CV, we will explain the reasons and readers should be cautious in interpreting the data.

7.1. Potency

We use two groups of static metrics to show how much complexity BABEL has injected into the obfuscated programs. The first group consists of basic statistics about the call graph and control-flow graph (CFG), including the number of edges in both graphs and the number of basic blocks. These metrics have been used to evaluate obfuscation techniques in related work [16].

In addition to basic statistics, we also calculate two metrics used to quantify program complexity, proposed by historical software engineering research. The measures are the cyclomatic number [49] and the knot count [71]. Both metrics reflect Gilb's statement that logic complexity is a measure of the degree of decision making within a system [34]. They also have been considered for evaluating obfuscation effects [18]. The cyclomatic number is defined as $e - n + 2$ where e and n are the numbers of edges and vertices in the CFG. Intuitively, the cyclomatic number represents the amount of decision points in a program [20]. The knot count is the amount of edge crossings in the CFG when all nodes are placed linearly and all edges are drawn on the same side.

Table 2 shows the comparison between binaries with and without BABEL obfuscation on the complexity measures we have chosen, at the obfuscation level of 30%.[4] To be conservative, by the time of measurement we have stripped the code belonging to the GNU Prolog runtime itself, so the extra complexity (if there is any) should be purely credited to BABEL's obfuscation. We use IDA Pro [6], an advanced commercial reverse engineering tool, to disassemble the binary and generate call graphs and control-flow graphs. As can be seen, the obfuscated binaries have a significant advantage on all metrics. The geometric mean of all six programs shows that BABEL can vastly increase program complexity. Note that different from static complexity produced by obfuscation methods using opaque predicates, the additional control-flow branches injected by BABEL are "real" in the sense that all branches can be feasible at run time.

Our potency evaluation is not an ideal methodology for measuring the same aspect of Code Virtualizer. The reason is that Code Virtualizer transforms binary instructions into bytecode and the reverse engineering tool we use, i.e., IDA Pro, is incapable of handling this situation. Table 3 shows the complexity of binaries with and without Code Virtualizer obfuscation, obtained in the same way as Table 2. The data suggests that after Code Virtualizer is applied, the complexity of the protected binaries has a notable decrease compared to the original ones. In reality, this is a consequence of the aforementioned issue. Because IDA Pro is unable to analyze the re-encoded functions, the potency evaluation inevitably misses a significant portion of the complexity.

7.2. Resilience

In general, the resilience of an obfuscation technique is hard to assess because reverse engineering can be a very ad-hoc process. On the other hand, very few practical deobfuscation tools are publicly available to the community. Because of these difficulties, some previous work on software obfuscation failed to evaluate resilience properly. In our evaluation, we choose binary diffing to assess BABEL's resilience after intensive investigation, although it does not mean binary diffing is the only deobfuscation technique.

Binary diffing is a commonly used reverse engineering technique which calculates the similarity between two binaries. We consider binary diffing a deobfuscation technique because it reveals the connection from an obfuscated program to its original version. Given a program binary and its obfuscated version, if a binary diffing tool reports high similarity score for the comparison (ignoring potential false positives), then in some sense the differ has successfully undone the obfuscation effect. Most historical work on deobfuscation known to us uses similarity-based metrics to evaluate the effectiveness of their techniques [62], [21], [75].

Binary similarity can be calculated based on either syntax or semantics. The syntax mostly refers to the control

4. We choose 30% because it is a more realistic configuration in practice. In case that readers are interested in BABEL's potency at other other obfuscation levels, please refer to an extended version of this paper [67].

TABLE 2. PROGRAM COMPLEXITY BEFORE AND AFTER BABEL OBFUSCATION AT 30% OBFUSCATION LEVEL

Program	# of Call Graph Edges			# of CFG Edges			# of Basic Blocks			Cyclomatic Number			Knot Count		
	Original	BABEL	Ratio	Original	BABEL	Ratio	Original	BABEL	Ratio	Original	BABEL	Ratio	Original	BABEL	Ratio
bzip2	353	5964	16.9	5382	19771	3.7	3528	17078	4.8	1856	2695	1.5	3120	12396	4.0
mcf	78	5449	69.9	854	14233	16.7	583	13086	22.4	273	1149	4.2	153	8792	57.5
regexp	72	5276	73.3	855	13290	15.5	591	11802	20.0	266	1490	5.6	1135	9530	8.4
svm	511	6739	13.2	5375	20752	3.9	3545	18533	5.2	1832	2221	1.2	2972	11521	3.9
oftpd	455	5810	12.8	2035	15501	7.6	1667	14422	8.7	370	1081	2.9	1277	9911	7.8
mongoose	1027	6762	6.6	2788	17115	6.1	2086	16079	7.7	704	1038	1.5	493	9491	19.3
Geom.Mean.	279.2	5972.7	21.4	2220.4	16555.5	7.5	1570.2	14987.8	9.5	633.0	1502.4	2.4	1002.3	10199.1	10.2

TABLE 3. PROGRAM COMPLEXITY BEFORE AND AFTER CODE VIRTUALIZER (CV) OBFUSCATION AT 30% OBFUSCATION LEVEL

Program	# of Call Graph Edges			# of CFG Edges			# of Basic Blocks			Cyclomatic Number			Knot Count		
	Original	CV	Ratio	Original	CV	Ratio	Original	CV	Ratio	Original	CV	Ratio	Original	CV	Ratio
bzip2	353	261	0.7	5382	3868	0.7	3528	2826	0.8	1856	1044	0.6	3120	713	0.2
mcf	78	34	0.4	854	461	0.5	583	329	0.6	273	134	0.5	153	68	0.4
regexp	72	66	0.9	855	525	0.6	591	377	0.6	266	150	0.6	1135	603	0.5
svm	511	357	0.7	5375	3267	0.6	3545	2358	0.7	1832	911	0.5	2972	302	0.1
oftpd	455	390	0.9	2035	1727	0.8	1667	1435	0.9	370	294	0.8	1277	542	0.4
mongoose	1027	585	0.6	2788	2063	0.7	2086	1638	0.8	704	427	0.6	493	442	0.9
Geom.Mean.	279.2	190.4	0.7	2220.4	1489.0	0.6	1570.2	1117.0	0.7	633.0	365.9	0.6	1002.3	358.3	0.3

flows of the binary and syntax-based binary diffing usually takes a graph-theoretic approach which compares the call graphs of two binaries and further the control flow graphs of pairs of functions between two binaries, looking for any graph or subgraph isomorphism. The intuition is, if two binaries have similar call graphs, the functions located at corresponding nodes in the call graph isomorphism are likely to be similar ones; if two functions have similar control flows, they are likely to implement the same computation logic.

On the other hand, semantics-based binary diffing focuses more on the observable behavior of the binaries. There are various ways to describe program behavior, e.g., the post- or pre-condition of a given chunk of code and certain effects the code commits such as system calls. If two binaries have matched behavior, a semantics-based binary diffing tool will consider them similar.

In general, syntax-based similarity is less strict than semantics-based similarity. Relatively, syntax-based differs tend to report more false positives while semantics-based differs tend to get more false negatives. To avoid bias as much as possible in the evaluation, we pick binary differs of both kinds to test BABEL's resilience to reverse engineering. We employ CoP [44] and BinDiff [2], of which CoP is a semantics-based binary differ and BinDiff is syntax based [65]. To measure BABEL's resilience to a differ, we randomly pick 50% functions from each program in Table 1, obfuscate them with BABEL, and then launch the differs to calculate the similarity between the original and obfuscated functions.

7.2.1. Resilience to Semantics-Based Binary Diffing. CoP, a "*semantics-based obfuscation-resilient*" binary similarity detector [44], is currently one of the state-of-the-art semantics-based binary diffing tools. The detection algorithm of CoP is founded on the concept of "longest common subsequence of semantically equivalent basic blocks." By constructing symbolic formulas to describe the input-output relations of basic blocks, CoP checks the semantic equivalence of two basic blocks with a theorem prover. It is reported that this new binary diffing technique can defeat many traditional obfuscation methods. CoP is built upon several cutting-edge techniques in the field of reverse engineering, including the binary analysis toolkit BAP [14] and the constraint solver STP [29]. CoP defines the similarity score as the number of matched basic blocks divided by the count of all basic blocks in the original function.

Fig. 9 is the box plot showing the distribution of similarity scores. For all programs, the third quartile of the scores is below 20%. Considering that the original paper of CoP reports over 70% similarity in most of their tests on transformed or obfuscated programs, the scores calculated from BABEL-obfuscated functions are not convincing evidence of similarity. One may notice that there are a few outliers in Fig. 9, i.e., the similarity scores for some functions can reach 100%. These functions are all "simple" ones, namely they have only one basic block and very few lines of C code. With the presence of false positives, it is very likely that the binary differ can report 100% similarity for these functions, according to CoP's similarity score definition.

7.2.2. Resilience to Syntax-Based Binary Diffing. BinDiff is a proprietary syntax-based binary diffing tool which is the de facto industrial standard with wide availability. It has motivated the creation of several academia-developed binary differs such as BinHunt [30] and its successor iBinHunt [50].

Given two binaries, BinDiff will give a list of function pairs that are considered similar based on a set of different algorithms. In addition to the similarity level like CoP reports, BinDiff also reports its "confidence" on the results, based on which algorithm is used to get that score. It is not completely clear to us how each of BinDiff's

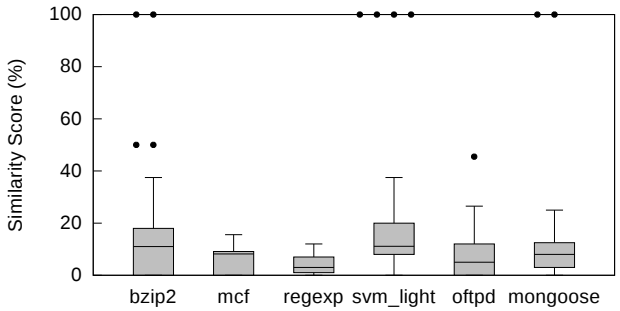

Figure 9. Distributions of similarity scores between the original and BABEL-obfuscated functions in the evaluated programs.

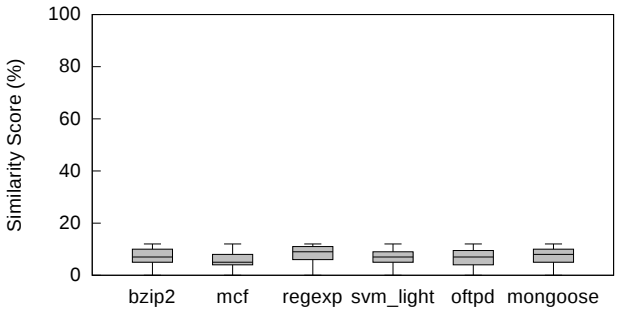

Figure 10. Distributions of similarity scores between the original and CV-obfuscated functions in the evaluated programs.

TABLE 4. FUNCTION MATCHING RESULT FROM BINDIFF ON BABEL-OBFUSCATED PROGRAMS

Program	# of Obfuscated	# of Matched	Match Rate
bzip2	54	6	11.11%
mcf	13	7	53.85%
regexp	11	3	27.27%
svm_light	52	10	19.23%
oftpd	48	4	8.33%
mongoose	102	39	38.24%
Overall	280	69	24.64%

TABLE 5. FUNCTION MATCHING RESULT FROM BINDIFF ON CV-OBFUSCATED PROGRAMS

Program	# of Obfuscated	# of Matched	Match Rate
bzip2	54	7	12.96%
mcf	13	4	30.77%
regexp	11	0	0.00%
svm_light	52	1	1.92%
oftpd	48	2	4.17%
mongoose	102	11	10.78%
Overall	280	25	8.93%

algorithms works and how BinDiff ranks the confidence level. Therefore, we report how many obfuscated functions are correctly matched to their originals by BinDiff regardless of the similarity score and the confidence. This makes sure BABEL does not take any unfair advantage over its opponent in the evaluation of performance. In other words, the results reported here indicate a lower bound of BABEL's resilience to syntax-based binary diffing.

Table 4 shows how many obfuscated functions in each program are matched (although some of them get low similarity scores or confidence). Since BinDiff can match functions solely based on their coordinates in the call graphs, two functions can be matched even if they have totally different semantics. This explains why BinDiff can achieve a relatively high matching rate for mongoose, because mongoose has the largest number of functions and potentially has a more iconic call graph. Nevertheless, only 26.22% of the obfuscated functions are matched by BinDiff over all six programs. Note that matching does not yet imply successful deobfuscation or recovery of program logic, especially for syntax-based binary differs. In that sense, we believe BABEL's performance is satisfying.

7.2.3. Comparing BABEL with Code Virtualizer. We present Code Virtualizer's resilience to CoP and BinDiff in Fig. 10 and Table 5, respectively. The data are obtained in experiments of which the settings are consistent with the resilience evaluation on BABEL. Based on the data, it seems that Code Virtualizer is more resilient to CoP and BinDiff than BABEL. However, as aforementioned in the potency evaluation, reverse engineering binaries protected by virtualization-based obfuscators like Code Virtualizer requires specialized approaches. Since neither CoP nor BinDiff is made aware of the fact that the obfuscated parts of the binaries have been transformed from code to data, their poor performance is not a surprising result. After all, a major weakness of virtualization-based obfuscation is that although the original program may be well obfuscated, the virtual machine itself is still exposed to attacks. By reverse engineering the logic of the virtual machine and revealing the encoding format of the bytecode, attackers can effectively deobfuscate the protected binaries [62], [75].

7.3. Cost

We measure execution slowdown introduced by BABEL from the obfuscation level of 10% to 50%. We use the test cases shipped with the obfuscated software as the performance test input for CPU-bound applications. For FTP server oftpd, we transfer 10 files ranging from 1KB to 128MB. For HTTP server mongoose, we sequentially send 100 quests for a 2.5KB HTML page. We conduct the experiments on a desktop with Xeon E5-1607 3.00GHz CPU and 4GB memory running 64-bit Ubuntu 12.04 LTS, over a 1Gbps Ethernet link. We run each test 10 times and report the average slowdown. For servers, time spent on network communication is included by our measurement.

Unlike potency and resilience, we can easily conduct a fair comparison between BABEL and Code Virtualizer on execution overhead. In the comparison, we configure Code Virtualizer to minimize obfuscation strength and maximize execution speed. The implication of our comparison setting is that, if BABEL can achieve comparable or better performance than a mature commercial product, the runtime overhead introduced by BABEL should be acceptable for practical use. To show that the functions we obfuscate are

TABLE 6. TIME OVERHEAD INTRODUCED BY BABEL AND CODE VIRTUALIZER (CV)

Program	10% obfuscated			20% obfuscated			30% obfuscated			40% obfuscated			50% obfuscated		
	Coverage	Slowdown		Coverage	Slowdown		Coverage	Slowdown		Coverage	Slowdown		Coverage	Slowdown	
		BABEL	CV		BABEL	CV		BABEL	CV		BABEL	CV		BABEL	CV
bzip2	0.00%	1.5	1.9	0.00%	1.5	1.9	27.78%	8.0	×	33.34%	100.9	×	33.34%	105.8	×
mcf	0.66%	1.0	12.0	4.30%	6.9	52.4	15.54%	65.8	×	69.33%	135.9	×	83.33%	169.3	×
regexp	18.33%	138.5	660.9	19.74%	173.8	834.8	19.74%	198.7	1122.2	28.91%	285.9	×	28.91%	288.8	×
svm_light	13.33%	1.0	58.0	20.00%	4.0	×	26.67%	4.1	×	26.67%	4.1	×	33.33%	11.8	×
oftpd	-	1.0	1.0	-	1.1	1.1	-	1.1	1.1	-	1.1	×	-	1.1	×
mongoose	-	1.0	1.4	-	1.2	8.8	-	1.6	9.3	-	1.7	×	-	2.1	×

Coverage denotes the percentage of CPU time taken by the obfuscated functions in the execution of the original programs (not available for IO-bound servers). The percentage provided is a lower bound because some functions may be inlined into others. × indicates that the corresponding test failed due to program crash or incorrect output.

non-trivial, we use `gprof` to get their performance coverage, i.e., the percentage of CPU time taken by the obfuscated functions in the execution of the original programs. Note that the percentage we obtain is just a lower bound because some functions may be inlined and `gprof` will contribute their execution time to other functions. We are only able to get the coverage data for the four CPU-bound applications, because the CPU time spent by the two original server programs is too short for meaningful profiling. Profiling server programs usually requires dedicated profilers and we are unaware of the existence of such tools for the two server programs we picked.

Table 6 gives the experiment results, showing that BABEL outperforms Code Virtualizer in most of the cases we tested. In particular, BABEL's obfuscation is more reliable in the sense that the obfuscated programs exit normally and give correct output on test input, while Code Virtualizer fails to provide reliable obfuscation on most of the tested programs when obfuscation level reaches 40%. Both BABEL and Code Virtualizer impose considerably high performance overhead after the obfuscation level reaches 30%, for many of the CPU-bound applications. This is expected, because in general such heavy-weight obfuscation methods should be avoided when protecting program hot spots [7]. In the evaluation, the coverage of many applications exceeds 30% after the obfuscation level reaches 50%, which is rarely the case if BABEL and Code Virtualizer are to be deployed in practice. Regardless, the key point of this evaluation is to demonstrate that BABEL's performance cost is lower than an industry-quality obfuscator which shares certain similarity with BABEL.

7.4. Stealth

By evaluating stealth we investigate whether BABEL introduces abnormal statistical characteristics to the obfuscated code. In the stealth evaluation, we pick the 30% obfuscation level.

Some previous work measures obfuscation stealth by the byte entropy of program binaries [73], for byte entropy has been used to detect packed and encrypted binaries [45]. Since BABEL does not re-encode original binary code, possessing normal byte entropy may not be a strong evidence of stealth. Therefore, we employ another statistical feature, the distribution of instructions, to evaluate BABEL. This

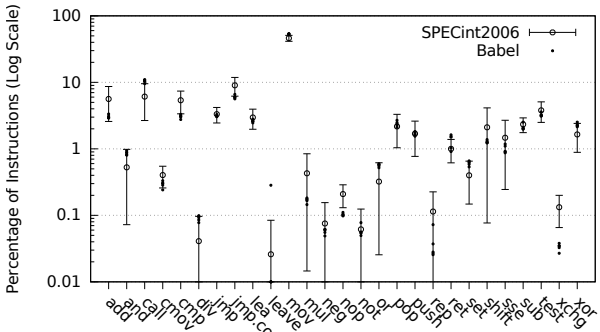

Figure 11. Instruction distributions of SPECint2006 programs (mean and standard deviation) and BABEL-obfuscated integer programs.

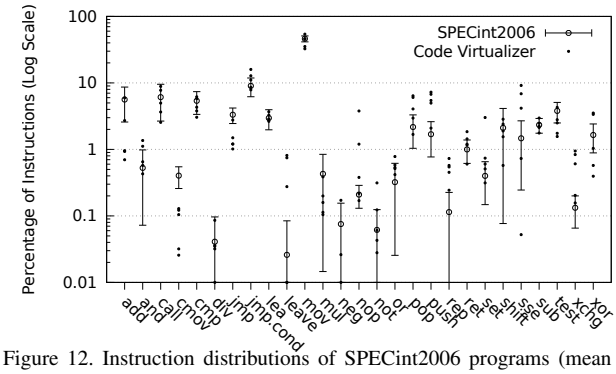

Figure 12. Instruction distributions of SPECint2006 programs (mean and standard deviation) and CV-obfuscated programs

metric has also been employed by previous work [54], [16]. To tell whether BABEL-obfuscated programs have abnormal instruction distributions, we need to compare them with normal programs. Since the scale of programs used in our evaluation is relatively small, we select the SPEC2006 benchmarks as the representatives of normal programs. We group common x86 instructions into 27 classes and calculate the means and standard deviations of percentages for each group within SPEC2006 programs. Since integer programs and floating-point programs have different distributions, we only compare bzip2, mcf, regexp, oftpd, and mongoose with SPECint2006.

Fig. 11 presents the comparisons for integer programs. For the majority of instruction groups, their distributions in BABEL-obfuscated programs fall into the interval of

normal means minus/plus normal standard deviations. There are some exceptions such as `mov`, `call`, `ret`, `cmp`, and `xchg`. However, their distributions are still bounded by the minimum and maximum of SPEC distributions (not shown in the figure). Hence, we believe these exceptions are not significant enough to conclude that BABEL-obfuscated programs are abnormal in terms of instruction distribution.

Meanwhile for binaries obfuscated by Code Virtualizer, the instruction distributions are significantly more deviant, as shown by Fig. 12. It should be noted that when we tried to disassemble binaries processed by Code Virtualizer, the disassembler reports hundreds of decoding errors, presumably because Code Virtualizer transforms legal instructions to bytecode which cannot be correctly decoded by the disassembler. Nevertheless, this disassembly anomaly itself can also be strong evidence of obfuscation. Overall, the experiments indicate that BABEL has better stealth performance than Code Virtualizer.

There may be of a concern that solely the existence of a Prolog execution environment can be the evidence of obfuscation. This can be tackled by developing a customized Prolog engine. Previous work has shown that a Prolog engine can be implemented with less than 1,000 lines of Pascal or C code [37], [72].

8. Discussion

8.1. Generalizing Translingual Obfuscation

Although it is usually quite challenging to translate programs in one language to another language with very different syntax, semantics, and execution models, many of the obstacles can be circumvented when the translation is for obfuscation purposes and not required to be complete. In our translation from C to Prolog, we designate the task of supporting C memory model, which is one of the most challenging issues in translating C, partially to the C execution environment itself. This solution is not feasible in general-purpose language translations. Meanwhile, some of our translation techniques are universally applicable to a class of target languages that share certain similarities. For example, the control flow regularization methods we proposed can be adopted when translating C to many declarative programming languages. We believe that translingual obfuscation has the potential to be made a general framework that supports various source and target languages.

8.2. Multithreading Support

Our current implementation of BABEL does not support C multithreading, and the main reason is that some components of GNU Prolog are not thread safe. Since GNU Prolog is a Prolog implementation for research and educational use, some language features are not supported. However, many other Prolog implementations that are more mature can indeed support multithreading well [8]. By investing enough engineering effort, we should be able to improve the implementation of GNU Prolog and ensure that it supports concurrent programming. Therefore, we do not view the current limitation as a fundamental one.

8.3. Randomness

Some obfuscation techniques improve the security strength by introducing randomness. For example, the virtualization-based obfuscators usually randomize the encoding of their virtual instruction set [28] so that attackers cannot crack all randomized binaries by learning the encoding of a single instance. Although this randomization is ineffective once attackers learned how to systematically crack the virtual machine itself, the idea of randomization does have some value.

Our current design of translingual obfuscation does not explicitly feature any randomness. However, since translingual obfuscation is orthogonal to existing obfuscation techniques, it can be stacked with those techniques that do introduce randomness. Translingual obfuscation itself has the potential to feature randomness as well. One promising direction could be making some of the foreign language compilation strategies undeterministic. Previous research [27] has shown that mutating compilation configurations can effectively disrupt some deobfuscation tools.

8.4. Defeating Translingual Obfuscation

In general, translingual obfuscation is open design and does not rely on any secrets, although it can be combined with other secret-based obfuscation methods. All of our justification on the security strength of translingual obfuscation assumes that attackers do possess the knowledge that we have translated C into Prolog. Indeed, with this knowledge attackers can choose to convert the binary to Prolog first rather than directly getting back to C. Either way, attackers will face severe challenges.

We would like to emphasize again that we do not argue it is impossible to defeat translingual obfuscation. Instead, we argue that Prolog is more difficult to crack than C, in the translingual obfuscation context. As long as a BABEL-translated Prolog predicate is compiled as native code, recovering it to a high-level program representation faces all the difficulties encountered in C reverse engineering, including the hardness of disassembly and analysis [68]. In §3 we revealed the deep semantics gap between Prolog source code and its low-level implementation. Thanks to this gap, we expect that recovering the computation logic of native code compiled from Prolog-translated C source code will consume a significant amount of reverse engineering effort.

What makes defeating translingual obfuscation even more challenging is that, the obfuscated code is not only a plain combination of normal Prolog plus normal C but a tangled mixture of both. The execution of obfuscated programs will switch back and forth between the two language environments and there will be frequent interleaving

of different memory models (see §5). This also imposes challenges to reverse engineering.

There is another point that grants translingual obfuscation the potential to significantly delay reverse engineering attacks. As stated in §2.3, translingual obfuscation is not limited to Prolog. There are many other programming languages that we can misuse for protection. By mixing these languages in a single obfuscation procedure, the difficulty of reverse engineering will be further increased.

9. Related Work

9.1. Programming Language Translation

People seek to translate one programming language to another, especially from source to source, for portability, re-engineering, and security purposes. The source-to-source translation from C/C++ to Java is one of the most extensively explored topics in this field, leading to tools such as C2J [40], C++2Java [3], and Cappuccino [15], etc. Trudel et al. [66] developed a converter that translates C to Eiffel, another object-oriented programming language. A tool called Emscripten can translate LLVM intermediate representation to JavaScript [76]. Since C/C++/Objective-C source code can be compiled into LLVM intermediate representation, Emscripten can also be used as a source-to-source translator without much additional effort. The C-to-Prolog translation introduced in this paper is partial since we need to keep the original C execution environment; however, our translation is for software obfuscation and being partial is not a limitation. Instead, we show that for our purpose, many technical issues commonly seen in programming language translation can be either addressed or circumvented.

9.2. Obfuscation and Deobfuscation

Software obfuscation can be on either source level or binary level. For source code obfuscation, Sharif et al. [61] encrypted equality conditions that depend on input data with some one-way hash functions. The evaluation shows that it is virtually impossible to reason about the inputs that satisfy the equality condition with symbolic execution. Moser et al. [51] demonstrate that opaque predicates can effectively hide control transfer destination and data locations from advanced malware detection techniques.

Obfuscation-oriented program transformations can also be performed at the binary level. Popov et al. [54] obfuscate programs by replacing control transfers with exceptions, implementing real control transfers in exception handling code, and inserting redundant junk transfers after the exceptions. Mimimorphism [73] transforms a malicious binary into a mimicry benign program, with statistical and semantic characteristics highly similar to the mimicry target. As a result, obfuscated malware can successfully evade statistical anomaly detection. Chen et al. [16] propose a control-flow obfuscation method making use of Itanium processors' architectural support for information flow tracking. In detail, they utilize the deferred exception tokens in Itanium processor registers to implement opaque predicates. Domas [25] developed a compiler which generates a binary employing only the `mov` family instructions, based on the fact that x86 `mov` is Turing complete. There are other binary obfuscation methods which heavily relies on compression, encryption, and virtualization [35], [47], [57]. Among these obfuscation techniques, binary packers using compression and encryption can be vulnerable to dynamic analysis because the original code has to be restored at some point of program execution. As for virtualization-based obfuscation, most current approaches are implemented in the decode-dispatch scheme [63]. Recent effort [56], [62] has identified the characteristics of the decode-dispatch pattern in the virtualization-obfuscated binaries so that they can be effectively reverse engineered.

As for deobfuscation, most recent work focuses on attacking virtualization-based obfuscation. Sharif et al. [62] has developed an outside-in approach which first reverse engineers the virtual machine and then decodes the bytecode to recover the protected program. Another deobfuscation method presented by Coogan et al. [21] chooses the inside-out method which utilizes equational reasoning to simplify the execution traces of protected programs. In this way, the deobfuscator extracts instructions which are truly relevant to program logic. A very recent method proposed by Yadegari et al. [75] improved the inside-out approach with more generic control flow simplification algorithms that can deobfuscate programs protected by nested virtualization. Without access to these tools, we cannot directly test BABEL's resilience to them. However, since BABEL completely reforms C programs' data layout and reconstructs the control flows with a much different programming paradigm, we are very confident with BABEL's security strength against these approaches.

Binary diffing is another widely used reverse engineering technique that takes program obfuscation into account. Binary differs identify the syntactical or semantic similarity between two different binaries, and can be used to detect programming plagiarism and launch similarity-based attacks [13]. BinDiff [2] and CoP [44], the two differs we use for evaluating BABEL's resilience, are currently the state of the art. Other examples of binary differs include DarunGrim2 [5], Bdiff [1], BinHunt [30], and iBinHunt [50]. Although these tools can defeat certain types of program obfuscation, none of them are designed to handle the complexity of translingual obfuscation.

10. Conclusion

In this paper, we present translingual obfuscation, a new software obfuscation scheme based on translations from one programming language to another. By utilizing certain design and implementation features of the target language, we are able to protect the original program against reverse engineering. We implement BABEL, a tool that translates part of a C program into Prolog and utilizes Prolog's unique language features to make the program obscure. We evaluate BABEL with respect to potency, resilience, cost, and stealth

on real-world C programs of different categories. The experimental results show that translingual obfuscation is an adequate and practical software protection technique.

Acknowledgments

We thank Herbert Bos and the anonymous reviewers for their valuable feedback which has greatly helped us improve the paper. This research was supported in part by the National Science Foundation (NSF) grants CNS-1223710 and CCF-1320605, and the Office of Naval Research (ONR) grant N00014-13-1-0175.

References

[1] "Binary Diff (bdiff)," http://sourceforge.net/projects/bdiff/.

[2] "BinDiff," http://www.zynamics.com/bindiff.html.

[3] "C++ to Java converter," http://www.tangiblesoftwaresolutions.com/Product_Details/CPlusPlus_to_Java_Converter_Details.html.

[4] "Code Virtualizer: Total obfuscation against reverse engineering," http://oreans.com/codevirtualizer.php.

[5] "DarunGrim: A patch analysis and binary diffing tool," http://www.darungrim.org/.

[6] "IDA: About," https://www.hex-rays.com/products/ida/.

[7] "Protecting better with Code Virtualizer," http://www.oreans.com/Release/ProtectBetter.pdf.

[8] "SWI-Prolog manual," http://www.swi-prolog.org/man/threads.html.

[9] "VMProtect software protection," http://vmpsoft.com.

[10] A. W. Appel and D. B. MacQueen, "Standard ML of New Jersey," in *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming*, ser. PLILP '91, 1991.

[11] H. At-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[12] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," *J. ACM*, vol. 59, no. 2, pp. 6:1–6:48, May 2012.

[13] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proceedings of the 29th IEEE Symposium on Security and Privacy*, ser. SP '08, 2008.

[14] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV '11, 2011, pp. 463–469.

[15] F. Buddrus and J. Schödel, "Cappuccino—A C++ to Java translator," in *Proceedings of the 1998 ACM Symposium on Applied Computing*, ser. SAC '98, 1998, pp. 660–665.

[16] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P.-C. Yew, "Control flow obfuscation with information flow tracking," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '42, 2009, pp. 391–400.

[17] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," The University of Auckland, Tech. Rep., 1997.

[18] ——, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '98, 1998.

[19] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation—tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, Aug. 2002.

[20] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*. Benjamin-Cummings Publishing Co., Inc., 1986.

[21] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: A semantics-based approach," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11, 2011, pp. 275–284.

[22] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008, pp. 255–266.

[23] E. D. Demaine, "C to Java: converting pointers into references," *Concurrency: Practice and Experience*, vol. 10, no. 11-13, pp. 851–861, 1998.

[24] D. Diaz, S. Abreu, and P. Codognet, "On the implementation of GNU Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 253–282, 2012.

[25] C. Domas, "Turning 'mov' into a soul-curshing RE nightmare," in *Proceeding of the 2015 Annual Reverse Engineering and Security Conference*, ser. REcon '15, 2015.

[26] W. Drewry and T. Ormandy, "Flayer: Exposing application internals," in *Proceedings of the 1st USENIX Workshop on Offensive Technologies*, ser. WOOT '07, 2007.

[27] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proceeding of the 23rd USENIX Security Symposium*, ser. USENIX Security '14, 2014, pp. 303–317.

[28] E. Eilam, *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

[29] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. CAV'07, 2007, pp. 519–531.

[30] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," in *Proceedings of the 4th International Conference on Information and Communications Security*, ser. ICICS '08, 2008.

[31] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, ser. FOCS '13, 2013, pp. 40–49.

[32] R. Ghiya and L. J. Hendren, "Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '96, 1996.

[33] S. Ghosh, J. Hiser, and J. W. Davidson, "Replacement attacks against vm-protected applications," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12, 2012, pp. 203–214.

[34] T. Gilb, *Software Metrics*, ser. Winthrop computer systems series. Winthrop Publishers, 1977.

[35] F. Guo, P. Ferrie, and T. Chiueh, "A study of the packer problem and its solutions," in *Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID '08, 2008, pp. 98–115.

[36] N. D. Jones and S. S. Muchnick, "A flexible approach to interprocedural data flow analysis and programs with recursive data structures," in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '82, 1982, pp. 66–74.

[37] S. N. Kamin, *Programming Languages: An Interpreter-Based Approach*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[38] N. Kettle and A. King, "Bit-precise reasoning with affine functions," in *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, ser. SMT '08/BPR '08, 2008, pp. 46–52.

[39] D. E. Knuth and R. W. Floyd, "Notes on avoiding 'go to' statements," *Information Processing Letters*, vol. 1, no. 1, pp. 23–31, 1971.

[40] C. Laffra, "C2J: A C++ to Java translator," *Advanced Java: Idioms, Pitfalls, Styles and Programming Tips*, 2001.

[41] J. Launchbury, "A natural semantics for lazy evaluation," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93, 1993, pp. 144–154.

[42] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03, 2003, pp. 290–299.

[43] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption," in *Proceedings of the 44th Annual ACM Symposium on Theory of Computing*, ser. STOC '12, 2012, pp. 1219–1234.

[44] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14, 2014, pp. 389–400.

[45] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 40–45, 2007.

[46] S. Marlow, A. R. Yakushev, and S. Peyton Jones, "Faster laziness using dynamic pointer tagging," in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '07, 2007.

[47] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *Proceedings of the 23rd Annual Computer Security Applications Conference*, ser. ACSAC '07, 2007, pp. 431–441.

[48] J. Martin and H. Muller, "Strategies for migration from C to Java," in *Fifth European Conference on Software Maintenance and Reengineering, 2001*, 2001, pp. 200–209.

[49] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976.

[50] J. Ming, M. Pan, and D. Gao, "iBinHunt: Binary hunting with inter-procedural control flow," in *Proceedings of the 15th Annual International Conference on Information Security and Cryptology*, ser. ICISC '12, 2012.

[51] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proceedings of the 23rd Annual Computer Security Applications Conference*, ser. ACSAC '07, 2007.

[52] J. Nagra and C. Collberg, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[53] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02, 2002.

[54] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in *Proceedings of 16th USENIX Security Symposium*, ser. USENIX Security '07, 2007.

[55] L. Ramshaw, "Eliminating go to's while preserving program structure," *Journal of the ACM*, vol. 35, no. 4, pp. 893–920, 1988.

[56] R. Rolles, "Unpacking virtualization obfuscators," in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, ser. WOOT '09, 2009.

[57] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, ser. ACSAC '06, 2006, pp. 289–300.

[58] M. Sagiv, T. Reps, and R. Wilhelm, "Solving shape-analysis problems in languages with destructive updating," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 1, pp. 1–50, Jan. 1998.

[59] A. Sahai and B. Waters, "How to use indistinguishability obfuscation: Deniable encryption, and more," in *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, ser. STOC '14, 2014, pp. 475–484.

[60] A. Sepp, B. Mihaila, and A. Simon, "Precise static analysis of binaries by extracting relational information," in *Proceedings of the 18th Working Conference on Reverse Engineering*, ser. WCRE '11, 2011, pp. 357–366.

[61] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proceedings of 15th Network and Distributed System Security Symposium*, ser. NDSS '08, 2008.

[62] ——, "Automatic reverse engineering of malware emulators," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, ser. SP '09, 2009, pp. 94–109.

[63] J. E. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[64] S.-Å. Tärnlund, "Horn clause computability," *BIT Numerical Mathematics*, vol. 17, no. 2, pp. 215–226, 1977.

[65] D. Thomas and R. Rolf, "Graph-based comparison of executable objects," in *Proceedings of the Symposium sur la Securite des Technologies de l'Information et des Communications*, ser. SSTIC '05, 2005.

[66] M. Trudel, C. Furia, M. Nordio, B. Meyer, and M. Oriol, "C to O-O translation: Beyond the easy stuff," in *Proceedings of the 19th Working Conference on Reverse Engineering*, ser. WCRE '12, 2012, pp. 19–28.

[67] P. Wang, S. Wang, J. Ming, Y. Jiang, and D. Wu, "Translingual obfuscation," 2016, arXiv:1601.00763 [cs.CR].

[68] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proceedings of the 24th USENIX Security Symposium*, ser. USENIX Security '15, 2015.

[69] D. H. D. Warren, "An abstract Prolog instruction set," SRI International, Tech. Rep. 309, October 1983.

[70] M. H. Williams and G. Chen, "Restructuring pascal programs containing goto statements," *The Computer Journal*, vol. 28, no. 2, pp. 134–137, 1985.

[71] M. Woodward, M. Hennell, and D. Hedley, "A measure of control flow complexity in program text," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 1, pp. 45–50, Jan. 1979.

[72] D. Wu, A. W. Appel, and A. Stump, "Foundational proof checkers with small witnesses," in *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, ser. PPDP '03, 2003, pp. 264–274.

[73] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang, "Mimimorphism: A new approach to binary code obfuscation," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10, 2010, pp. 536–546.

[74] B. Yadegari and S. Debray, "Bit-level taint analysis," in *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '14, 2014, pp. 255–264.

[75] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, ser. SP '15, 2015.

[76] A. Zakai, "Emscripten: An LLVM-to-JavaScript compiler," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '11, 2011, pp. 301–312.