CodeBender:
a tool for remote software protection using orthogonal replacement

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

In a typical client-server scenario, a server provides valuable services to client applications, which run remotely on untrusted client computers. Typical examples are video on demand, online games, voice-over-ip communications and many others. However, the user on the client side often holds administrative privileges on his/her machine and could tamper with the client application to fulfill the service violating the service usage conditions or service agreements. For example, by tampering with the application, the user may gain personal benefit or unfair treatments when accessing the remote service.

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.

In this article we present CodeBender\(^1\), a tool that implements a novel client replacement strategy to counter the malicious host problem. The client code has limited validity and, when expired, the server provides a brand new client that replaces the former one. The reverse engineering efforts of the adversary are deterred by the complexity of analysis of frequently changing, always different (orthogonal) program code.

Most desktop applications already resort to software updates to distribute fixes or new functionalities. Software updates are generally accepted by users, especially when they are motivated by security reasons.

CodeBender is implemented as a plug-in for Eclipse, one of the most popular IDE (Integrated Development Environment) for Java. Once the application is ready for deployment, many orthogonal versions can be generated and scheduled for automatic update.

The basic idea of orthogonal client replacement is to keep on substituting 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 analysis of any previous client versions to tamper with the current client. Orthogonality is achieved through the application of semantic preserving transformations (code obfuscations) and code splitting.

The process for generating orthogonal clients is summarized in Figure 1. We use the underlying concepts of program obfuscation as a basic random transformation for providing new code for replacement. This code is then split to move part of the client into the server so as to make any client version strongly coupled with the corresponding server and turn any update into a mandatory change. Code is verified to assess that it is orthogonal with respect to all the other versions.

In general not the entire application has strong protection requirements, in fact most of it could be devoted to draw the graphical user interface or to handle the interaction with the user. To control costs, the protection should be limited to just a portion of the entire application, namely to the critical part. The constraints to be considered when identifying the critical part may change a lot from one application to another, because they depend on security

\(^1\) CodeBender is freely available for download at http://selab.fbk.eu/ceccato/codeBender
requirements that are specific to each program. For this reason, the identification of the critical part should be done manually by someone with a deep knowledge on the code, e.g. by the developers of the application.

Figure 1. Process implemented by CodeBender for generating an orthogonal client. After transforming and splitting the code, orthogonality is assessed.

1 Random transformation

New clients are automatically generated by applying code transformations that can be randomly selected from a catalog of obfuscations (many obfuscations are available in the literature, see sidebar on code obfuscation). Code obfuscation modifies a program in order to make it more difficult to understand and to reverse engineer, while preserving its functionality.

However, even if obfuscated programs are harder to understand and analyze, an attacker with enough resources could eventually understand them (obfuscation is broken) and consequently code tampering can happen (empirical evidence of this is given by controlled experiments [3]).

At the moment, CodeBender implements just a single kind of obfuscation based on the use of opaque predicates. An opaque predicate is a conditional expression whose value is constant (always True or always False) and known to the obfuscator. However, it is difficult for the attacker to statically deduce the predicate value.

Opaque predicates are evaluated in decision points, in order to hide the correct execution flow to the attacker. Moreover, random or buggy code is added into the basic blocks controlled by those branches that are never triggered by opaque predicates.

The basic idea for creating an opaque predicate is to construct a pointer-intensive dynamic data structure and to maintain a set of pointers on this structure. Opaque predicates can then be designed using these pointers. Their outcome could be statically determined only if precise inter-procedural alias analysis would be performed on such complicated data structure.

Our implementation relies on a data structure exemplified in Figure 2(a). This data structure is a composition of several nodes, where each node maintains two pointers to two other nodes (r and l). In the example, \( A \cdot r \cdot r \) is an alias to \( A \cdot l \) (arrows in red), because they both refer to \( B \). We can update the structure and keep this condition valid, for example, (1) by making \( A \cdot r \) refer to \( D \), or (2) by adding a brand new node, node \( Z \) and than assigning \( A \cdot r=Z \) and \( Z \cdot r=B \), this second option is shown in Figure 2(b). As part of the obfuscation, new statements are added to the code to continuously update the data structure. These statements are randomly chosen from a list of operations that, while mutating the data structure, guarantee a known subset of alias conditions to remain valid. Nodes are added, removed and updated, so that aliases among pointers are frequently changed, thus making it very hard to statically detect whether two pointers refer to the same entity, even with the support of automatic analysis tools.
Figure 2. Pointer intensive data structure used to obfuscate source code.

Code obfuscation

Many algorithms for code obfuscation have been proposed in the literature. In the taxonomy by Collberg et al. [6], they have been classified into layout, data and control-flow obfuscation. 

**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/re-naming identifiers fall within the domain of layout obfuscation.

*Identifier renaming* is an instance of layout obfuscation that removes relevant information from the code by changing the names of classes, fields and operations into meaningless identifiers, so as to make it harder for an attacker to guess the functionalities implemented by different parts of the application. Identifier renaming is a widely implemented obfuscation technique, implemented by many commercial and academic obfuscators. Identifier renaming has no performance overhead.

**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. Obfuscation based on *Opaque predicates* [7] is a control-flow obfuscation that tries to hide the original behavior of an application by complicating the control flow with artificial branches. An opaque predicate is a conditional expression whose value is known to the obfuscator, but is hard to deduce statically by an attacker. An opaque predicate can be used in the condition of a newly generated *if* statement. One branch of the *if* statement contains the original application code, while the other contains a bogus version of it. Only the former branch will be executed, causing the semantics of the application to remain the same. In order to generate resilient opaque predicates, pointer aliasing can be used, since inter-procedural static alias analysis is known to be intractable.
2 Splitting

Code transformation alone does not guarantee an adequate level of protection. In fact, even if obfuscated code is hard to understand, it is still semantic equivalent to the original code, i.e., it implements exactly the same functionality. CodeBender overcomes this threat by partitioning the client code, namely splitting the code, into two. Only one part is delivered and run on the client, the other part is moved and run on the server. Functional equivalence is preserved as long as the two parts are executed together.

Since there is a finite number of configurations that can be resorted to change the boundary of code splitting, it can guarantee only a limited number clients with code that is functionally different. However, in practice we observed that, when obfuscation and splitting are used together, they are able to generate a substantially higher number of orthogonal clients.

In an extreme case, all the sensitive code could be moved to the server, and only safe code (possibly just the user interface) would remain on the client. This solution degenerates into the barrier slicing solution [2] (see sidebar on related works) 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). However, we prefer to avoid the barrier slicing solution for performance reasons. In fact, in our experience, splitting shown smaller performance overhead than barrier slicing. This means that there are some client’s computations that should not be moved to the server because of the major performance penalty associated with their server-side execution.

Figure 3 shows how a code sample is changed by splitting. First of all a program variable (or a set of variables) is chosen to be moved to the server (in the example we choose $f_1$). In the first step, Figure 3(b), all the uses of $f_1$ are replaced by ask, and all the assignments to $f_1$ are replaced by an update. Since the variable is removed from the client, its value in not known locally and must be provided by the server as required. When an ask statement is executed (use), a message is sent to the server and the execution is suspended until the server responds with the required value. On the other hand, when the execution hits an update (assignment), a message is sent to the server to propagate the new value.

In the second step, Figure 3(c), updates and asks are replaced by the same synchronization call to sync, in order to make them harder to distinguish for an attacker. Calls to sync have two parameters. The first parameter, namely the synchronization index, is used by the server to discriminate between update and ask, and to distinguish among variables in case more than one have been split. The index is decided at split-time, in the example indexes 1 and 6 mean update, while 2, 3, 4, 5 stand for asks. The second parameter contains the new value to be sent on update, or a random value when nothing has to be sent (on ask), the example just reports 0 for sake of simplicity.

Upon receiving a synchronization message, the server examines the synchronization index (first parameter). If the index corresponds to a valid requests, the server returns the current value or, in case of valid update index, the moved variable is updated with the value coming from the client (stored in the second parameter). Otherwise, if the index is invalid or if no value has to be provided (update case), a random or wrong value is sent back.

The dependency of the client on the missing variables (that have been moved to the server) implies a very important property, i.e., mandatory update. When a client update is delivered, the client user can not refuse to accept it, because the server will also be updated, with different moved variables and different synchronization
Software updates can be distributed on a decided rate, adopting the update mechanisms commonly available in modern operating systems. The application should be stopped, updated and then restarted. More advanced mechanisms that do not require the application to be stopped could be developed on a case-by-case basis. Their automation is out of the scope of the present work.

3 Orthogonality check

The notion of orthogonality is a cognitive notion, based on the amount of knowledge about an expired client an attacker can reuse when trying to understand a new version. As such, it is hard to define it precisely and operationally. However, the proposed approach requires a way to estimate it. We resort to 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.

Two distinct portions of code are recognized as clones when they contain exactly the same code. A looser definition of clones allow them to contain some differences, depending on the clone detection algorithm. For instance, variables could have been renamed but, as long as the syntactic structure is the same, two pieces of code may be still considered clones.

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

A clone relation in CC-Finder is defined as an equivalence relation (reflexive, transitive, and symmetric) on fragments, where a fragment is defined to be a contiguous 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.

Of course very small clones will be always reported, but they do not represent a problem as long as they do not group into longer sections. In fact, small clones do not leak sensitive information that an attacker can use to correlate a client version with the next one. Moreover, similar statements in different client versions could come from the different portions of the original code (transformation is non-deterministic). In this case, they would confuse the adversary, rather than helping. Past experience [5] led us to consider as not significant those clones whose size is lower than five statements.

Not every statement can be obfuscated or removed from the client. For instance, interactive code (user input/output, graphical user interface), local resource access code (file access, system calls) must run on the client, so that the related library calls can not be modified when applying code transformation. However, such code is usually not critical with respect to code tampering. We call these fragments of code 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 (i.e., the part of the application potentially sensitive to the malicious user’s attacks) that are not invariable.

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

As shown in the control flow of Figure 1, after obfuscation and splitting a candidate update must pass the orthogonality check in order to be deployed. In case the check is not satisfied, code transformation (obfuscation and splitting) is iterated until orthogonality is achieved. This poses a theoretical limit to the approach, because
even though the complexity of code transformation is linear in the size of the code to transform, the number of iterations is potentially unbounded. However, as reported in the following (Section 4), the algorithm is able to terminate in a reasonable amount of time on real case studies.

4 Practical adoption

CodeBender extends Eclipse by providing two buttons that trigger the two corresponding functionalities, i.e., Random transformation and Split.

As a first case study, CodeBender was used to protect CarRace, a network application consisting in around 900 lines of code. CarRace implements an on-line game that allows two remote players to compete in the same race. A game participant could tamper with the client application to remove constraints (e.g., to run faster than allowed or to take shortcuts) and gain unfair advantages, thus the application needs to be protected somehow. However, it is not the entire application that has strict security requirements, but just a (critical) part of it, 200 lines of code. In the car race game, we are not interested in protecting choreographic details such as the layout or the displayed colors. The critical part of the system is represented by the physical model of the car, that enforces all the sensitive constraints.

The second case study is JHotDraw, a framework offering two-dimensional drawing facilities, it consists of 18,000 non-commented lines of code and 2,000 methods. We considered this application to study how CodeBender scales when the source code to protect is large. The critical part of JHotDraw consists of 1,600 lines of code.

When the “Random transformation” button is pressed, source files are transformed according to the procedure described above. Then code is split by pressing the “Split” button. A dialog pops up and asks the user which program variable to split. The portion of the critical part that participates in the computation of such variable will be moved to the server. The removed code is stored in a separate file and packed so as to be ready to be deployed to the server. Proper synchronization statements are added to make the two halves (server and client) run in parallel and exchange data as required. By selecting different variables, different portions of code will be removed from the client, thus assuring significantly different new client versions for replacement.

Since the variables to split are not infinite and code transformations are non-deterministic, it could happen that the just obtained candidate client is not orthogonal to previously installed clients. Thus, before deploying the candidate update, we asses orthogonality using CC-Finder. CC-Finder requires to specify the source code language (in our case Java), and an input directory, containing all the previously deployed clients, together with the current candidate. The output of CC-Finder tells us whether the newly generated client contains clones larger than the threshold (we used a 100 tokens threshold). When no such clone can be found the client generation process terminates and the new, orthogonal version of the application can be deployed.

Figure 4 shows the time required by the orthogonality check for an increasing number of candidate clients. To estimate orthogonality, we resorted to SRA (ratio of similarity between different files), a metric computed by CC-Finder that measures the ratio of inter-file clones. Both for CarRace and JHotDraw this value was quite low (respectively 1.6% and 4.9%).

While the generation of 1,000 candidate orthogonal clients took just 3 minutes for CarRace, on the larger case study (JHotDraw) it required 8 hour and 37 minutes. Our evaluation of CodeBender on the two case studies supports the feasibility of the approach, because no observable delay has been reported by the user of the protected applications. Both in small and large case study applications, we were able to generate up to 1,000 orthogonal versions in a working day. If we over-estimate that such application will live for 5 years before a new version with new features will be available, we can guarantee a replacement every 2 days, thus significantly reducing the time for an attacker to tamper with it.
5 Lessons learned

After testing CodeBender on actual case studies we learned some important lessons that are summarized here:

- Even though obfuscation and splitting are unable to produce unlimited orthogonal clients, in practice, the amount of orthogonal clients produced by their combination was estimated to be enough to cover the whole lifespan of the case study applications.

- To guide the decision on how many orthogonal versions should be available to protect an application, developers should estimate (1) the expected life of the application and (2) what replacement rate can be regarded as acceptable for the users.

- Protection techniques usually involve some performance penalty. To limit such a penalty, we apply protections only to the critical part, and leave the rest unprotected. This worked well on our case studies, but it poses a limit in the degree of automation of the approach. In fact we had to identify the critical part manually. However, this required a limited, acceptable effort.

- Protection mechanisms often come at some cost, for instance computation and communication overhead. Cost estimations and their practical evaluation are the cornerstone concepts that can drive the success or the failure of the adoption of any technique. Our approach showed a low overhead, that has been judged affordable, at least in our case studies.

In our future work, we intend to investigate more advanced metrics to estimate orthogonality and integrate new obfuscations in the generation of orthogonal clients. Moreover, we are evaluating the possibility of deploying CodeBender in industrial environments.
No definitive solution exists in the literature to solve the problem of protecting the integrity of code running on a remote, potentially malicious, client. Available techniques are based on different assumptions and offer different degrees of security (usually at different degrees of performance overhead). The research area of code protection is a very active one and alternative solutions are still under investigation.

**Tamper resistant software:** Tamper resistant software [9] typically calculates a checksum on its code and checks whether the checksum corresponds to the expected value. In offline applications this expected value has to be stored inside the software and the decision whether tampering has occurred has to be taken locally by the client software itself. Typically this approach is complemented with obfuscation techniques that harden understanding. White-box cryptography [12] aims to hide cryptographic keys into applications, either in large collection of lookup tables or in executable code. The latter case is a form of tamper resistant software, as code modifications will alter the key and consequently cripple the functionality of the application. When these software techniques are used to protect standalone, non-networked applications, their security is limited. Self-checking software can be attacked with hardware support [17] and the tamper response mechanism is often a weak point.

**Remote attestation:** Networked applications suffer less from these issues. The integrity checksums do not have to be present in the client software and the comparison between the run-time and the pre-computed checksum can be performed remotely by the service provider, which is not under control of an attacker. Some techniques generate proofs of authenticity based on the computations performed on the untrusted host, e.g., by embedding trace gathering code in the original program and locally cross checking the trace [1]. Swatt [15] and Pioneer [14] 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). genuinity [11] 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 [16].

**Code splitting:** Alternatively, one can limit the impact of tampering by moving critical code away from untrusted platforms or by computing on encrypted data and/or with encrypted functions. When only certain portions of the client code can be verified, 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 [2], this is a form of server-side execution. 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 security and cost of this protection scheme has been studied [4] to search for an optimal solution.
Dynamic replacement: Other techniques use dynamic replacement to periodically replace the copy of the program running on the untrusted machine with the goal of limiting the amount of time that the attacker has at hand to reverse engineer the application. Some research prototypes implemented dynamic replacement of protection code using code mobility features offered by dynamic aspect-oriented platforms [8] or by ad-hoc JVM extensions [13]. The open question is how to generate different functionally equivalent versions of the program so that the reverse engineering information obtained by the attacker from all previous versions cannot be used to comprehend how the new ones work.

References