Remote software protection by orthogonal client replacement *

Mariano Ceccato, Paolo Tonella
Fondazione Bruno Kessler—IRST
Trento, Italy
{ceccato,tonella}@fbk.eu

Mila Dalla Preda
University of Verona
Verona, Italy
mila.dallapreda@univr.it

Anirban Majumdar
University of Trento
Trento, Italy
anirban.majumdar@unitn.it

ABSTRACT

In a typical client-server scenario, a trusted server provides valuable services to a client, which runs remotely on an untrusted platform. Of the many security vulnerabilities that may arise (such as authentication and authorization), guaranteeing the integrity of the client code is one of the most difficult to address. This security vulnerability is an instance of the *malicious host* problem, where an adversary in control of the client's host environment tries to tamper with the client code.

We propose a novel client replacement strategy to counter the malicious host problem. The client code is periodically replaced by new orthogonal clients, such that their combination with the server is functionally-equivalent to the original client-server application. The reverse engineering efforts of the adversary are deterred by the complexity of analysis of frequently changing, orthogonal program code. We use the underlying concepts of program obfuscation as a basis for formally defining and providing orthogonality. We also give preliminary empirical validation of the proposed approach.

Categories and Subject Descriptors

C.2.0 [**Computer-communication networks**]: General—*Security and Protection*; D.2.0 [**Software Engineering**]: General—*Protection mechanisms*

General Terms

Security

Keywords

Obfuscation, Clone Detection, Program Transformation, Software Security, Remote Trusting

*This work was supported by funds from the European Commission (contract N^o 021186-2 for the RE-TRUST project)

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

1. INTRODUCTION

A network application is an application that needs to exchange information over the network in order to work correctly. It involves a service provider, usually called *server*, and a service consumer, usually called *client* and a communication channel between them. We consider a scenario where the server is running on a trusted machine, while the client is running on an untrusted machine that might corrupt the client code for malicious purposes. This means that, before providing the requested service to the client, the server has to verify whether the client is executing according to its expectations, i.e., whether the client is in a valid state. This situation defines the *remote trusting problem*, where the trusted machine, i.e., the server, has to ensure that the application running on the untrusted machine, i.e., the client, has not been tampered with.

In this work, we face the problem of ensuring the integrity of part of the client code, later called the client's *critical part* and denoted with CP. In this setting, the attacker's goal is to tamper with the client application without being detected by the server. The attacker has full access to the client application and he/she can use any static and dynamic techniques to reverse engineer it. Given the practical limitations of available obfuscation technology and the theoretical ones investigated by Barak et al. [2], we make the assumption that an attacker with enough time and resources can perform a successful attack. However, it should be noticed that a successful attack involves substantial program comprehension effort, which is necessarily carried out by human beings. Based on this observation, we propose a protection scheme which periodically replaces the client code with a new version. In order to obstruct the comprehension of the new client code, we aim at generating code that is as orthogonal as possible to the previous versions. In this way, the attacker cannot take advantage of previous attempts. Given sufficient time and resources, an attacker can crack any obfuscated client, i.e., an attacker can mount a successful attack on any client code. Orthogonal replacement tries to address this issue by limiting the time that an attacker has to tamper with a particular version of the client.

Orthogonality is achieved through the application of different semantic preserving transformations (i.e., obfuscations) to the critical part CP and through splitting of the transformed application between the server and the client. We denote with $CP_1, ...CP_i$ the code of the critical part obtained through semantic preserving transformations. It turns out that obfuscation alone might not be enough to gain orthogonality. In fact, it might be hard or even impossible to make CP_i orthogonal to $CP_1, ..., CP_{i-1}$, because CP_i must behave the same as $CP_1, ..., CP_{i-1}$ to preserve the overall application's semantics. Our proposal is to split the code of CP_i between the client and the server in order to leave on the client code that is substantially different from the previous versions. The

splitting of the critical code CP_i is denoted with (C_i, S_i), where C_i denotes the code left on the client and S_i the code moved to the server. Orthogonal client replacement requires the new version C_i of the client code to be orthogonal to the previous ones $C_1, ..., C_{i-1}$. Since the semantics of C_i does not need to be the same as $C_1, ..., C_{i-1}$ (the overall semantics of CP is preserved in CP_i, not that of each client $C_1, ..., C_{i-1}$), the proposed protection scheme offers substantially more possibilities of code transformation than those achievable via pure client code obfuscation.

The notion of orthogonality used in this work is a cognitive notion. In fact, from an attacker's perspective a program P is orthogonal to another program Q if tampering with P does not reduce the effort involved in tampering with Q, in that no information acquired during the first attack can be reused to mount the second one. This notion of orthogonality refers to program comprehension activities carried out by humans beings. Hence, it is hard to define more precisely and to quantify. Consequently, in this work we resort to a practical and computable approximation, given by code (dis-)similarity. The idea is that two programs P and Q are orthogonal if they are dissimilar enough, so that analyzing P does not provide any clue for the analysis of (portions of) Q, since no portion of Q is similar to any portion of P. Code similarity has been deeply investigated in the area of clone detection [14, 13, 3]. Most available clone detection algorithms can be adapted to our purpose, i.e., to produce an approximate quantification of our notion of orthogonality.

The paper is organized as follows: first, we characterize our protection scheme in terms of the attacks it is intended to defeat (Section 2). Then, we describe the solutions available from the literature for the problem we are addressing and we explain how our proposal differs from the existing ones (Section 3). The core of the paper is Section 4, where we describe our proposal for a novel protection scheme based on orthogonal client replacement. In Section 5, we provide a preliminary validation of the approach based on two case studies. Conclusions and future work are presented at the end of the paper.

2. ATTACK MODEL

The *remote trusting scenario* consists of a trusted machine (called *server*) providing services to an untrusted machine (called *client*). The server has to ensure that the application running on the client has not been tampered with. This security problem is also called the remote attestation or software integrity problem. In this scenario, an attacker is a malicious user that aims at altering the client's behavior, either by means of static or dynamic analysis, in order to gain some personal advantage while going undetected from the server. In order to mount a successful attack, the attacker needs to understand the inner working of the application that he/she wants to modify.

During the execution of the application, the attacker has access to all the successive versions of client $C_1, ..., C_n$ delivered by the server. There are no limits on the static and dynamic code analysis techniques that he/she can use in order to reverse engineer them. Moreover, the attacker can access the information exchanged between the current client and the server (C_i, S_i) in the communication acts occurring during the execution of the application. Given some understanding of the client code and of its interaction with the server, the attacker can either modify the code running on the client or the content of the client memory (client state) in order to gain some personal advantage. However, the attacker can analyze and deduce information only from those portions of the critical code that reside on the client during the execution of the application. In fact, the attacker has not access to any of the successive versions of the server code $S_1...S_i$. An attacker succeeds when he/she is able to gain enough information form the analysis of $C_1, ..., C_i$ in order to maliciously modify the client code C_i without being caught by the server S_i.

DEFINITION 1. *An attacker A is said to have mounted a successful attack on (C_i, S_i) if A can alter the execution state of C_i, either by static or dynamic analysis and manipulation, so as to obtain an unpermitted behavior from C_i which goes undetected by S_i (i.e., if S_i is providing any service to C_i, it continues to do so, since S_i considers C_i trusted).*

3. RELATED WORK

Software based schemes for remote attestation have been proposed as a possible alternative to purely hardware based solutions. They usually make the assumption that precise information about the hardware hosting the client's execution is available or can be obtained. These protection schemes include Swatt [16] and Pioneer [15], applicable to embedded devices and desktop computers. These solutions verify that no malicious modification has occurred in the software by computing a checksum of the in-memory program image. These protection schemes can accurately estimate the time needed to compute the checksum since they have a precise knowledge of the client hardware and memory layout. This information can be used to detect attacks, since, in general, attacks introduce indirections that increase the execution time (e.g., redirecting memory checksum to a correct copy of the application while a tampered one is running). The main drawback of this solution is the assumption to have a collaborative user that provides precise information about the client hardware and its memory layout. Without this assumption, namely without an accurate prediction of the checksum computation time, the attacker could bypass the protection scheme through the so called memory copy attack [18]. When computing the checksum, the code is accessed in data mode, while when it is executed it is accessed in execution mode. The basic idea of the memory copy attack is to redirect every access in data mode to the original code in order to return the correct checksum even when executing a tampered application. Genuinity [12] is a protection scheme that, in order to deal with the redirection problem, incorporates the side-effects of the instructions executed during the checksum procedure itself into the computed checksum. The authors suggest that the attackers only remaining option, i.e., simulation, cannot be carried out sufficiently quickly to remain undetected. The possibility of adding code with no side-effects to unused portions of a code page is a possible way to bypass the Genuinity protection scheme [17].

When users are non-collaborative we cannot rely on the prediction of the computational time. In this case, a possible software only solution consists of exploiting the information exchanged during the communication between the client and the server in order to verify the integrity of the client application through assertions. If on the one hand the client could send false information to the server in order to satisfy the assertion, on the other hand it has to be honest as regarding those information that the server needs to provide the desired service (otherwise the network application does not execute). This means that only certain portions of the client code can be verified through assertion. Thus, a possible solution consists of using assertion to verify the integrity of part of the client application and to move to the server those portions of application that cannot be verified by the server through assertions [4]. The fragments of code that need to be moved to the server can be computed through barrier slicing. This solution introduces both a communication overhead and a computation overhead on the server. The

trade-off between the security and cost of this protection scheme has been studied in [5]. It is worth mentioning that the idea of splitting an application between client and server has been already used by Zhang and Gupta in order to prevent software piracy [19].

The solution provided in this work is based on orthogonal replacement and uses the combination of different obfuscations, whose utility has been studied by Heffner and Collberg in [9]. It can be used whenever barrier slicing [4] is not a viable approach for performance reasons. When the barrier slice containing the security-sensitive portion of the client requires unacceptable computational or network resources to be run on the server, we can leave the most performance-intensive portions of code on the client and use orthogonal client replacement to achieve software protection.

4. ORTHOGONAL CLIENT REPLACEMENT

In the following, the fragment of application code that is responsible for maintaining the portion of state that we want to protect is called the *critical part* CP. We assume this portion of the application code to be given as an input to the protection scheme.

The basic idea of orthogonal client replacement is to keep on substituting the critical part of the client with new versions that are orthogonal to the previous ones. Ideally, orthogonality ensures that an attacker cannot use the knowledge gained from the (static and dynamic) analysis of any previous client version to tamper with the current code of the client's critical part. Orthogonality is achieved through the application of semantic preserving transformations (code obfuscations) and code splitting.

- **Obfuscation**: By applying semantic preserving transformations that aim at obstructing code comprehension to CP, we obtain $CP_1, ..., CP_i$. However, obfuscation alone might not be enough to create different versions of the critical part that are orthogonal to each other. In fact, CP_i has the same semantics of $CP_1, ..., CP_{i-1}$, so it might be hard or impossible to make it orthogonal to the previous versions, i.e., to $CP_1, ..., CP_{i-1}$. However, it is possible to select a portion of the critical part CP_i (by splitting CP_i between client and server) that is orthogonal to the previously selected portions of $CP_1, ..., CP_{i-1}$.

- **Code splitting**: We split the code of the current CP_i between the server and the client obtaining (C_i, S_i). The splitting process ensures that the portion of code C_i that resides on the client is orthogonal with respect to the previous code residing on the client $C_1, ..., C_{i-1}$ (observe that for some C_j with $1 \leq j \leq i-1$ we might have $S_j = \emptyset$, in this case $C_j = CP_j$).

Thus, the i-th iteration of the orthogonal replacement process aims at generating a pair (C_i, S_i) such that: (1) C_i is orthogonal to the previous versions of the client code $C_1, .., C_{i-1}$; and, (2) (C_i, S_i) is functionally equivalent, denoted $\equiv$, to CP: $(C_i, S_i) \equiv (CP, \emptyset)$

4.1 Orthogonality

Intuitively, a statement s of client code C_j is orthogonal to a statement p of client code C_i, denoted $s \perp p$, when the analysis of statement s in C_j does not reveal any information about statement p in C_i. Orthogonal client replacement aims at generating new versions of the client code where all the statements of the new client are orthogonal to all the statements of the previous clients. However, there are portions of the critical code, such as system calls, library calls, and I/O operations, that cannot be modified when applying the semantic preserving transformations to CP and whose computation cannot be moved to the server. We call these fragments of the critical part *invariable*. This means that there is a limit on the degree of orthogonality that we can achieve. In other words, we can be orthogonal only with respect to those portions of the critical part that are not invariable. According to Figure 1, let

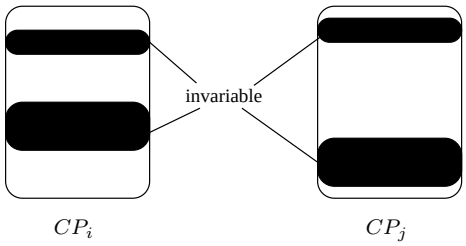

Figure 1: CP_i and CP_j share the invariable part

$Black$ denote the instructions of the critical part that are invariable, namely that are necessarily common to all the transformed versions of the critical part and must be necessarily left on the client. Let $White$ be the instructions of the critical part that can be modified or moved to the server. In particular, every possible variant CP_i of the critical part will share the invariable fragments, i.e., for all CP_i we have that $CP_i = Black \cup White(CP_i)$. Moreover, for every CP_i we have that $Black$ and $White(CP_i)$ form a partition, i.e., $Black \cup White(CP_i) = CP_i$ and $Black \cap White(CP_i) = \emptyset$. The splitting (C_i, S_i) is such that $C_i = Black \cup White(C_i)$, where $White(C_i)$ denotes the variable statements of C_i. Thus, we define the orthogonality between C_i and C_j only with respect to their variable part, namely $White(C_i)$ and $White(C_j)$. In particular, we say that C_i is orthogonal with respect to C_j if all the statements in $White(C_i)$ are orthogonal to all the statements in $White(C_j)$.

DEFINITION 2. *C_i is orthogonal to C_j, denoted $C_i \perp C_j$, if $\forall p \in White(C_j), \forall q \in White(C_i)$: $p \perp q$.*

This means that two versions of the client code are orthogonal when they differ in everything but the invariable part $Black$, that cannot be changed or moved by definition. It is clear that the invariable part $Black$ is application dependent and defines a limit on the degree of protection that can be achieved by our technique. For instance, if the code of the critical part presents many invariable statements, namely if $Black$ is almost equal to CP, very little can be either modified or moved to the server. In this case, orthogonal replacement offers limited support. On the other hand, when $Black$ contains few statements of the critical part, many orthogonal client copies can be generated by our technique.

The notion of orthogonality is a cognitive notion, based on the amount of knowledge about $s \in C_j$ a programmer can reuse when trying to understand $p \in C_i$. As such, it is hard to define precisely and operationally. However, the proposed approach requires a way to estimate it. While in this section we keep the notion of orthogonality quite abstract, in Section 5 we provide an approximation of this notion, based on clone detection, that can be used in practice. It should be however noticed that our approach is more general than its instantiation based on clone detection. In the future, better approximations of our cognitive notion of orthogonality may lead to implementations of our technique that better match our original idea.

Of course, there could be other forms of orthogonality (e.g., message orthogonality) which may increase the level of protection achieved. In this work, we address explicitly only statement orthogonality, which might also indirectly result in other forms of

orthogonality. Further investigation of such forms is part of our ongoing research.

4.2 Orthogonal client generation

Orthogonal client generation
INPUT
CP: Client critical part
$C_1, ..., C_{i-1}$: previous client code
OUTPUT
C_i: Next client, S_i: next server
BEGIN
1 `Repeat`
2 $CP_i =$ **RandomTransform**(CP)
3 $CP := CP_i$
4 $(C_i, S_i) :=$ **MoveCompToServer**$(CP_i, C_1, ..., C_{i-1})$
5 `Until` $(C_i \perp C_1) \wedge ... \wedge (C_i \perp C_{i-1})$
6 Output (C_i, S_i)
END

Figure 2: Orthogonal client generation algorithm

The algorithm in Figure 2 describes in more details the process that we use to generate orthogonal client copies. The condition of the `Until` at line 5 ensures that the new client code C_i is orthogonal to the previous ones.

By choosing $C_i = Black$ and $S_i = White(C_i)$ we can trivially satisfy the condition at line 5. This solution coincides with the barrier slicing solution [4] and actually requires no further orthogonal replacement of the client, since the attacker is left with no possibility of tampering with the client code (no sensitive client code is left on the client). In this work, we assume that the barrier slicing solution is not applicable for performance reasons. This means that there are some client's computations belonging to $White(CP_i)$ that cannot be moved to the server because of the major performance penalty associated with their server-side execution. They must be left on the client even though they are security sensitive. These performance-intensive statements of CP can be properly annotated in order to ensure that the computation they implement remains on the client in all successive client versions. This does not mean that they remain unchanged on the client (that would prevent orthogonal client generation): they can be transformed during the execution of Step 2 of the algorithm in Figure 2, but the transformer has to keep track of the annotations, so that the newly generated code has still information about what client portions cannot be moved to the server for performance reasons. The split step (Step 4 in the algorithm shown in Figure 2) will not be allowed to move them to the server.

The procedure **RandomTransform** picks up a set of semantic preserving transforms from a catalog and applies them to CP. Such transformations perform proper propagation of the annotations that mark the performance-intensive statements, so that they are available also in the new code. Function **MoveCompToServer** decides which portions of the transformed critical part to move to the server so as to guarantee orthogonality of the new client with respect to the previous clients $C_1...C_{i-1}$. However, statements marked as performance-intensive must be left on the client by **MoveCompToServer** (hence excluding the barrier slicing solution).

One way to gain orthogonality is to keep on the client the portion of code of the transformed critical part CP_i obtained during the i-th iteration that already differs from the previous versions of the client code $C_1, ..., C_{i-1}$. Let $OrthSt(White(CP_i), White(C_j)) = \{p \in White(CP_i) \mid \forall q \in White(C_j) : p \perp q\}$ be the set of statements of $White(CP_i)$ that are orthogonal to all the statements in $White(C_j)$. In other words, $OrthSt(White(CP_i), White(C_j))$ denotes the portion of the critical code $White(CP_i)$ that is orthogonal to the client code $White(C_j)$. The portion of CP_i that should be left on the client, i.e., C_i, can be computed as:

$$C_i = Black \cup \bigcap_{1 \le j \le i-1} OrthSt(White(CP_i), White(C_j)) \cup PerfIntens(White(CP_i))$$

The idea is to leave on the client the statements of CP_i that are orthogonal to all the statements of all the previous client versions, plus the invariable part $Black$ and the performance-intensive statements, that cannot be moved or modified. The `Until` condition of the algorithm in Figure 2 (line 5) evaluates to false whenever $PerfIntens(White(CP_i))$ is not orthogonal to the previous clients. In such a case, alternative transformations must be tried in order to achieve orthogonal client generation (i.e., the algorithm iterates until the orthogonality condition is met).

4.3 Transformation catalog

Orthogonality of the clients is achieved through the use of a transformation catalog of obfuscations. An obfuscating transformation modifies a program in order to make it more difficult to understand and to reverse engineer, while preserving its functionality. Here we briefly elucidate the salient aspects of obfuscating transformations from Collberg *et al.* [6].

The quality of an obfuscating transformation is measured in terms of its *potency*, *resilience* and *cost*. The potency of an obfuscating transformation measures the obscurity that has been added to a program, namely how much more complex is the obfuscated program to analyze with respect to the original one. The resilience of an obfuscation measures how difficult it is to break for an automatic deobfuscator. The cost of an obfuscating transformation measures the computational overhead added to the obfuscated program with respect to the original one.

Obfuscating transformations are usually classified according to the information they target. In the taxonomy by Collberg *et al.* [6], three types of obfuscations are discussed.

- **Layout obfuscation:** This category of transforms changes or removes useful information from the intermediate language code or the source code without affecting the instructions that contribute to the actual computation. Usually removing debugging information, comments, and scrambling/renaming identifiers fall within the domain of layout obfuscation.

- **Data obfuscation:** This category of transforms targets data and data structures contained in the program. Using these transformations, data encoding can be changed, variables can be split or merged, and arrays can be split, folded, and merged.

- **Control-flow obfuscation:** The objective of this category of transforms is to alter the flow of control within the code. Reordering statements, methods, loops and hiding the actual control flow behind irrelevant conditional statements classify as control-flow obfuscation transforms.

5. EMPIRICAL VALIDATION

We conducted a preliminary evaluation of the proposed approach by instantiating its components and conducting some case studies. Specifically, we chose a particular definition of orthogonality and

a particular set of obfuscating transforms. While the empirical results that we have obtained may suffer from some of these specific choices, the proposed approach is quite general and can accommodate further improvements that overcome the limitations of the current implementation. So, this empirical validations should be regarded as a proof of concept, rather than an actual and thorough assessment of the method.

We first describe how we approximate in practice the notion of orthogonality, followed by a description of obfuscating transformation we have used. We give also some details about the tool implementing the transforms and the orthogonality check. Next comes the description of the two case studies, followed by results and discussion.

5.1 Clone based orthogonality

Orthogonality from the program comprehension point of view is hard to define and quantify, so we resort to a practical and computable approximation, given by code similarity. In particular, we rely on clone detection techniques to gain a list of potential clones. Let $Clone(\,White(C_i), White(C_j))$ be the portions of code that are recognized as clones between $White(C_i)$ and $White(C_j)$ according to some given clone detection algorithm (intuitively the fragment of code that $White(C_i)$ and $White(C_j)$ have in common). In this setting the term *cloned statements* refers to the pairs of matching statements in $Clone(\,White(C_i), White(C_j))$. Cloned statements $(p, q) \in White(C_i) \times White(C_j)$ can be easily obtained once clones for $White(C_i)$ and $White(C_j)$ are known: they are the corresponding statements in the cloned code portions. Let $ClonedSt(\,White(C_i), White(C_j))$ be the set of cloned statements in $Clone(\,White(C_i), White(C_j))$. Orthogonality between statements of different clients can then be defined in terms of clones.

DEFINITION 3. *A statement s of $White(C_i)$ is c-orthogonal with respect to a statement p of $White(C_j)$, i.e., $s \perp_c q$, if the following holds:*

- $(p, q) \notin ClonedSt(\,White(C_i), White(C_j))$

This definition of c-orthogonality between statements leads to the following notion of c-orthogonality between clients.

DEFINITION 4. *C_i is c-orthogonal with respect to C_j, denoted $C_i \perp_c C_j$, if $\forall p \in White(C_j), \forall q \in White(C_i)$: $p \perp_c q$.*

Of course, the criteria of the splitting function can also be restated in terms of c-orthogonality.

5.2 Alias-based opaque predicates

For this work, we have implemented one particular type of potent obfuscation based on the use of *opaque predicates*. An opaque predicate is a conditional expression whose value is known to the obfuscator, but is difficult for an adversary to deduce statically. A predicate Φ is defined to be *opaque* at a certain program point p if its outcome is only known at obfuscation time. Following Collberg *et al.* [7], we write Φ_p^F (Φ_p^T) if predicate Φ always evaluates to False (True) at program point p for all runs of the same program. We call such predicates *Opaquely True (False)* at program point p. The notation $\Phi_p^?$ is used to denote an *Opaquely Unknown* predicate, i.e., one whose value depends on a program input supplied externally (by the user, by the operating system, etc.), such that it sometimes evaluates to True and sometimes to False during different program executions.

The opaque predicates used in our transformation tool use the concept of pointer aliasing. The rationale behind using such predicates is that precise inter-procedural static alias analysis is intractable.

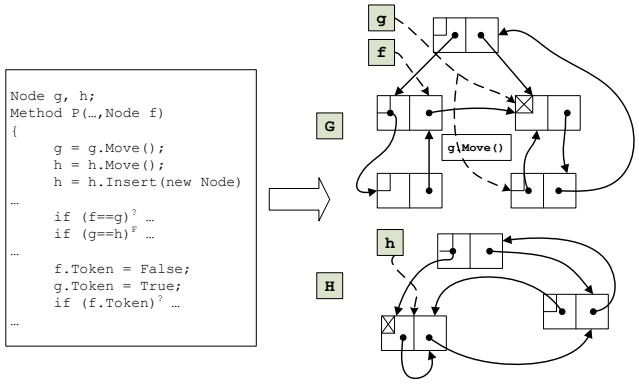

Figure 3: Opaque predicates constructed from objects and aliases (after [6])

Collberg *et al.* [7] proposed a technique which takes advantage of such intractability to construct resilient opaque constructs.

The basic idea is to construct a dynamic data structure and maintain a set of pointers on this structure. Opaque predicates can then be designed using these pointers and their outcome can be statically determined only if precise inter-procedural alias analysis can be performed on this complicated data structure. Figure 3 is an adaptation of Collberg *et al.* 's technique. Method `P`'s control flow is obfuscated using alias-based opaque predicates. Some method calls (e.g., `Move`) are used to manipulate two global pointers `g` and `h` which point to different connected components (`G` and `H`) of a dynamic data structure such as a linked list. The statement `g=g.Move()` will update `g` to move to a different location within `G`. The statement `h=h.Insert(new Node)` inserts a new node into `H` and updates `h` to point to some node within `H`. Method `P` and other methods that call it are also given an extra pointer argument `f` which refers to objects within `G`. Opaque predicates like $\Phi:$ if $(f == g)^?$ may either be True or False since `f` and `g` move around within the same component. `g==h` must be False since `g` and `h` alias to nodes within different components.

5.3 Program transformation tool

We used Txl [8] to realize the alias-based opaque predicates obfuscation described above. Our tool relies on an external secure random number generator (implemented in Java) and requires the alias specification file as input. This file contains pairs of pointers that are always (or never) aliases of each other and instructions to be called to change the pointer-based data structure, while keeping the invariant alias conditions known to the obfuscator.

In Figure 4, we can see the effect of the transformation. A basic block is split into a random number (2) of pieces (of random length). Each piece is inserted into the True (or False) branch of an `if` statement that uses an Opaquely True (or Opaquely False) predicate as condition. The other branch of the `if` statement is filled with randomly generated code which will be never executed.

The code we generate randomly consists of a sequence (of random length) of assignments to local variables (`tmp`) and to class fields (`f1` and `f2`). The expressions used on the right hand side of the assignments are formed by randomly selecting arithmetic operators and identifiers (of proper types) from a pool containing local variables, visible class fields and constant values.

The alias-based data structure is frequently changed by proper update instructions (invocations to method *updateAlias()*). When to update it is also decided on a random basis.

When applying opaque predicate based obfuscation, the code

size increases (as apparent from the example in Figure 4). However, most of the newly inserted code is never executed, because it is guarded by opaque predicates, so it is not expected to cause major performance overhead (low *cost*). On the other hand, the obfuscation is expected to be quite *resilient*, because of the additional control and data dependencies between original and injected code. *Potency* descends from the difficulty of precise static alias analysis.

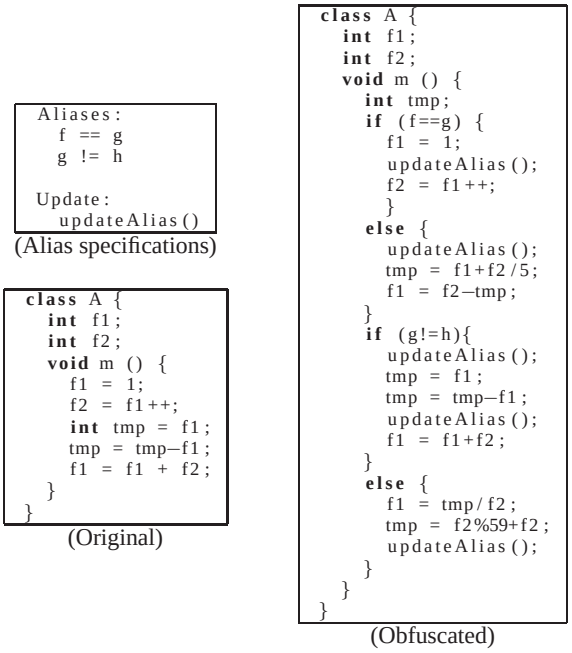

Figure 4: Effect of the obfuscation

5.4 Clone detection tool

For testing orthogonality, we rely on a source code clone detection tool called CCFinder [11]. This tool has been extensively evaluated in large scale empirical surveys and has been found to be effective in detecting clones at the source code level [1, 10].

A clone relation in CCFinder is defined as an equivalence relation (reflexive, transitive, and symmetric) on *fragments*, where a fragment is defined to be part of the source file and represented by an ID, and the coordinates from where it starts and ends. A clone relation exists between two fragments if and only if the token sequence included in them is identical. The first step of clone detection in CCFinder is *Lexical Analysis*, where the lines of the source files are transformed into a series of tokens based on the lexical rules of that language. The token sequence is then *transformed* with the aim of regularizing the identifiers based on certain transformation rules. A *pattern match* is then performed on all the substrings of the transformed token sequence. Here, equivalent pairs are detected as clones. In the last step, *formatting* is performed to reflect the clone pairs in the corresponding source files. CCFinder produces also summary metrics about the discovered clones [10].

5.5 Case studies

We took two Java applications as our case studies. The first one is a car race game and the second one a chat client. Both are network based and have embedded message send/receive primitives used by the clients and the server to communicate. The rationale for choosing these two applications is that both reflect interesting real-life scenarios of client-server computing and have secrets which a software developer would be interested in protecting.

The car race game, for example, has methods that change the speed, direction, amount of fuel left, and distance covered. When an instantiated car object executes on a client platform under the control of an adversary, these parameters are left unprotected from the adversary, who might in turn tweak the client code to gain unfair advantage over other non-malicious competitors. Similarly, in case of the chat application, an adversary might be interested in violating the chatroom policies by illegally creating his/her own room, joining a forbidden room, or accessing administrative privileges.

The critical part of the car race application (referred to as CP_{race}) consists of about 220 LOC. For the chat client (referred to as CP_{chat}) it is 110 LOC. Since CCFinder strips out comments, whitespaces and system calls from the source code, the number of tokens reported by CCFinder for each of CP_{race} and CP_{chat} is less than the number we would have obtained by applying a standard tokenizer to them. On average, the number of tokens per statement (TPS) is 14 for CP_{race}, 12 for CP_{chat}.

5.6 Results

We ran our experiment on a Pentium Centrino clocked at 2.0GHz with 1GB of RAM. Both the Txl-based program transformation tool and CCFinder were run on Windows XP.

In the first part of our experiment, we tested the degree of orthogonality of the clients generated by our program transformation tool, by applying the obfuscation step of the algorithm alone (Step 2, *RandomTransform*). We measured the number of clones detected by varying the *minimum clone length* (expressed as number of tokens) parameter of CCFinder. For this parameter we considered five consecutive multiples of the average number of tokens per statement TPS for each of the two applications. For each of the corresponding observations, the pool of clients generated was kept constant to 10.

The minimum clone length of CCFinder is a critical parameter for the algorithm in Figure 2. In fact, a too small value of this parameter could make the algorithm iterate for a long time (possibly, infinitely) because the c-orthogonality condition is never met, due to the large number of reported clones and the impossibility of moving to the server the performance-intensive statements. The size of the code generated by the opaque predicate-based obfuscator grows exponentially with the number of iterations (every new opaque predicate doubles the size of the block to be nested in its opaquely-true branch), hence convergence after a high number of iterations means also generation of an exponentially big code size. At the same time, most of the reported clones might be false positives when the minimum clone length is too small. What could happen is that CCFinder reports them as clones because they involve the same (short) token sequence, but any programmer would gain no information about one from the other, since they do not represent any meaningfully related computation. So, a too small minimum clone length is detrimental to the algorithm performance (up to making it unusable), while delivering no additional protection to the user (protecting false positives of clones is useless).

On the other hand, choosing a too large value for the minimum clone length simplifies the job of the algorithm, but might result in unacceptable false negatives, i.e., clients that are considered c-orthogonal only because they contain small clones, but indeed contain serious leaks of information an attacker might take advantage of. Hence, we must choose a value for the minimum clone length parameter which is: (1) big enough to allow convergence of the algorithm in a reasonable number of iterations; (2) small enough to prevent leak of information from clones with a length below the

chosen threshold.

Critical Part	Min. clone length		Clone Count
	Statements	Tokens	
CP_{race}	1	14	123
CP_{race}	2	28	33
CP_{race}	3	42	6
CP_{race}	4	56	1
CP_{race}	5	70	0
CP_{chat}	1	12	69
CP_{chat}	2	24	27
CP_{chat}	3	36	5
CP_{chat}	4	48	1
CP_{chat}	5	60	0

Table 1: Number of clones detected by CCFinder at increasing minimum clone length

As shown in Table 1, CCFinder detects a large number of clones involving approximately one or two duplicated statements (min. clone length 14/12, 28/24 respectively). This number decreases drastically as clones involving more statements are looked for. Almost no clone and no clone at all are detected when the clone size reaches 4 and 5 statements respectively. This suggests a value for the minimum clone length parameter between 3 and 4 statements, i.e., where the number of residual clones, that will be handled by means of code splitting, becomes low. At the same time, missing clones of size 2-3 statements does not seem to hinder the level of protection offered to the user. In fact, we expect that tracing as few as 2-3 statements back to the code of previous clients does not represent a substantial help for the attacker. Based on such qualitative considerations about the data shown in Table 1, we fixed the minimum clone length parameter to 50. We are aware that more empirical work is required to actually show that this choice is meaningful from the point of view of program understanding and (malicious) reverse engineering.

Table 2 shows the performance of our tool. We generated up to 1000 clients for each of CP_{race} and CP_{chat} and detected clones using CCFinder using a constant minimum clone length, equal to 50. As expected, when the number of previously generated clients increases, so does the number of clones detected by CCFinder. Hence, more iterations of the algorithm in Figure 2 are necessary to converge to a new c-orthogonal client. For both applications the target of generating at least 1000 c-orthogonal clients was achieved in a total computation time (including generation and detection) which is less than one hour for the car race game and less than half an hour for the chat client. Of course, we set this target, but we do not know if in practice it is a reasonable target. In fact, the number of orthogonal clients that can be generated determines the time left to the attackers to mount a successful attack. Given the expected life time T of an application, the life time of a single orthogonal client is T divided by the number of orthogonal clients we expect to be able to generate within time T. Knowing whether the resulting orthogonal client's lifetime is short enough to deliver the desired level of protection is hard in general and not in the scope of the present work. In our two examples, assuming a total application's lifetime of 5 years, availability of 1000 orthogonal clients would allow client replacement approximately every 2 days. However, as apparent from Table 2, it is reasonable to assume that, if needed, in a 5 year time substantially more than 1000 clients could be generated for these two applications. Correspondingly, the frequency of client replacement could be made even higher.

5.7 Discussion

As shown in the case studies, the proposed protection technique is able to generate a large number of orthogonal clients within a reasonable computation time. In our instance of the orthogonal client replacement strategy there is a trade-off between minimum clone length and cost of client generation (hence, number of generated clients). Reducing the minimum clone length is expected to improve security, but below some level no additional protection is delivered, at the price of a blowup of the algorithm's execution cost, which may possibly fail to generate any more client. We think that this trade-off deserves further investigation. As a proof of concept, our case studies showed that it is feasible to generate a large number of clients having a minimum clone length of around 4 statements. This seems to correspond to a practically relevant configuration of the proposed approach.

The usage of other available obfuscations in combination with opaque predicates could make our approach much more resilient. In particular, variable splitting and encoding could make similarity detection (differential attack) much harder even when substantial portions of the same computation are left on the client during code splitting. Once more, we ran a proof of concept experiment, showing that opaque predicates alone are powerful enough to make the approach viable in a reasonable setting. The more sophisticated and diversified are the transformations in our catalog, the higher the level of protection we are expected to deliver. Hence, having reasonably good results with a transformation catalog consisting of a single transform is very promising.

The obfuscatory strength of alias based opaque predicates relies on the intractability of precise static analysis of aliasing. However, predicates can be evaluated to hold True or False through debugging when the application executes (dynamic analysis attack). So, an attacker could try to remove the code in a branch that was never executed in past runs. Of course, in doing this, relevant code could be removed accidentally, because there is no way to distinguish between branches that are required by the original application, but are executed infrequently, and branches introduced by obfuscation, and never executed. One might try to increase the resiliency of the opaque predicates used in our protection scheme by deliberately adding predicates that evaluate to True (False) infrequently, but may cause dramatic application's failures if removed. As with any transform taken from the catalog employed by the proposed algorithm, the intrinsic strength of the obfuscation determines the target replacement frequency, but does not hinder the applicability of the proposed method itself.

6. CONCLUSION

In this paper, we have addressed the issue of remote software protection by proposing a novel approach which replaces the copy of an application running on a remote untrusted host with a new orthogonal version of it, under the assumption that state-of-the art obfuscation can be defeated if the attacker has enough time and resources, but it can be used effectively if the attacker is time-constrained due to replacement. We came up with a formal treatment of orthogonality and carried out a proof of concept experiment which deterred the reverse engineering efforts of an adversary looking for code similarity in order to learn from past versions of the client. We were successful in generating up to 1000 copies of orthogonal clients. Assuming that an application has a lifetime of 5 years, this allows for a replacement of the client with a new orthogonal once every 2 days, thereby reducing the time left to the attackers considerably. Orthogonality ensures that old clients are useless when mounting an attack against a new one.

Critical Part	No. of clients	No. of clones	Generation time	Detection time
CP_{race}	10	1	9”	9”
CP_{race}	50	9	44”	40”
CP_{race}	100	21	1’ 28”	1’ 21”
CP_{race}	500	160	16’ 30”	6’ 38”
CP_{race}	1000	347	35’ 20”	13’ 26”
CP_{chat}	10	1	6”	2”
CP_{chat}	50	7	26”	8”
CP_{chat}	100	11	51”	15”
CP_{chat}	500	97	5’ 11”	3’ 2”
CP_{chat}	1000	218	17’ 51”	5’ 59”

Table 2: Average tool performance

In the future, we intend to incorporate a full-fledged obfuscation catalog in our source code program transformation tool, so that the obfuscation added by opaque predicates could be complemented by potent data obfuscations. We also plan to investigate the trade off issues between minimum clone length, level of protection achieved and computational cost of the algorithm, so as to achieve an optimal balance.

7. REFERENCES

[1] B. S. Baker. Finding clones with dup: Analysis of an experiment. *IEEE Transactions on Software Engineering*, 33(9):608–621, 2007.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im) possibility of obfuscating programs. *Lecture Notes in Computer Science*, 2139:19–23, 2001.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM ’98: Proceedings of the International Conference on Software Maintenance*, pages 368–377, Washington, DC, USA, 1998. IEEE Computer Society.

[4] M. Ceccato, M. Dalla Preda, J. Nagra, C. Collberg, and P. Tonella. Barrier slicing for remote software trusting. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 1–10, Paris, France, 2007. IEEE Computer Society.

[5] M. Ceccato, M. Dalla Preda, J. Nagra, C. Collberg, and P. Tonella. Trading-off security and performance in barrier slicing for remote software trusting. Technical report, Fondazione Bruno Kessler-IRST, http://se.fbk.eu, 2008.

[6] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obduscating transformations. Technical Report 148, Dept. of Computer Science, The Univ. of Auckland, 1997.

[7] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (*POPL ’98*)*, pages 184–196. ACM Press, 1998.

[8] J. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.

[9] K. Heffner and C. Collberg. The obfuscation executive. In *Proceedings of the 7th International Conference on Information Security, ISC’04*, volume 3255 of *LNCS*, pages 428–440, 2004.

[10] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and implementation for investigating code clones in a software system. *Inf. Softw. Technol.*, 49(9-10):985–998, 2007.

[11] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[12] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of 12th USENIX Security Symposium*, 2003.

[13] R. Komondoor and susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the Static Analysis Symposiu, SAS’01*, volume 2126 of *LNCS*, pages 40–56, 2001.

[14] G. Myles and C. Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing, SAC’05*, pages 314–318, New York, NY, USA, 2005. ACM.

[15] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. K. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP), Brighton, UK, October 23-2-6*, pages 1–16, 2005.

[16] A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, pages 272–283, 2004.

[17] M. C. Umesh Shankar and J. D. Tygar. Side effects are not sufficient to authenticate software. Technical Report UCB/CSD-04-1363, EECS Department, University of California, Berkeley, 2004.

[18] P. van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computing*, 2(2):82–92, April-June 2005.

[19] X. Zhang and R. Gupta. Hiding program slices for software security. In *CGO ’03: Proceedings of the international symposium on Code generation and optimization*, pages 325–336, Washington, DC, USA, 2003. IEEE Computer Society.