Hiding Program Slices for Software Security*

Xiangyu Zhang Rajiv Gupta
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Abstract

Given the high cost of producing software, development of technology for prevention of software piracy is important for the software industry. In this paper we present a novel approach for preventing the creation of unauthorized copies of software. Our approach splits software modules into open *and* hidden *components. The open components are installed (executed) on an unsecure machine while the hidden components are installed (executed) on a secure machine. We assume that while open components can be stolen, to obtain a fully functioning copy of the software, the hidden components must be recovered. We describe an algorithm that constructs hidden components by slicing the original software components. We argue that recovery of hidden components constructed through slicing, in order to obtain a fully functioning copy of the software, is a complex task. We further develop security analysis to capture the complexity of recovering hidden components. Finally we apply our technique to several large Java programs to study the complexity of recovering constructed hidden components and to measure the runtime overhead introduced by splitting of software into open and hidden components.*

1. Introduction

Development of technology for the prevention of software piracy is important for the software industry. The advent of mobile computing will only make the problem of software piracy worse. In the future it will be commonplace for users to carry applications on a mobile computing device. However, these applications would often be transferred and executed on remote compute servers that would be part of the ubiquitous computing infrastructure in the future. In light of the greatly improved computing power of modern day processors, it is acceptable to expend a fraction of this computing power on protecting software. The goal of this work is to develop an approach to prevent malicious users of the software from creating fully functioning unauthorized copies of protected software.

We propose the development of a novel approach which splits software modules into *open* and *hidden* components. The open components can be installed and executed on an unsecure machine while the hidden components are installed on a secure machine. While open components can be stolen, they are *incomplete* (i.e., they only provide a subset of an applications functionality). The hidden components are constructed in a manner that causes a great deal of effort to be required in finding the missing hidden components by observing the code of the open component and its runtime interactions with the hidden component. Let us consider a couple of scenarios in which this approach can be used to provide software protection.

Untrustworthy User. Consider a very common setting in which a legally obtained software has been installed on client machines of an organization so that it can be freely used by all authorized users of the client machines within the organization. In this setting our aim is to prevent these authorized users from transferring the software to other machines outside the organization for unauthorized use. Typical solutions to preventing software piracy, namely using a serial number or an authenticating key, are really not applicable in this scenario since software is being stolen by an authorized user who already has access to such information. However, our proposed approach is effective in this scenario. While the open components are installed on the client machines, the hidden components can be installed on a secure device. For example, the hidden components can be installed on a smart card if they are sufficiently light weight and these secure smart cards can be issued to the users. If the hidden components are heavy weight, they can be installed on a secure server. The client machines must interact with the secure smart card or server to provide fully functioning software. In this scenario the theft of fully functioning software is clearly prevented as users cannot steal the hidden components.

*Supported by a grant from IBM and National Science Foundation grants CCR-0220262, CCR-0208756, CCR-0105535, and EIA-0080123 to the University of Arizona.

Untrustworthy Server. While the above scenario is common place today, in the future we expect to encounter the following scenario frequently. Users will carry mobile devices which will host the applications used by the user. When these applications are used, due to the limited computing ability, limited battery power, or communication bandwidth, the applications will be executed on remote servers. The servers will be considered untrustworthy and thus concern for piracy of software transferred to these servers for execution will arise. To counter these concerns our proposed approach is also effective. The hidden components will be constructed to be light weight so that they can be executed on the user's mobile device while the heavy weight open components can be transferred to remote servers for execution. Again while theft of open components is possible, the software is protected by preventing the theft of hidden components.

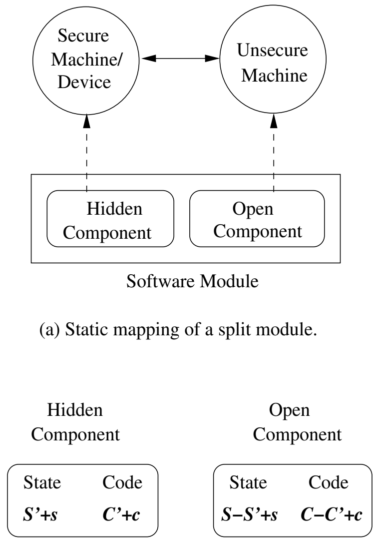
Given the above overall idea of software splitting, the key challenge of this work is to design a splitting transformation that simultaneously addresses two factors: *robustness* of the partitioning in protecting software and *cost* of splitting in terms of the runtime overhead it introduces. An attack designed to pirate a copy of the split software must find a way to predict contents of the hidden components by observing their dynamic input output behavior. Thus, the robustness of protection will depend upon the nature of splitting transformation used. For creating hidden components through splitting we employ program slicing techniques as slices can be chosen such that they do not perform a high level function that is easy to predict. The difficulty of recovering hidden components is measured by the computational complexity of attacks designed to recover hidden components.

The cost of transformed modules is also an important issue. There are two kinds of cost issues. The first type of cost is the communication costs between open and hidden modules which are important to consider because the hidden components will reside on a device that is different from the machine on which open components execute. The second cost issue arises due to the nature of the computing device on which the modules execute. In the "untrustworthy user" scenario the hidden modules may be executed on a smart card while in the "untrustworthy server" scenario the hidden modules execute on the user's mobile device. Thus, in these situations the hidden modules should be light weight computations that involve light weight communication with the open modules. Our splitting algorithm restricts the communication costs by placing restrictions on the type of code that can be placed in hidden components and the execution costs of hidden components by selecting a small subset of modules for splitting.

The remainder of the paper is organized as follows. In section 2 we motivate and present our splitting transformation based upon slicing. In section 3 we describe our approach for characterizing the degree of security provided by a splitting transformation. In section 4 we present the results of applying the proposed methods to several large Java programs. Related work is discussed in section 5. Conclusions are given in section 6.

2. The Splitting Transformation

The basic principle behind our approach is to automatically split critical software into two components for security: an *open* component and a *hidden* component. Let (S, C) denote the program's runtime state S and code C. The hidden component contains $(S' + s, C' + c)$ which represents part of the program's state as well as the code that maintains that state. The open component contains $(S - S' + s, C - C' + c)$ which is the remainder of program's runtime state and code. The (s, c) part represents additional variables and new code respectively that implement interactions between the components.



(a) Static mapping of a split module.

(b) Runtime state of a split module.

Figure 1. Software splitting.

The open component is installed locally on an unsecure machine while the hidden component is installed on a secure machine or device. The application is invoked through an open component; however, the open component must interact with the hidden component through remote procedure calls to function correctly. While an adversary trying to steal the software can copy the open components of the software, he has no access to the hidden components. Thus, to obtain a complete copy of fully functioning software, an adversary must study the interactions between the open and hidden components and attempt to construct the missing hidden code.

Table 1. Opportunities for constructing hidden components from whole methods.

	`jfig`	`jess`	`bloat`	`javac`	`jasmin`
Number of Methods	2987	1622	3839	1898	645
Self-contained Methods	21	6	35	16	7
Self-contained > 10	6	6	9	8	5
Excluding Initializers	0	0	1	8	3

2.1. Hiding Whole Modules

The most obvious approach for splitting a program into its hidden and open components is to simply select one or more complete modules and treat them as hidden components. An approach that constructs hidden components along programmer defined boundaries has a number of problems. Since a module can be expected to perform a coherent function, by examining the open part of the program, an adversary may be able to guess the function that is performed by the hidden module. For example, if function *pop()* is removed from the implementation of a *stack module*, by examining other functions that remain in the module the adversary may be able to guess that *pop()* is missing. Thus, the adversary may be able to create the functionality provided by the hidden components.

Let us assume that the adversary cannot guess the functionality of the module. In this case we still need to find a suitable module for hiding. We examined several large *Java* programs and examined the suitability of the *methods* present in them for hiding. For this purpose we defined the notion of a *self-contained* method. If the execution of a method on a secure device can be carried out by simply transferring a set of scalar values between the unsecure machine and the secure device, then we consider the method to be self-contained. Note that a self-contained method may access data that is not local to the method as such data can be passed to the hidden component in form of additional parameters. On the other hand any method that invokes other methods or operates on entire aggregates (e.g., arrays or other data structures) are considered not to be self-contained. The motivation of this choice should be clear - execution of self-contained functions on the secure device involves simple low cost interaction between the unsecure machine and the secure device.

As Table 1 shows, while each of the programs contain a large number of methods, the number of self-contained methods is quite small. If we further exclude small methods which contain no more than 10 *Java byte code* statements, the number falls even further. Finally if we also exclude methods that are simply initializers, as their behavior can be easily learned by observing their interaction with the open part of the program, the number of methods remaining is very small. For `jfig` and `jess` there is not a single method that is self-contained, is not an initializer, and is made up of more than 10 byte codes. Thus, it is clear that creating hidden components using entire methods is not a practical strategy.

2.2. Hiding Module Slices

For software splitting to be truly effective, the problem that must be solved is the splitting of the modules into open and hidden components in such a way that the functional behavior of the hidden components cannot be easily understood by examining the open components and tracing their runtime interaction with hidden components. We propose the construction of hidden components out of *program slices* such that their behavior cannot be easily understood. A program slice is composed of three types of entities: *variables*, *expressions and assignments*, and *control flow* statements. To understand why a program slice is a good candidate for forming a hidden component, let us see how the complexity of the hidden component increases with the inclusion of each of the above three types of entities.

Variables. Consider the splitting of function *f* that takes a subset of local variables belonging to *f* and creates a hidden component *Hf* which is responsible for maintaining the values of these variables. The remainder of the function performed by *f* forms the open component *Of*. We refer to the variables whose values are maintained by the hidden component as *hidden variables*. The variables in *f* that are selected to be hidden variables are replaced by single variable during the creation of *Of*. Interactions between *Of* and *Hf* are introduced to perform two functions. When *Of* computes a new value for a hidden variable v, the new value of v is sent to *Hf* so that the value of v can be correctly updated. When *Of* needs to use the value of v it receives the current value of v from *Hf*.

By statically examining *Of* it is hard to determine how many variables have been hidden in *Hf* as all references to hidden variables are replaced by a single variable in *Of*. However, through dynamic analysis the hidden components can be quite easily recovered. The adversary can observe the values being exchanged by *Of* and *Hf* over a period of time and start relating definitions in *Of* with uses in *Of*. The reason why this is possible is that the useful values returned by *Hf* to *Of* are always sent to *Hf* by *Of* at an earlier point in execution.

Expressions and Assignments. Since hiding variables alone is not enough, a subset of statements (expressions and assignments) that are involved in computing the values of hidden variables are also moved to the hidden component. The expressions and statements that are hidden include all those statements that belong to *forward data slices* constructed by following data dependence edges originating at definitions of hidden variables. Even if a single local variable from *f* is selected for hiding in *Hf*, additional variables may be *fully* or *partially* hidden in *Hf*.

Since, in addition to a number of variables being hidden, the computation of their values is also hidden, the useful values returned by *Hf* to *Of* are typically not identical to the values received by *Hf* from *Of* at earlier points in execution. Establishing relationship between these values is difficult since it is not known how many variables are being maintained by *Hf* and what is the form of expressions that relate these values. One can generate guesses for the form of the relationship and try to recover the precise relationship by observing the values exchanged between *Of* and *Hf*. The difficulty of this task depends upon the complexity of relationships.

Control Flow. To further increase the complexity of recovering hidden components, we can also hide part of the control flow in *f* by transferring control flow constructs, partially or fully, to *Hf*. We propose to achieve such hiding by moving the control ancestors of selected statements that belong to forward data slices of hidden variables. In particular, control ancestors are hidden if doing so will simultaneously introduce a control flow construct in the *Hf* and remove or alter the control flow in *Of*. For example, if all the statements that form a loop body are moved to *Hf*, then the enclosing looping construct may be moved to *Hf*. If all statements of an *else* clause have been moved to *Hf*, the predicate can also be moved to introduce control flow in *Hf*. In addition, the control flow construct *if-then-else* is replaced by construct *if-then* in *Of*.

While it may be reasonable to generate guesses for simple expressions that relate values sent to *Hf* by *Of* and values received by *Of* from *Hf*, if these relationships are made more complex by involving control flow and branch conditions, the task of recovering hidden components becomes even more difficult.

From the above discussion we see that one can expect that in general the complexity of recovering hidden components can be quite high if they are constructed by taking slices of original modules. However, additional issue of controlling the cost of splitting modules in terms of the runtime overhead they introduce must also be addressed.

Function Selection. The number of functions that are selected for splitting affects the overall cost. In some situations the software developer may identify the critical modules that must be protected. If such information is not available we propose the following strategy for selecting functions for splitting. We construct the call graph for the program and find a cut across the call graph. The functions that are part of the cut are split. This approach guarantees that during any execution at least some split function would be executed. We can also give preference to splitting functions that are not involved in direct or indirect recursion. The consequence of choosing a non-recursive function *f* is that only a single instance of *Hf* will exist at any time and thus the storage required for execution of *Hf* can be allocated statically. If a recursive function is split, then multiple instances of the function, and hence its hidden component, will exist simultaneously. To distinguish between these instances, an *instance id* is introduced so that only the open and hidden components that correspond to each other interact.

To further ensure that the overhead of executing split functions is not high, we restrict the selection of a function *f* for splitting and the manner in which it is split as follows.

- In constructing a cut through the call graph we avoid functions that are called from inside a loop. This restriction avoids splitting functions that are called repeatedly.
- No function calls made by *f* are hidden in *Hf*. If *Hf* contains no function calls, then there will be no need to replicate the environment in which the called functions must execute.
- Finally only scalar variables local to *f* are considered as candidate hidden variables, that is, aggregate data structures such as arrays are not hidden in *Hf*. This restriction is made to limit the amount of storage needed by the hidden components and the amount of communication between the open and hidden components.

Function Splitting Details. Let us assume that we have selected a function *f* and a local variable *v* in *f* for splitting of *f*. The hidden component *Hf* is constructed such that it consists of a set of code fragments removed from *f* and each of these fragments is identified by a unique label. The execution of statements placed in *Hf* is triggered by placing calls in *Of* at points from where they are removed. The function *Hf* has two parameters, a label *id* that identifies the statements in *Hf* that needs to be executed and an array which contains values from *Of* which are needed by *Hf* to perform the computation. *Hf* also returns a single value which may be the value needed by *Of* to continue execution. Below we summarize the steps of generating *Hf* and *Of* from a given function *f*.

Step 1 constructs the program slice *Slice(f,v)* starting from the statements that define v.

Step 2 examines the statements in *f* and *Slice(f,v)* to determine the set of fully and partially hidden variables.

Step 3 examines each statement in *Slice(f,v)* and splits it between *Of* and *Hf*. There are several cases that are considered for a statement $lhs \leftarrow rhs$: (i) both lhs and rhs are placed in *Hf*; (ii) only the lhs is placed in *Hf* because the rhs is an entity that cannot be placed in *Hf* (e.g., a function call); (iii) only the rhs is placed in *Hf* because the lhs variable cannot be placed in *Hf* (e.g., it is an array reference); and (iv) neither lhs or rhs can be placed in *Hf* and thus the statement is simply left in *Of*. When control flow is being transferred to *Hf* then all statements belonging to the control flow construct that are being transferred are moved as a single unit to *Hf*.

Step 4 examines all statements that are not in *Slice(f,v)*, but contain a reference, definition or use, to a partially transferred variable. If the variable on the lhs (say x) is partially hidden, then the expression rhs is evaluated in *Of* and the new value of variable x is sent to *Hf* so that it can be updated. If a partially hidden variable x is referenced by the rhs, then preceding the statement $lhs \leftarrow rhs$ in *Of*, a call to *Hf* is introduced to get the value of x.

While we have discussed module splitting in context of splitting a function *f* by hiding local variables of *f* in *Hf*, our approach is more broadly applicable. Global program variables can also be hidden in *Hf*. Simple modifications to the above algorithm accomplish this task. We can select a global variable for hiding and then identify all statements in each of the functions that refer to the global variable. If a function meets the characteristics outlined earlier, then slices starting from statements referring to the selected global variable are computed for transfer to *Hf*. Thus, essentially the algorithm described is applied to each of the functions that refers to the selected global variable. On the other hand, if the function does not meet the required characteristics, it is not sliced. Instead corresponding to each reference to the global variable, an appropriate call to a hidden function is made either to update the value of the global variable on the hidden side or fetch its value for use in the open side.

Our approach is also applicable to object oriented software. Consider a *class* which contains data in form of *class fields* and code in form of *class methods*. If we want to simply split a selected *method*, we can select one of its local variables to initiate splitting. On the other hand, in order to split the entire class into open and hidden components, we can view the class fields as globals and class methods as functions and apply the method for hiding global variables described above. A simple modification to the above algorithm is needed to handle multiple instances of the class that may be created by the program. Every time a class instance is created by the open component, a unique *instance id* is assigned to this instance. A call to the server side is made causing it to create a corresponding class instance which contains the *hidden class fields*. In addition, this instance is associated with the same *instance id*. Calls to *Hm*, where m is a method, include the *instance id* so that the hidden component located on the secure device can apply the hidden part of the method to the appropriate class instance. Our implementation of the proposed method, which is used in experiments described later, has been carried for object oriented programs written in *Java*.

An Example. The example in Fig. 2 illustrates our algorithm. We perform splitting of function *f* by deciding to transfer variable a to the hidden component *Hf*. We compute the forward data slice of a which causes transfer of several expressions and assignments to *Hf*. All expressions and assignments that are moved to *Hf* must only involve references to scalar variables. Thus, when we take a forward transitive closure over data dependences originating at the only definition of a, we terminate the slice at definitions of array elements as we do not transfer array elements to *Hf*. Once having identified the expressions and statements in a's forward data slice we observe that all statements in the body of the *while* loop have been included in the slice. Therefore, we include the entire *while* loop in the slice and thus hide the control flow associated with the loop. Similarly because all statements in the *then* clause of the *if-then-else* statement are included in the slice, we also include the branch condition in the slice.

The open and hidden components are then formed such that all statements in the slice are moved to *Hf* and all remaining statements remain in *Of*. It should also be noted that the transfer of the above statements causes additional variables (b, i, and sum) to be completely hidden in *Hf* as all their definitions are included in the slice and thus also moved to *Hf*. Calls to *Hf* are introduced in *Of* so that values are sent back and forth as needed by the statements in the two components. Note that in some cases, when *Of* calls *Hf*, no value needs to be returned to *Of*. Thus, an arbitrary value denoted as *any* is returned by *Hf* in this case.

3. Security Analysis

While the systems characterized in our motivating scenarios can be threatened in a number of ways and compromised at a number of points, our objective is to address the following threat. We would like to prevent an adversary from obtaining a complete copy of the software by recovering hidden components through observation of the runtime behavior of the open components. While in general it cannot be guaranteed that recovery of hidden components is

```
function f(···)
    int a, b, c, i, sum;
    int w, x, y, z ···;
    int[] A,[] B, ···;
    ...
    a ← 3x + y;
    ...
    b ← a + w;
    ...
    A[ b − 1 ] ← ···;
    ...
    if (y > 10) then
        c ← a · x + w;
    else
        c ← 2x + w;
    endif
    ...
    B[0] ←c + 1 ;
    ...
    i ← a;
    ...
    sum ← 0;
    while (i < z) do
        sum ← sum + i
        i ← i + 1;
    endwhile
    g(sum )
    ...
endfunction
```

```
function Of(···)
    int c, t;
    int w, x, y, z ···;
    int[] A,[] B, ···;
    ...
    t ← Hf([x, y], l1);
    ...
    t ← Hf([w], l2);
    ...
    A[ Hf([], l3) ] ← ···;   ①
    ...
    t ← Hf([y, x, w], l4);   ②
    if (t == 1) then
        c ← 2x + w;
        t ← Hf([c], l5);
    endif
    ...
    B[0] ←Hf([], l6);   ③
    ...
    t ← Hf([], l7);
    ...
    sum ← 0;
    t ← Hf([sum], l8)
    ...
    t ← Hf([z], l9)
    g(Hf([], l10);   ④
    ...
endfunction
```

```
int function Hf (int[] t, id)
        static int a, b, c, i, sum;
        switch id
            ...
            l1: a ← 3t[0] + t[1];
                return(any);
            l2: b ← a + t[0];
                return(any);
            l3: return(b − 1);
            l4: if (t[0] > 10) then
                    c ← a · t[1] + t[2];
                endif
                return((t[0] > 10)?0 : 1);
            l5: c ← t[0];
                return(any);
            l6: return(c + 1);
            l7: i ← a;
                return(any);
            l8: sum ← t[0];
                return(any);
            l9: while (i < t[0])
                    sum ← sum + i;
                    i ← i + 1;
                endwhile
                return(any);
            l10: return(sum);
            ...
        endswitch
endfunction
```

Figure 2. Splitting of function f initiated with slicing of variable a. The boxed statements in f form the slice. The execution of statements in the slice is partially or fully performed by Hf.

impossible to achieve, we can characterize the complexity of the effort required to recover hidden components.

We present a characterization of the complexity of hidden components which directly effects the complexity of the task of recovering hidden components. The recovery of a hidden component amounts to recovery of individual code fragments that make up the hidden component. To understand how these code fragments may be recovered, we first identify the information that can be collected to characterize their dynamic behavior. Each time the hidden component returns a value to the open component, it potentially provides some information on its behavior. In particular, if the hidden component returns a value that is used by the open component in its computations, the value should be collected for use in a procedure for recovering the hidden code. For example, if the hidden code that computes the returned value is a linear expression, linear regression may be employed to identify the code using the colleced values.

We define the notion of *information leak point* (ILP) to collect dynamic information on a hidden component. An ILP is a point in the open component at which part of the state of the hidden component is revealed. Thus, ILPs correspond to points in a open component at which values are returned by the hidden component for use in future computations by the open component. Such values are referred to as *leaked values*. The example in Fig. 2 contains four ILPs marked as ①, ②, ③, and ④ in the open component. Note that at all other calls to hidden component return useless values. The values returned at ILPs are dependent upon values of variables whose values are passed from the open to hidden components. Thus, identifying the hidden code that computes values leaked at a specfic ILP requires relating the values of the above mentioned variables with the values returned at the ILP point. If each of the ILPs are broken, i.e. the computations that they represent are recovered, we consider the hidden component as being recovered.

The complexity of recovering the code corresponding to each ILP can be characterized in terms of the complexity of the code itself. We characterize the code complexity in terms of its *arithmetic* and *control flow* complexities.

Arithmetic Complexity. Let us consider a path P in an open component along which an ILP is encountered. The arithmetic complexity of the function that relates the value returned at ILP point with variable values that are *observable* in the open component is one measure of the complexity of the code in the hidden component. A value is *observable* in the open component either because it is the result of a computation in the open component or it is leaked to the open component by the hidden component. Thus, given a leaked value

$$lv = f_{ILP}^{P}(observable\ values\ used)$$

we characterize the arithmetic complexity of f_{ILP}^{P}, denoted by $AC(f_{ILP}^{P})$, by a triple of the form

$$< Type, Inputs, Degree > .$$

In the above triple $Type$ is one of the following: *Constant* if f_{ILP}^{P} is a compile-time constant; *Linear* if f_{ILP}^{P} is a linear expression; *Polynomial* if f_{ILP}^{P} is a polynomial; *Rational* if f_{ILP}^{P} is a rational whose quotients are polynomials; and *Arbitrary* if f_{ILP}^{P} involves arithmetically more complex operators (e.g., exponential, log, mod) or non-arithmetic operators (e.g., boolean, relational). The $Inputs$ is the number of variables in the open component whose values are used by f_{ILP}^{P} and $Degree$ is the highest degree polynomial involved in f_{ILP}^{P} in case the $Type$ is not arbitrary.

Above we have defined the arithmetic complexity for an ILP with respect to a single path. We define the overall complexity of an ILP across all paths as the maximum arithmetic complexity observed across all paths, i.e.

$$AC(f_{ILP}) = \underset{\forall P}{MAX}(AC(f_{ILP}^{P}))$$

where the MAX is defined according to the partial order $Constant \prec Linear \prec Polynomial \prec Rational \prec Arbitrary$.

Control Flow Complexity. Control flow is also a major contributor to the complexity of the code corresponding to an ILP. Code involving multiple paths is more complicated to recover than straight line code. If the predicates that distinguish between the paths are hidden and/or the control flow itself is hidden (i.e., they are present in the hidden component but not in the open component) it makes the task of recovery even more complex. Thus, we characterize the control flow complexity of an ILP, denoted as $CC(f_{ILP})$ by the following triple:

$$< Paths, Predicates, Flow >$$

where $Paths$ is the number of paths specified as *constant* or a runtime *variable*, $Predicates$ may be all *open* or some may be *hidden*, and $Flow$ may be entirely *open* or partially/fully *hidden*.

Example of ILP Complexity Characterization. Given the above characterization of ILP complexity, the complexities of the three ILPs in Fig.2 are as follows:

①

$$f_{ILP} = b - 1 = a + w - 1 = 3x + y + w - 1$$
$$AC(f_{ILP}) =< Linear, 3, 1 >$$
$$CC(f_{ILP}) =< constant, -, - >$$

②

$$f_{ILP} = y > 10$$
$$AC(f_{ILP}) =< Arbitrary, 1, - >$$
$$CC(f_{ILP}) =< constant, -, - >$$

③

$$f_{ILP}^{TRUE} = c + 1 = ax + w = 3x^2 + y + w$$
$$f_{ILP}^{FALSE} = c + 1$$
$$AC(f_{ILP}) =< Polynomial, 3, 2 >$$
$$CC(f_{ILP}) =< constant, hidden, hidden >$$

④

$$f_{ILP} = sum + \sum_{i=3x+y}^{z-1} i$$
$$AC(f_{ILP}) =< Polynomial, 4, 2 >$$
$$CC(f_{ILP}) =< variable, hidden, hidden >$$

Algorithm for Estimating ILP Complexity. An algorithm for automatically computing the complexities of ILPs should be useful for choosing between splitting alternatives. While the computation of control flow complexity for an ILP is straightforward, the computation of arithmetic complexity requires some discussion. In our definition of arithmetic complexity we consider each relevant path P and derive precise arithmetic expression for an f_{ILP} along the path. In practice, due to the presence of loops and conditionals, considering each path is not realistic. Therefore we develop a simple algorithm which, under the assumption that no symbolic evaluation is performed to derive precise expressions, computes a conservative estimate (lower bound) for $AC(f_{ILP})$ without deriving precise expressions for f_{ILP} along different paths.

The detailed algorithm and an example illustrating it is given in Fig. 3. The example is a slightly modified version of the example in Fig. 2. The algorithm operates by propagating arithmetic complexities along def-use edges. Iterative analysis is needed due to loops in the data dependence graph created by loop carried data dependences. Given a statement $v = exp$, the arithmetic complexity of v is computed by function $EVAL$ that examines exp. The *propagated* arithmetic complexity of v to its uses varies. First if v's value is observable constant or variable, the propagated arithmetic complexity is set to *Constant* or *Linear* respectively. This value is further adjusted if v's value is prop-

Definitions:

v is *observable* at $s : v = exp$ iff either v is assigned in the open component or s is hidden but v's value at s is *definitely* leaked at a later use of v at n (we set $LeakedDefn(u_v@n) = s$).

$AC(d_v@n)$ - arithmetic complexity of definition of variable v by statement n.

$AC(u_v@n)$ - arithmetic complexity of use of variable v by statement n.

$EVAL(exp)$ - computes arithmetic complexity of expression exp based upon the operator used by exp and arithmetic complexities of operands used by exp.

$PC(d_v@n', u_v@n)$ - arithmetic complexity of variable v's value *propagated* along def-use edge $n' \rightarrow n$.

$Iter(L)$ - arithmetic expression for number of loop iterations of loop nest L in terms of observable values.

$RAISE(PC, Iter(L))$ - adjusts PC, the propagated complexity of a variable, when its value is propagated from inside a loop nest L to a use outside the loop nest L. This adjustment is made based upon the arithmetic complexity of $AC(Iter(L))$.

Iteratively Compute:

Given statement $n : a = b\ op\ c$

$AC(d_a@n) = EVAL(b\ op\ c)$

$AC(u_b@n) = \underset{\forall(d_b@n', u_b@n)}{MIN} PC(d_b@n', u_b@n)$

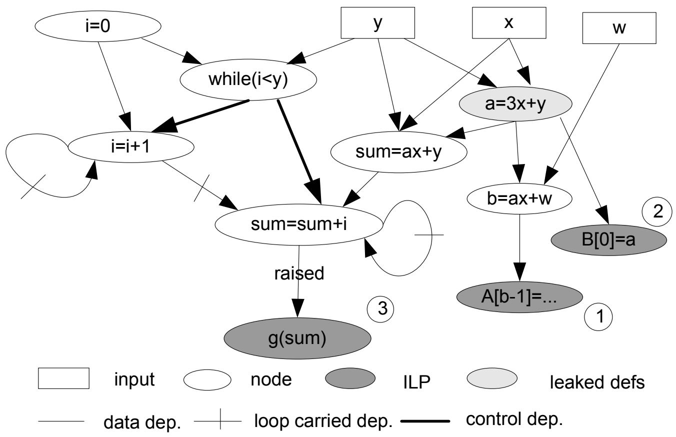
$$PC(d_b@n', u_b@n) = \begin{cases} < Constant, -, - > & if\ b's\ value\ at\ n'\ is\ observable\ and\ constant \\ < Linear, -, - > & elseif\ b's\ value\ at\ n'\ is\ observable\ but\ varying \\ AC(d_b@n') & otherwise \end{cases}$$

$PC(d_b@n', u_b@n) = RAISE(PC(d_b@n', u_b@n), Iter(L))$, if loop nest L must be exited for flow of value along def-use edge $(d_b@n', u_b@n)$

Output:

Given an ILP at statement s, ILP_s, where value of exp is leaked
if exp is v st $LeakedDefn(u_v@s)$ is $d : v = exp'$
then $AC(ILP) = AC(exp')$
else $AC(ILP) = AC(exp)$ endif

function f$(\cdots)$
$\boxed{int\ a, b, sum, i;}$
int w, x, y, $\cdots$;
int[] A,[] B, $\cdots$;
$\cdots$
$\boxed{a \leftarrow 3x + y;}$
$\cdots$
$\boxed{b \leftarrow ax + w;}$
$\cdots$
$\boxed{A[\ \mathbf{b\text{-}1}\ ] \leftarrow \cdots;}$ ①
$\cdots$
$\boxed{B[0] \leftarrow \mathbf{a}\ ;}$ ②
$\cdots$
$\boxed{sum \leftarrow ax + y;}$
$\cdots$
$i \leftarrow 0;$
while $(i < y)$
$\boxed{sum \leftarrow sum + i}$
$i \leftarrow i + 1$
endwhile
$\boxed{g(\mathbf{sum}\ )}$ ③
$\cdots$
endfunction



Inputs: $AC(y) = AC(x) = AC(w) =< Linear, 1, 1 >$

Statements: $AC(d_i@i = 0) =< constant, -, - >;$
$AC(d_i@i = i + 1) =< constant, -, - >;$
$AC(d_a@a = 3x + y) =< Linear, 2, 1 >;$
$AC(d_{sum}@sum = ax + y) =< Polynomial, 3, 2 >;$
$AC(d_{sum}@sum = sum + i) =< Polynomial, 3, 2 >;$
$AC(d_b@b = ax + w) =< Polynomial, 3, 2 >;$

ILPs: ①$AC(f_{ILP_1}) =< Polynomial, 3, 2 >$
②$AC(f_{ILP_2}) = AC(3x + y) =< Linear, 2, 1 >$
③$AC(f_{ILP_3}) =< Polynomial, 3, 2 >$

Figure 3. Estimating arithmetic complexity.

agated from inside a loop to outside the loop by $RAISE$ based upon the arithmetic complexity of expression corresponding to number of loop iterations. Thus, only if the value of v is not observable, and it is not being propagated along an edge from the inside of a loop to the outside, the propagated arithmetic complexity is same as the arithmetic complexity of v.

It should be noted that sometimes the value of v can be observable even though $v = exp$ is hidden because it may be leaked at a use of v. If there is a use of v, say use_v, in the open component such that every time this use is executed, we can conclude with certainty that the value of v at use_v came from a specific hidden definition of v, say $v = exp$, then we say that use_v *definitely* leaks the hidden definition $v = exp$. In the example shown in Fig. 3, the hidden definition of a by statement $a = 3x + y$ is definitely leaked by the use of a in $B[0] = a$. Finally, although not shown in the algorithm, in addition to arithmetic complexity *Type*, the identities of the variables and their degrees are also propagated along the edges. The latter are needed to compute the number of *Inputs* and the *Degree* components of arithmetic complexity.

Practical Limitations of Automated Recovery. Let us assume that no control flow is involved and thus breaking an ILP strictly requires recovery of a straightline code segment. Linear regression [12], polynomial interpolation [17], and rational interpolation [10] are known techniques that can be applied to recover a f_{ILP} of the corresponding arithmetic complexity. However, as far as we know, there are no automatic methods that can recover an *arbitrary* type f_{ILP}. Even when existing techniques are applicable the following difficulties must be overcome to recover f_{ILP}. First the adversary does not know the complexity of hidden code and hence he must try all of the above techniques. Second depending upon the number of inputs involved and the degree of the polynomials, a large number of input output pairs for the f_{ILP} may be needed to recover the code. Third, even though f_{ILP} may actually be dependent upon a small number of variables, the adversary must assume that it is dependent upon all the variables whose values are sent to the hidden component from the open component.

If control flow is present, the application of above techniques becomes much more complex. While a large number of input output pairs may be available, these pairs must be divided into subgroups corresponding to different paths as different paths may correspond to different computations [16]. Since the adversary does not know how many paths must be considered, as some of the predicates and the control flow they form may be hidden, it is not known how many categories are there. Thus, it is unclear how this path based categorization can be achieved.

4. Experimental Results

We have implemented our technique in context of *Java* programs. To implement our technique we have made use of the `bloat` [19] facility from Purdue which provides the basic infrastructure for analyzing and transforming *Java* programs. Our experiments are based upon the following *Java* programs:

`jess` - Rule engine and scripting environment [18].
`bloat` - Java optimizer written entirely in Java [19].
`javac` - Java compiler J2SDK 1.4.0_01 [20].
`jasmin` - Java assembler interface [21].
`jfig` - 2D graphics editor [22].

For the purpose of these experiments, we initiate the proposed splitting algorithm with respect to a single local variable belonging to each method selected for splitting. This variable is selected to be the one which creates an ILP with the highest maximum arithmetic complexity across all ILPs created by different local variables. The characteristics of the splitting performed in terms of the number of methods chosen for splitting, total number of statements in the constructed slices, and the number of ILPs present after splitting are given in Table. 2. Next we characterize the complexities of ILPs and measure the runtime overhead introduced by our approach.

4.1. Complexity of ILPs

Tables 3 and 4 respectively summarize the arithmetic and control flow complexities of the ILPs present. As we can see, most of the ILPs are classified as *linear* or *arbitrary*. Due to the nature of the first four programs, most of the hidden computations were *linear* in nature. Since `jfig` contains many more arithmetic computations, it does contain many *polynomial* and *rational* hidden computations. Moreover, since many predicates were hidden in all programs, a significant number of ILPs were classified as *arbitrary*. The number of inputs on which ILPs depend was found to be small and so were the degree of the polynomials. In case of the `javac` program, entire loops were hidden. Thus, the number of inputs is listed as *varying* as it was dependent upon number of loop iterations – in each iteration a different array element was being sent to the hidden side. For the same reason the number of paths in the hidden code corresponding to some ILPs was *variable*. In general from Table 4 we can see that the control flow complexity is quite high as numerous ILPs depend upon hidden predicates and hidden control flow.

4.2. Runtime Overhead

We conducted an experiment to measure the runtime overhead of using splitting. The splitting was performed

Table 2. Split characteristics.

Benchmark	Number of		
	Methods Sliced	Statements in Slice	ILPs
`javac`	7	168	67
`jess`	11	192	57
`jasmin`	6	47	31
`bloat`	16	161	99
`jfig`	17	583	160

Table 3. Arithmetic complexity of ILPs.

Benchmark	Number of ILPs with *Type*					*Inputs*	*Degree*
	Constant	Linear	Polynomial	Rational	Arbitrary	(maximum)	(maximum)
`javac`	5	38	1	0	23	varying	2
`jess`	8	13	2	0	34	4	2
`jasmin`	3	15	1	0	12	4	2
`bloat`	25	22	12	0	40	5	2
`jfig`	8	62	23	31	36	7	6

Table 4. Control flow complexity of ILPs.

Benchmark	Number of ILPs with		
	Paths = *variable*	Predicates = *hidden*	Flow = *hidden*
`javac`	3	42	35
`jess`	0	28	16
`jasmin`	0	16	12
`bloat`	0	63	49
`jfig`	15	105	63

Table 5. Runtime overhead caused by software splitting.

Benchmark	Input Size	Component Interactions	Runtime Before → After	% Increase
`javac`	33K	875	2.13 → 3.37 sec	58%
	355K	4642	7.91 → 11.27 sec	43%
`jess`	dilemma (5K)	51	0.82 → 1.07 sec	31%
	fullmab (12K)	813	5.39 → 6.11 sec	13%
	hard (.5K)	11	5.53 → 5.67 sec	3%
	stack (2K)	63	0.78 → 1.05 sec	35%
	wordgame (5K)	48	8.55 → 8.83 sec	3%
	zebra (7K)	143	2.67 → 3.16 sec	18%
`jasmin`	small (124K)	117	1.14 → 1.27 sec	14%
`bloat` (1)	161smin.jar (149k)	73	22.93 → 23.87 sec	4%
	jess.jar(290k)	41	79.29 → 82.53 sec	4%

(1) Since `bloat` is a library, an optimizer based upon `bloat` was used to carry out the experiments.

under the guidelines and restrictions described earlier in the paper. We generated the open and hidden components and ran them on two separate linux based machines that communicated over the local area network. The results of executing the programs after splitting on various inputs are summarized in Table 5. No data was collected for `jfig` as it is an interactive application and thus small changes in execution time have little consequence. While there were many interactions between the open and hidden components, the runtime overhead is reasonable and comparable to overheads reported for other techniques by researchers. For example, guards introduced in [2] for making software tamper resistant increase execution times by 6% to 32.2%. The obfuscation transformations introduced in [8] for flattening existing control flow and introduce new control flow when applied to 50% of the program can increase execution time by a factor of 4 (of course, the overhead would be lower if the transformations are applied less frequently). Other works that we have studied do not report overhead costs.

5. Related Work

The techniques for software protection can be broadly classified as follows. First we have techniques that prevent or discourage *software piracy*, i.e. they are aimed at preventing the creation of illegal copies of the software. Second there are techniques for making software *tamper resistant*. Assuming that a user has access to the software, even legally, he or she may try to tamper with it to remove authentication code so that it can be freely distributed for illegal use. Third, assuming that the software has been tampered with, modified, and distributed illegally to users, *watermarking* is used by the producer of the software to identify illegal copies of the software. Code *obfuscation* transformations are employed to hide a watermark or tamper resistance code embedded in the software so that it cannot be easily detected and removed from the software.

This work is aimed at preventing software piracy, i.e. preventing creation of illegal copies of the software. While a legal user can make copies of the open components and tamper with them, security comes from not being able to supply hidden components, i.e. while tampering is possible it does not lead to obtaining a working copy of the software that can be distributed for illegal use. Below we describe some specific related works.

Software piracy. Our approach for prevention of software piracy through application splitting appears to be closest to work being pursued at Netquatrz [1]. However, their work is not publicly available and thus a detailed comparison is not possible. In [6] an approach is described that uses values generated by an *Electronic Security Device* (ESD) that is attached to the serial port of the machine to encrypt and/or decrypt user data. Pseudocode is used to implement protection functions including encryption and decryption of user data. Together with the pseudocode, a corresponding pseudocode interpreter is also embedded within the application. An obfuscation tool is used to hide the interpreter. It is claimed that it can take months of effort to break this method. In [11] software updates are exploited to carry out the function of protection against software piracy. User is forced to obtain upgrades which alter the format of the result data produced by the software. If the user does not obtain these upgrades, the data produced by the updated software cannot be read by the outdated copies of the software. Thus, sharing of data among the users is no longer possible. The weakness of this approach is that users that do not want to share data can continue to use the software.

Tamper resistant software. In [2] a network of security units, called guards, work together to detect changes to the binary. The guards essentially perform checksums on parts of the binary to detect if the software has been modified. By including multiple guards the task of detecting and removing the guards is made complex and hence protection against code modification is provided. The guards introduced are small and thus the runtime cost and code size increase due to them is very small. Another approach described in [7] provides a mechanism that redundantly tests for changes in the executable code, as it is running, and reports modifications. The above methods do not provide protection against software piracy to the extent that the method we have proposed does.

Software watermarking and code obfuscation. Software watermarks [5] can be introduced in the software and rendered highly undetectable through code obfuscation transformations [3, 4, 8]. Code obfuscation can also be used to prevent reverse engineering of the software. In [8] the transformations employed flatten the control flow that is present in the original program and introduce new control flow in code segments that were originally straightline code segments. Aliases are also introduced systematically. For example, control flow transfers are performed through indirect addressing carried out through aliased pointers. The overhead of obfuscation techniques can be significant both in terms of code size increase and execution time overhead. While our approach does more than obfuscation, as it prevents illegal copying of software, we argue that hiding part of the software is an effective way to obfuscate the software.

Encryption Function. Sander and Tschudin [14, 15, 13] proposed the concept of an encrypted function. They apply a homomorphic cryptosystem to protect the mobile code. This method protects computations of polynomials by encrypting constants and transforming code to produce output in encrypted form. In contrast our approach is applicable to more general computations such as non-polynomial computations involving complex control flow.

6. Conclusions

We presented a novel approach for protecting software from being stolen. By splitting software modules into open and hidden components, we ensure that an adversary cannot steal the entire software. Given that the adversary cannot steal the hidden components, he or she must construct these components by observing their runtime interaction with open components. Only by constructing the hidden components can a working pirated copy of the software be constructed. However, we construct the hidden components such that they are hard to identify because they are formed through program slices whose function is hard to determine. The hidden components contain both a part of original program's runtime state and part of its code. The number of variables that form the hidden state and the relationships among the values as defined by the hidden code are both unknown to an adversary. Thus, it is not possible to easily determine the functions performed by the hidden functions so constructed. We demonstrated that while a simple approach to splitting that hides entire modules is not practical, our approach based upon hiding slices is very effective. Moreover by carefully forming and selecting the hidden components we keep the runtime overhead interactions between open and hidden components reasonably low.

Acknowledgements. We are grateful to Michael Smith and anonymous reviewers for their feedback. Their comments were very helpful in revising the contents and presentation of the material.

References

[1] *Asymmetric Application Segmentation: A New Technology Paradigm for Software Licensing* - http://www.netquartz.com.

[2] H. Chang and M.J. Atallah, "Protecting Software Code By Guards," *ACM Workshop on Security and Privacy in Digital Rights Management*, Philadelphia, Pennsylvania, November 2001

[3] C. Collberg, C. Thomborson, and D. Low, "Breaking Abstractions and Unstructuring Data Structures," *IEEE International Conference on Computer Languages*, Chicago, IL, 1998.

[4] C. Collberg, C. Thomborson, and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, CA, 1998.

[5] C. Collberg and C. Thomborson, "Software Watermarking: Models and Dynamic Embeddings," *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, TX, 1999.

[6] M.J. Granger, C.E. Smith, and M.I. Hoffman, "Use of Pseudocode to Protect Software from Unauthorized Use," United States Patent 6,334,189 B1 12/25/2001.

[7] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan, "Dynamic Self-Checking Techniques for Improved Tamper Resistance," *ACM Workshop on Security and Privacy in Digital Rights Management*, Philadelphia, Pennsylvania, November 2001

[8] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of Software-based Survivability Mechanisms," *International Conference of Dependable Systems and Networks*, Goteborg, Sweden, July 2001.

[9] M.J. Granger, C.E. Smith, and M.I. Hoffman, "Use of Pseudocode to Protect Software from Unauthorized Use," United States Patent 6,334,189 B1 12/25/2001.

[10] D. Grigoriev, M. Karpinski, and M. Singer, "Computational Complexity of Sparse Rational Interpolation," *SIAM Journal on Computing*, Vol. 23, No. 1, pages 1-11, 1994.

[11] M. Jakobsson and M. Reiter, "Discouraging Software Piracy Using Software Aging," *ACM Workshop on Security and Privacy in Digital Rights Management*, Philadelphia, Pennsylvania, November 2001.

[12] D.C. Montgomery et al., *Introduction to Linear Regression Analysis*, Wiley, New York, 2001.

[13] T. Sander and C.F. Tschudin, "On Software Protection Via Function Hiding", *Proceedings of Information Hiding*, Springer-Verlag, LNCS 1525, pages 111-123. Berkeley, CA, 1998.

[14] T. Sander and C. Tschudin, "Protecting Mobile Agents Against Malicious Hosts", *In G. Vigna (ed.) Mobile Agents and Security, LNCS*, Feb. 1998.

[15] T. Sander and Christian F. Tschudin, "Towards Mobile Cryptography", *IEEE Symposium on Security and Privacy*, pages 215-224, May 1998.

[16] A. Sigal, R. Lipton, R. Rubinfield, and M. Sudan, "Reconstructing Algebraic Functions From Mixed Data," *SIAM Journal on Computing*, Vol. 28, No. 2, pages 488-511, 1999.

[17] R.E. Zippel, "Interpolating Polynomials From Their Values," *Journal of Symbolic Computation*, Vol. 9, pages 375-403, 1990.

[18] `jess` - http://herzberg.ca.sandia.gov/jess.

[19] `bloat` - http://www.cs.purdue.edu/s3/projects/bloat.

[20] `javac` - http://java.sun.com.

[21] `jasmin` - http://mrl.nyu.edu/~meyer/jasmin.

[22] `jfig`-http://tech-www.informatik.uni-hamburg.de/applets/javafig/index.html.