bianlian: Remote Tamper-Resistance with Continuous Replacement*

Christian Collberg† Jasvir Nagra‡
Will Snavely§

University of Arizona Technical Report TR08-03

Abstract

In this paper we describe bianlian, a system for producing tamper-resistant client programs running over the Internet. The basic idea is to use *continuous replacement* of program blocks running on the client site to make it difficult for an adversary to analyze and modify the program. There are many potential applications, for example protecting client programs in networked computer games from being tampered with. We show the basic design of a continuous replacement system for Java, including a number of obfuscating and tamperproofing transformations.

In achieving tamper-resistance bianlian incorporates two novel ideas. First, it maintains an incorrect pool of code on the client such that no snapshot that an adversary takes contains the complete correct program. In addition, the type of incorrect code introduced ensures that on different input, executing the same trace of instructions will produce incorrect results. Secondly, bianlian achieves scalability through a distributed protection scheme which allows the server to offload the tamper-detection of one client to another client.

1 Introduction

Remote tamper-resistance is an important special case of tamperproofing. In this scenario, the program we want to protect (which we'll call C) runs remotely on an adversary's untrusted site (the *client* site), but is in constant communication with a trusted program S on our (*server*) site. In addition to providing services to the client, the server also wants to detect and respond to any tampering of C. Responding to tampering can take many forms, but typically communication is just terminated. Our assumption is that the server is providing a service without which the client can't make progress, and this means that refusing to provide that service is the ultimate punishment.

There are many applications that fit neatly into this model. In networked computer games, for example, the game server provides continuous information to the player clients about their surroundings. Players will often try to hack the clients to gain an unfair

*This work was supported in part by funds from the European Commission (contract N° 021186-2 for the RE-TRUST project).

†Department of Computer Science, University of Arizona.

‡Google Inc. This work was done in part while at the University of Trento, Italy.

§Department of Computer Science, University of Arizona.

advantage over other players, for example by being able to see parts of a map (which, for performance reasons, is being held locally) that they're not supposed to see [3,17].

Many solutions to the remote tamper-resistance problem are variants of various levels of "sharing" of the C code and data between the server and the client. At one extreme, all of the C code and data resides on and is executed by the server. This is sometimes known as *software as a service*. Whenever the client wants to make progress it has to contact the server, passing along any data it wants the server to process, and wait for the server to return computed results. This kind of server-side execution can lead to unacceptably high compute load for the server and unacceptably high latency for the client. On the other hand, since all the sensitive code resides server-side, there is no risk of the client tampering with it. At the other extreme, the client runs all its own code and does all the work. This requires the server to share all its data with the client, which can be bandwidth intensive. Since all computation is done client-side, it's more difficult for the server to guarantee that the client has not tampered with the code. Most systems will settle on an intermediate level solution: some computation is done server-side, some client-side, and this balances computation, network traffic, and tamper-detection between the two.

In this paper we introduce a system called bianlian[1] which uses continuous replacement of the client program to make it more difficult for an adversary to tamper with the code. The basic idea is to keep the client code in *constant flux*, to make it difficult to analyze and modify. To make the code difficult to analyze we use various obfuscating transformations [8]. However, since it is generally assumed that every obfuscating transformation can eventually be broken, and, since we cannot prevent the adversary from employing any computational resources available to it to do the analysis, we force the client to continuously update its code. Ideally, we are able to force updates at a rate that's high enough, and employ obfuscations that are potent enough, to together overwhelm the client's analytical resources.

The act of testing the response from a tamper-proofed client adds significant load on the server. The server must maintain a subset of state in order to check for a correct response from the client. Traditionally this keeps this method of tamper-resistance from scaling. By recognizing that checking state is simply another computational task which can be redistributed to clients, we can mitigate this cost to an extent. In many scenarios, the number of trusted clients vastly outnumber the number of untrusted clients (although the server may not be able to distinguish between them). By re-sending the task of checking the correctness of a computation to different randomly selected client, we limit the extent to which two malicious clients can collude.

The remainder of this paper is organized as follows. In Section 2 we present a system design that models *any* solution to the remote tamper-resistance problem. In Section 3 we show a prototype implementation for Java. In Section 4 we describe challenges to implementing continuous replacement in languages like Java that support concurrency and exception handling. In Section 5 we present related work and in Section 6 we conclude by describing some future directions to further improve the scalability of our design.

[1] bianlian is Mandarin for "changing face". Face-changing is a traditional Sichuan Opera art form that involves continuous replacement of face masks.

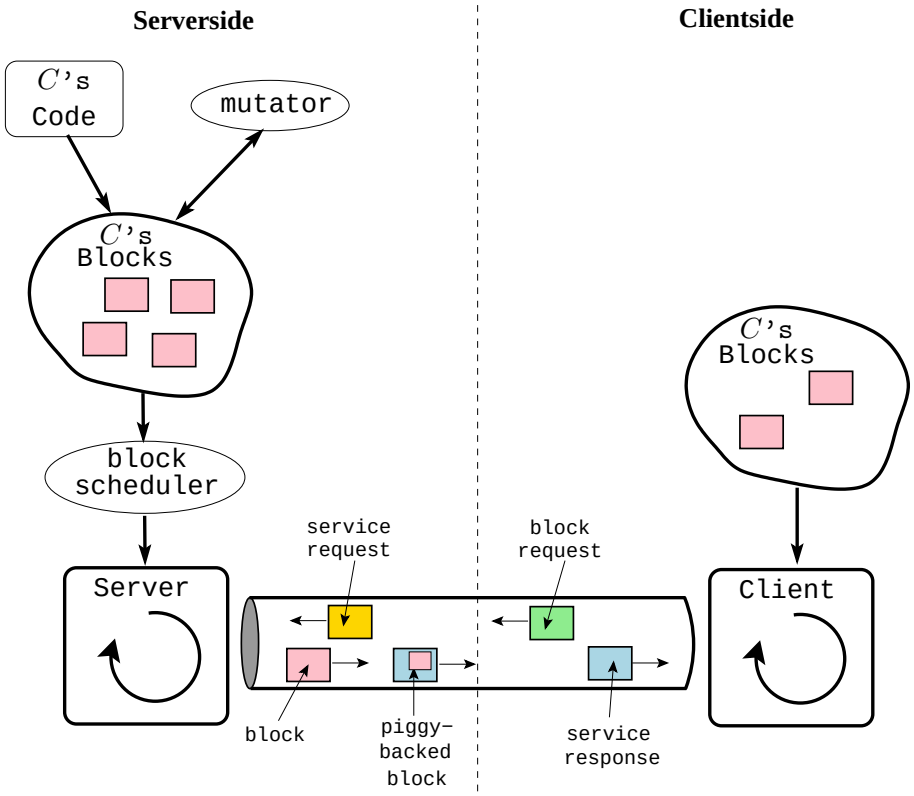


Figure 1: Overview of a remote tamper-resistance scenario.

2 Modeling remote tamper-resistance

Consider Figure 1 which models, at a very high level, remote tamper-resistance systems. As in typical client-server scenarios, the client sends *request-for-service* packets (yellow) over the network to the server which returns *service-response* packets (blue) in return. In a computer game, for example, the client may tell the server "I just entered dungeon 372!" to which the server responds with a list of the nearby monsters. What's different in a remote tamper-resistance system is that the server and the client both maintain a representation of the C code (the code run by the client), in what we'll call a *bag-of-blocks*. The client executes out of its bag by selecting a block to run, jumping to it, selecting the next block, etc. The server, on the other hand, has a *mutator* process which continuously modifies its bag of blocks, and shares any modified blocks with the client. Sharing can happen by the client asking the server for a block it doesn't have (at any one point in time the client might hold only a subset of all the code blocks), sending a *request-block* packet (in green) to the server and getting a new block (in pink) in return. The server may also *push* blocks onto the client. For performance reasons, the server may return multiple blocks at the same time, anticipating the blocks that the client will be needing in the near future, based on what it knows of the client's current state. A *block scheduler* process on the server determines which blocks to return to the client at what time.

The level of tamperproofing we achieve through this setup is determined by

1. the fraction of the total number of blocks that the server shares with the client,

2. the rate by which the server generates mutated blocks and pushes them onto the client, and

3. the rate by which the adversary can analyze the continuously changing program in the client's bag-of-blocks.

In a pure server-side execution scenario the server never shares blocks with the client (the client's bag of blocks is always empty), which means that the adversary has no program to analyze. In a scenario without tamperproofing the code blocks are transfered once and for all to the client, never change after that, and this gives the adversary ample time for analysis. In an intermediate scenario the server always keeps some blocks to itself, making it possible for the client to analyze some, but not all of, the program. When the server receives a request for one of its private blocks it will instead tell the client to supply the inputs to the block, and returns the result of executing it.

What's novel about the work in this paper is that the intermediate scenario is augmented by *continuous replacement* of blocks — not only does the client receive only some of the blocks, the blocks it's given are continuously replaced with new ones. These new blocks are often obfuscated versions of previously received blocks. For example, after having seen blocks A and B, the client may be given a block C which is a *merged* version of A and B. Or, after having seen A, the client gets B and C which are A *split* into two halves. Or, the client gets a version of A that is incorrect (but never executed) or has an altered API. The different versions can coexist or blocks may become obsolete over time. The server maintains the altered version of the program so it can continue to deliver blocks required to make the client execute correctly.

To reduce network traffic we want to keep the block replacement rate as low as possible. At the same time, we want to make sure the client doesn't have enough time to analyze the program between updates! To achieve this tradeoff, the mutator process (essentially an on-line obfuscator) generates blocks which are as difficult to analyze as possible. Also, at no point in time should the client's bag-of-blocks contain a complete and correct program. If it did, the adversary could simply take a snaps-shot of the bag and analyze it off-line. In fact, ideally, even if the client saves a copy of all the blocks it has ever seen, the adversary still shouldn't be able to fully analyze the code.

2.1 Block types

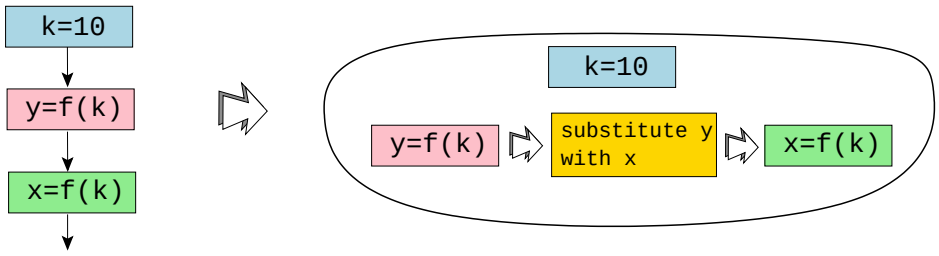
Let's look at some of the obfuscation strategies the mutator can use to generate a stream of blocks which is hard for the adversary to analyze. We have five different kinds of blocks:

measuring block: This type of block measures some aspect of the client and returns the result to the server for verification. At its simplest, we could compute a hash over another block or sets of blocks. This is similar to tamperproofing strategies used in single-process systems [4,9]. For higher levels of protection we could verify the integrity of the operating system or hardware on which the client is running, either by hashing or by timing some operation we ask the client to perform.

oblivious hashing block: An oblivious hashing [5] block weaves a hash computation into the control flow of a block and returns the hash value to the server for checking. The hash value can either be computed as a side-effect of the real control flow or as

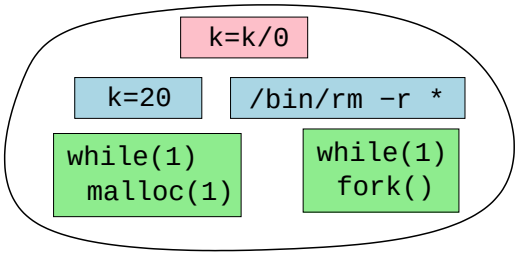
the result of challenge input values supplied by the server. The return value can be tested by the server or sent to a different client with sufficient data to allow them to perform this check on behalf of the server.

mutation block: The server can supply blocks which are themselves mini-obfuscators, modifying other blocks already in the bag. These mutators can introduce bugs into correct blocks after they've executed, correct bugs in incorrect blocks before they execute, transform a block that has no future use into a block which will be used soon, or perform semantics preserving transformations on blocks between executions. In this example, the green block is never explicitly transfered to the client but rather transformed by the yellow mutation block from the similar pink block:



For performance reasons these transformations will likely be relatively minor, but they can help keep mutation-rate high and communication-rate low.

unexecutable block: This type of block will, if executed, cause the program to crash or otherwise malfunction. The block will never actually be executed, but there will be bogus calls to it (perhaps protected by opaque predicates [7]) from other blocks. In this example, pink blocks cause the program to throw an exception, blue blocks destroy global data, and green blocks exhaust the client's computational resources:



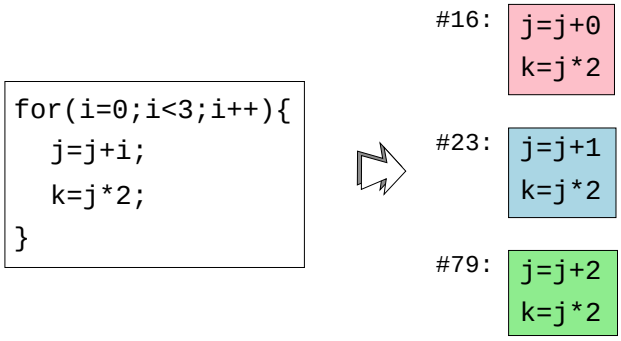
By making sure that the bag contains this type of block the server makes it harder for the adversary to execute blocks in isolation to experimentally figure out what each one does.

server-side block: Instead of performing a computation a server-side block passes its arguments to the server which computes the result and passes it back to the client. Making sure that the bag-of-blocks has a few server-side blocks ensures that it never contains a complete program and this makes complete analysis difficult.

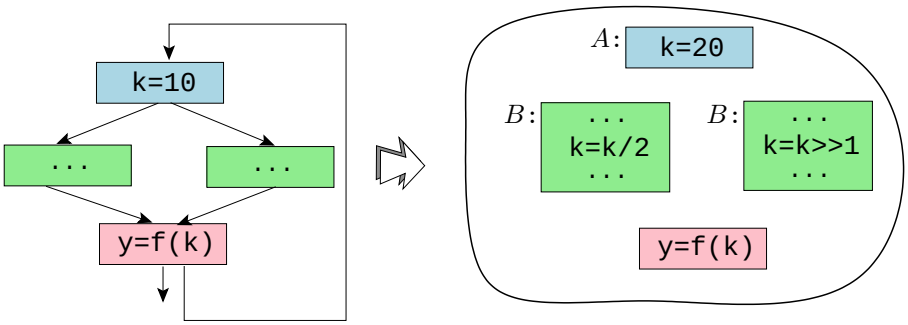
2.2 Block transformations

In addition to the five block types, we have three kinds of block transformations:

block unrolling transformation: Instead of supplying the client with a single block implementing a loop, the server can unroll the loop and send the client one iteration at a time. Each iteration can be obfuscated differently, and given different block numbers, making it difficult for the adversary to reconstitute the loop.



bad-block-good-block transformation: The server sends the client a buggy block A. The bug in A causes it to generate a wrong result, and, as a result, the client will not be able to analyze it in isolation. To maintain semantic equivalence one or more fixup blocks B have to be executed before A's result is used, i.e. B must post-dominate A. In this example, the bug in the blue bad block is corrected by the two green fixup blocks before the value of `k` is used by the pink block:



The block scheduler can complicate analysis by arranging for the buggy block and the fixup blocks not to be in the client's bag at the same time.

API mutation transformation: It's essential that the client cannot easily ignore new blocks sent to it by the server. The idea behind API mutation transformations is to modify the client code in the bag to effect a client-server API change. These are important transformations since the client cannot ignore them if he wants to continue getting service from the server. The RPCs the client makes to get service from the server can, for example, be renamed, arguments can be reordered, bogus arguments can be added, argument types can be changed, and calls can be split in two.

In an ideal scenario the client cannot tell one kind of block from another — specifically, it shouldn't be able to tell which blocks effect an API change. Knowing that *any* block could alter the API will force the client to accept *all* new blocks pushed to it by the server.

2.3 Tweaking the knobs

The main advantage of this model of remote tamper-resistance is that we have several knobs that we can tweak to balance the level of protection vs. performance degradation:

1. We can make the client bag smaller in s size to decrease the portion of the program available to the client to analyze at any one point in time, at the cost of increasing network traffic.

2. We can increase the rate of block push (the server telling the client to invalidate an old block and replace it with a new one) in order to increase the amount of analysis work the client has to do, at the cost of increased network traffic.

3. We can choose different mixes of blocks and block transformations, since each kind is likely to have different performance and protection characteristics.

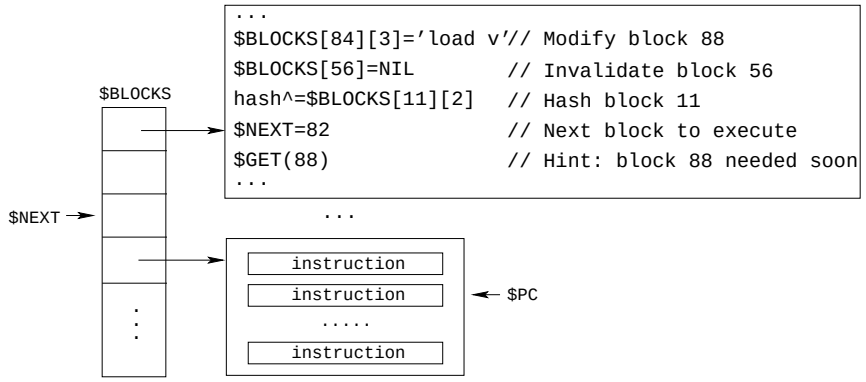
4. We can make blocks more or less obfuscated using any of the standard obfuscation algorithms, more obfuscating leading to longer analysis times but worse client performance.

5. We can vary the block size by making each block comprising one or more basic blocks, functions, or modules. Smaller blocks will increase the required analysis work but decrease client performance.

There are also several ways in which the server can detect that the client has tampered with its bag of blocks: a measuring or oblivious hashing block could return an unexpected value, the client could make a remote procedure call using an expired API, or the client could fail to respond to a *ping* after having crashed due to trying to execute an unexecutable block.

2.4 Block design

To make analysis difficult and slow, we should try to make the blocks as similar as possible: padded to have the same size, obfuscated to have the same structure, etc. This increases stealth and makes it hard for the attacker to know which blocks it can ignore and which it has to execute. If we allow blocks to access the virtual machine itself we can even get away with just one block kind. In this case, the VM consists of the interpretation loop and three variables, `$BLOCKS` (the bag of blocks), `$NEXT` (the next block to execute), and `$PC` (the current location):



3 Java prototype

A Java prototype called biànliǎn is currently under construction. A complete implementation would include a state-of-the-art obfuscator (such as SandMark [6]), but at the current time all obfuscating transformations are done by hand.

3.1 Organization

The main concept in the implementation is the `Block`. Every block inherits from an `AbstractBlock` class:

```
public interface AbstractBlock {
   public Object[] exec(Client host, Object[] args);
   public int next(Object[] args);
   public int[] dependencies();
   public ClientFrame getACK();
}
```

The `exec` method runs the code associated with a block, given an array of arguments, and returns an array of results. The `next` method tells the block interpreter which block it should jump to after executing the current block. The `dependency` method returns an array of block tags, indicating which blocks should be loaded into the client's block bag in order for the current block to execute successfully. Lastly, the `getACK` method returns a value the current block wishes to communicate to the server: a hash value, etc.

A particular server is dedicated to the maintenance of a particular program C; that is, the server keeps track of all the basic blocks associated with C. In our current prototype implementation, there are several different versions of C stored in this server, each with its own set of blocks. At any moment, the server can perform a version switch, changing the set of block available to the client. The structure of the block server can be divided into three parts: a request-reply module, a push module, and a block bag. The request-reply module listens for client requests for blocks, and satisfies these requests based on the current program version. The push module periodically sends groups of blocks to the client without request, to facilitate version switches, add functionality to the client, etc. Lastly, the block bag holds all blocks associated with C, organized into different versions.

The client side of this exchange — the executor of C — has three complementary parts: a block bag, an interpreter module, and a push-receive module. The client's block bag is a lookup table which caches program blocks received from the server based on their tags. The interpreter is responsible for controlling the flow of the program: this involves requesting blocks from the server, executing blocks, passing arguments between blocks, resolving block dependencies, etc. The push-receive module listens for blocks pushed across from the server, and incorporates such blocks into the block bag.

Here's a typical client-server exchange: the server S powers on, and sets its current program version to 0. S starts to listen for clients. A client C connects to S, and S sends across the first set of blocks associated with C. C executes these blocks until it requires more, and then queries the server for the blocks it needs. At some point the server decides to change the program version, and thus pushes a new set of blocks over to the client, replacing some or all of its current block bag. In our current implementation versions are hardcoded. Future implementations will instead generate them dynamically at runtime.

3.2 Hashing and Measuring blocks

Let's look more closely at these basic blocks. Given some program C, a basic block encapsulates some small aspect of C's code: for example a procedure call. The client-side interpreter executes some number of basic blocks in order to achieve the full functionality of C. Every block is assigned a tag (a 32-bit integer) at time of creation. This tag is used as a key to store blocks in the server and client lookup-tables (their block bags).

Each block is a Java class which implement the `AbstractBlock` interface. This interface serves to define certain important aspects of a block, namely:

1. A block must contain some amount of code which can be executed by the client;
2. A block may require other blocks to be present in the client's bag before executing (some block a might invoke code in block b, and thus we say a depends on b);
3. After it finishes executing, a block should be able to decide which block should execute next;
4. A block should be able to send data to the server, for example to report on the client's behavior.

(1) is represented by the `exec` method in the `AbstractBlock` interface: the client invokes a block by calling this method with some number of arguments. (2) is represented by the `dependencies` method, which returns a list of block tags, indicating blocks that must be present in the client's bag before `exec` can be called successfully. (3) is represented by the `next` method, which returns the tag of the block which should be executed next. This is not a static decision; that is, a block can dynamically decide which block to jump to next, based on the current state of the program. Lastly, (4) is represented by the `getACK` method, which returns a piece of data the block wishes to impart to the server. This data might represent some hash computation of another block, or some other measurement of the client. The client is expected to send this data to the server after the block finishes executing.

Here is an example of the `exec` method of a simple *measuring block*:

```
public Object[] exec(Client host, Object[] args) {
   long start = System.currentTimeMillis();

   int s = 0;
   for(int x = 0; x < 10000; x++) {
      s+=x;
   }

   long finish = System.currentTimeMillis();
   ACK.setData(finish - start);
   return null;
}
```

This block simply times some operation, and returns the result to the server for analysis through the ACK.

Here is an example of a *hashing block*:

```
public Object[] exec(Client host, Object[] args) {
   BlockBag bag = host.clientBlockBag();
   Block blk = bag.get(3);
   byte[] bytes = blk.getCode().getClassBytes();
   int hash = 0;

   for(byte b : bytes) {
      hash ^= b;
   }

   ACK.setData(hash);
   return null;
}
```

This block fetches another block from the client's bag, computes a simple `xor` hash over the bytes of that block, and return the result to the server for analysis through the ACK.

Here is an *oblivious hashing block*, which tests a block's functionality:

```
public Object[] exec(Client host, Object[] args) {
   BlockBag bag = host.clientBlockBag();
   AbstractBlock run = bag.get(10).getRef();

   String s = "a b c d e g h";
   Object[] answer = run.exec(host, new Object[]{s});

   ACK.setData(answer[0]);
   return null;
}
```

We fetch a block from the client's bag and run it with some argument provided by the server. We then return the answer to the server for analysis through the ACK.

3.3 Unexecutable blocks

Here is a trivial *unexecutable block*:

```
   public Object[] exec(Client host, Object[] args) {
      int answer = 5/0;
      Object[] result = {answer};
      return result;
   }
```

Blocks of this type are present to make it more difficult for an attacker to analyze blocks in isolation. In some cases we can sneak functionality into these unexecutable blocks. For example, in a Java Swing application, we could have a block like the one above serve as an `ActionListener`. The `exec` method would never be invoked by the interpreter, but the block would still (somewhat passively) contribute to the program's execution.

3.4 Server-side execution and API mutation

A *server-side block* is implemented through RPC calls, such as the following (used by the client):

```
currentWord = (String)rpc.invoke("getNextWord");
```

These calls can be changed between versions to reflect changes in the API, such as a change in the function name or a change in the number of arguments. The `invoke` method in the RPC system is very flexible, with the header:

```
public Object invoke(String name, Object...args)
```

Here are examples of two different stubs used by the API mutator. The only difference between the two is the method name.

```
private class NextWordV1 extends Stub {

   public NextWordV1() {
      super("nextWord", 0);
   }

   public Object execute(Object[] args) {
      return words[rand.nextInt(words.length)];
   }
}

private class NextWordV2 extends Stub {

   public NextWordV2() {
      super("getNextWord", 0);
   }

   public Object execute(Object[] args) {
      return words[rand.nextInt(words.length)];
   }
}
```

To change the API on the server side we use this method:

```
public void mutateAPI() {
   if(version == 0) {
      rpc.removeStub("nextWord");
      rpc.addStub(new NextWordV2());
      version = 1;
   } else if(version == 1) {
      rpc.removeStub("getNextWord");
      rpc.addStub(new NextWordV1());
      version = 0;
   }
}
```

This assumes two versions where the server alternates between the two. In a complete implementation we would generate an infinite, non-repeating sequence of APIs.

3.5 Block mutation

A block mutation makes changes to blocks already sent to the client. This can be done by pushing to the client a special block which, when executed, changes the code within one

or more other blocks in the block bag. In native binaries, this operation involves directly accessing and modifying code loaded in memory. Unfortunately the design of the Java Virtual Machine prevent such a direct manipulation of a loaded class. While any block has access to the class definition of every other public block, the change must be made by altering the class definition and reloading the desired class. This change must be carried out without causing calls from existing methods to fail.

Suppose block B wishes to transform some block A into a mutated block A'. Ultimately, this operation can take on several forms:

- Block B modifies the class definition of A, producing A'. The original definition for A is unloaded from the runtime system; A' is assigned the same tag as A, and is its definition is loaded.
- Block B modifies A, producing A'. A' is assigned a different tag and is loaded into the runtime system. A is optionally unloaded.

Either of these alternatives require the use of a modified class loader which supports dynamic loading and unloading of classes. Mutation can be achieved by a mutation block which gets the current definition of a block, modifies its bytecode, unloads the old instance and loads a new one. One way to achieve this is to take advantage of Sun's implementation of java which includes the `sun.tools.javac` package. This package contains a complete java compiler. If all client blocks carry their own obfuscated source code, a mutation block could modify their source and use this package to create new mutated instances of a method. More directly, bytecode editing libraries such as BCEL [1] and BLOAT [12] provide a simple API for a mutator block to directly modify and load a method. These APIs make it simple to alter the bytecode in a method substituting `iadd`s for `isub`s or changing the number and type of parameters a method takes.

There are two special cases with block mutation - complete block replacement and block deletion. In block replacement the client creates a new block C which contains the changes it wishes to make, and pushes it to the client. Block replacement can be used to achieve the same effect as block mutation without requiring client side java editing libraries. Nevertheless this is an unattractive option. One of the intents of mutation is to decrease the amount of network traffic required to introduce entropy in the client, however, replacement rather than mutation *increases* the amount of network traffic.

Here is a brief practical example of block replacement. We have a block (with id 9) in Version 0 which does the following:

```
public int next(Object[] args) {
   if(gameAlive)
      return 9;
   else
      return Block.END_BLOCK;
}
```

Thus, this block just loops until the game is over. To move to the next version (version 1), we prepare a block on the server side, also with id 9, and push it over:

```
HangMan2 = new Version("Hangman2",9,1);
List<Integer> T = new ArrayList<Integer>();
T.add(9);
HangMan2.setTransformers(T);
...
server.pushVersion(1, PushConsts.NO_JUMP);
```

When the server calls pushVersion, the "transformer" blocks in version 1 are pushed to the server. In this case, block 9 is pushed to the server, replacing the current, looping block, and making it do something else (prepare to move to another version, in this case, but any sort of obfuscation/change can be applied). `The NO_JUMP` argument specifies that the client's interpreter should make no unexpected changes to block execution order, but just proceed normally.

Block deletion is much simpler to implement. A block can remove blocks from the client simply by requesting the block be removed from the client block bag as follows:

```
BlockBag bag = host.clientBlockBag();
bag.remove(9);
bag.remove(10);
bag.remove(11);
bag.remove(12);
```

Of course the server cannot *force* the client to execute any type of block mutation, replacement or deletion. However, unless the client correctly executes these mutations, future blocks may execute incorrectly or not at all.

4 Implementation Challenges

The choice of programming language and the target platform can directly affect the difficultly and feasibility of implementing continuous replacement. A prototype implemented in a language like Java with exceptions, multi-threading and exceptions poses an unusual set of challenges.

4.1 Rewriting Method Calls

Java does not allow the direct manipulation of memory. As a result, loading arbitrary pieces of code into memory and executing them directly is not possible. Instead the server encapsulates and sends each `Block` inside a class. The client uses a custom class loader to instantiate an instance of this class and add it to the block bag. The `runBlock (blockNum, args)` method in the client then selects the appropriate block from bag and executes it.

When a program is decomposed into blocks, different methods end up in different blocks. As a consequence, method calls are no longer guaranteed to succeed because the target of a method invocation may not yet be loaded into the clients block bag. We show this in Figure 2.

To overcome this problem, we replace all method calls with calls to the interpreter requesting a particular block be executed. The block that is requested is the first block of the method.

```
public class MethodCalls {
    public void b () {
      ...code of method b...
    }
    public void a () {
      b();
    }
}
```

```
public class BlockA {
  Object[] exec(Client h, Object[] args) {
    interpreter.runBlock("BlockB'', null);
  }
}

public class BlockB {
  Object[] exec(Client h, Object[] args) {
      ...code of method b...
  }
}
```

Figure 2: Making method calls. In (a) method `a()` calls a method `b()`. In (b) these methods are sent to the client in two separate `Blocks`. The interpreter must ensure that `BlockB` is loaded before calling its `exec()` method. This check is carried out by replacing the direct call to `BlockB.exec()` with a call to the interpreter's runBlock method which carries out these checks.

The original method call `foo(args)` is replaced with `runBlock(22, args)` where 22 is the number of the block containing the instructions of `foo`. The `runBlock` method checks if the corresponding block has been loaded and if not, requests the block from the server. It then executes the block with argument `args`.

This technique doubles the number of method calls executed by a program and as a result halves the number of recursive calls which are possible on a machine with any given stack size. Nevertheless this technique has the advantages that it correctly handles inter-block exceptions.

4.2 Handling Exceptions

An exception is an event during the running of a program which disrupts the normal flow of execution. When an exception is *thrown*, the JVM packs information about the state of program into an object. Also included in this object is information about the event which caused the exception to be thrown, as well as its type. The JVM then searches up the call stack for a matching exception handler. If such a handler is found it is executed and the execution resumes immediately following the handler. This means that the remainder of the code in a protected block after an exception is never executed.

As shown in Figure 3, this model of exception handling can introduce errors in a continuous replacement system. In the figure, two simple methods are shown. In this example we assume that each method is served to the client in separate blocks. The block `b()` consists of a `try-catch` block containing an `if`-statement which may throw either a `IOException` or a `NullPointerException`. Only one of these exceptions is caught and handled by block `b()`.

The block `a()` calls block `b()` and thus must handle its uncaught exception. There are several alternatives to correctly manage this scenario in continuous replacement. One method is to ensure that all exceptions thrown in a block are handled within the block. In this case, all blocks can be considered independent by the continuous replacement client.

```
public class Exceptions {

    public void b () throws NullPointerException {
        try {
            if ( Math.random() > .5 )
                throw new IOException ( "oops" );
            else
                throw new NullPointerException ( "eek" );
        } catch ( IOException e ) {
            System.out.println ( "IOException caught in b()" );
        }
    }

    public void a () {
        try {
            System.out.println ( "a()" );
            b();
        } catch ( NullPointerException e ) {
            System.out.println ( "NullPointerException caught in a()" );
        }
    }

    public static void main ( String args[] ) {
        Exceptions e = new Exceptions();
        e.a();
    }
}
```

Figure 3: Splitting up exception handling. Method `b()` throws some exceptions which caught and handled in a parent method, `a()`. If methods `a()` and `b()` are split into different `Blocks`, the interpret must ensure that any uncaught exceptions are propagated correctly.

Unfortunately, this is not practical because many real-world programs handle exceptions at a location far from where they are thrown. Furthermore, this would result in very large blocks being delivered to the client.

Given that all method calls are now translated into calls to the interpreter, no transformation is needed to handle exceptions. Instead we ensure that the `runBlock` method catches no exceptions. All exceptions which are not handled by the current block get passed up to the interpreter and in turn up to the previous block which was executed until a block is found which handles the exception or the program terminates.

4.3 Multi-threaded Programs

Since Java supports multi-threaded applications a block which is executed by the client may contain a `new Thread().start()` instruction which results in a new thread in the client. When the current block completes execution, it returns control to the interpreter. However, the new thread it launched will continue execution. This second thread will not have the means of ensuring subsequent blocks are loaded nor of passing control to these blocks.

```
public class Multithread {
    public void c () {
      ...code of method c...
    }
    public void b () {
      c();
    }
    public void a () {
      new Thread(
        public void run () {
          b();
        }).start();
    }
}
```

```
public class BlockA {
  Object[] exec(Client h, Object[] args) {
    Interpreter i = new Interpreter();
    i.runBlock("BlockB'', null);
    i.start();
  }
}

public class BlockB {
  Object[] exec(Client h, Object[] args) {
    interpreter.runBlock("BlockC'', null);
  }
}

public class BlockC {
  Object[] exec(Client host, Object[] args) {
      ...code of method C...
  }
}
```

Figure 4: Multithreaded programs. In (a) method `a()` starts a new thread and executes `b()` which in turn executes `c()`. In (b) the methods `a()`, `b()` and `c()` are sent to the client in separate `Blocks`. The interpreter must ensure that `BlockB` is able to load and execute its dependents. This check is carried out by replacing construction of a new `Thread` with the construction of a new `Interpreter`

We handle multi-threaded programs in a similar way to method calls. We create a new instance of the interpreter with access to the original block bag and pass it the task of executing blocks in a new thread. This involves rewriting all calls to `new Thread()` with calls to `new Interpreter()`. The `run()` method of the original thread is rewritten as a new `Block` which is executed by the new instance of the interpreter.

There are some advantageous side effect of this design decision. The Java verifier places restrictions on how monitors are used in an application. In particular the verifier requires all monitor locks which are acquired during a method call are released before the method returns. Provided that the server maintains entire synchronized blocks or methods inside a single `Block`, this restriction is automatically maintained.

In practice synchronized methods or blocks are usually small pieces of code and maintaining an entire synchronized block or method in one `Block` is not a large restriction. In a future version of this paper, we will describe how to factor a synchronized block such that it can be split between two or more client `Blocks`.

5 Related work

In intermediate solution between running all code server-side with a high degree of protection from tampering, and running all code client-side with *no* guarantees at all, is for some code to execute server-side and some client-side. Zhang and Gupta [18] is an example of a system which makes use of *slicing* to split the program in two.

The scenario we're discussing in this paper is completely general — we assume nothing special about the server, client, or the connection between them. We think this is the most interesting scenario since it models a general distributed system, such as the Internet. In more restricted situations it's possible to design remote tamper-resistance systems with stronger guarantees. The Pioneer system [13,14], for example, assumes that the server has complete knowledge of the client (the CPU model, the memory size, memory latency) and the connection between them. It furthermore assumes that the client cannot communicate with any other system during an authentication phase. This allows the server to *measure* crucial aspects of the client (essentially by computing a hash over security sensitive code) to ensure that it runs untampered. This type of system is suitable for securing private networks such as military systems or connections between banks and ATMs.

The Genuinity [10] system tries to extend this idea to general distributed systems. There is, however, much controversy as to whether this actually works or not [11,15,16].

6 Conclusion and future work

In theory it is simply not possible to remotely guarantee a machine executes code which you send to it exactly without some additional hardware. For example, it is trivially apparently that any remote machine can execute additional instructions. Even server side execution of code cannot guarantee that input provided by a legitimate user on a potentially malicious remote machine is transmitted to a trusted server unaltered. Nevertheless complete server-side execution gives the server the strongest guarantee of the integrity of execution and confidentiality of its code.

Unfortunately the prohibitive cost of server-side execution on the server makes it unscalable and thus unusable in many applications. The system we describe gives an application a tunable tradeoff between a secure but computationally expensive complete server-side execution and insecure but computationally cheap complete client-side execution. Moreover, the design allows a continuous change between these two extremes allowing the server to switch between more or less security depending on the perceived level of threat without having to re-deploy or even change large parts of the system.

The current prototype describes only the interaction between a trusted server and an untrusted client. In future we hope to continue to improve the scalability of such systems by taking better advantage of the network. While the model we present requires less work for the server than complete server-side execution, as the number of clients increases, the amount of state and computation which the server must perform also increases linearly. We can take advantage of the fact that in many scenarios the number of non-malicious clients will largely outnumber the number of malicious ones. As a result, some of the transformations currently being performed can be sent to other clients.

For example, in Figure 5 we show that if there are two clients connected to a server, the server could send a fraction of the blocks to each client C_1 and C_2 while keeping the remaining fraction in its own cache. Each client would keep the data structures which are required to maintain its own state as well as those of the clients which are connecting to it. When any client requires the next block, it would send out a request to all other clients as well as to the server. The client which has the required block would then perform any obfuscating transformations which were required before delivering the requested block.

The load on the server in this scenario is considerably reduced. We continue to maintain the security guarantee where no one client has all of the source code for the program.

Nevertheless, if a large number of clients collude they may be able to gain access to a larger part of the program. Some of this risk maybe mitigated by performing some obfuscation on the blocks before delivering them to the individual clients. We may also be able to take advantage of the fact that all clients receive client requests for the next block to distribute the load of checking the correctness of clients.

In this report we've described an implementation for Java programs distributed as collections of class files. Ultimately, it would be more interesting to study protection of native code clients. We're considering building such a system on top of the Dynamo [2] dynamic optimization system, as it already supports the concept of blocks, block bags, and block transformations.

References

[1] BCEL, February 2004. jakarta.apache.org/bcel.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, 2000.

[3] Shane Balfe and Anish Mohammed. Final fantasy - securing on-line gaming with trusted computing. In Bin Xiao, Laurence Tianruo Yang, Jianhua Ma, Christian Müller-Schloer, and Yu Hua, editors, *ATC*, volume 4610 of *Lecture Notes in Computer Science*, pages 123–134. Springer, 2007.

[4] H. Chang and Mike Atallah. Protecting software code by guards. In *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001*, Philadelphia, PA, USA, November 2001. Springer Verlag, LNCS 2320.

[5] Y. Chen, R. Venkatesan, M. Cary, R. Pang, and S. Sinha andM. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *5th Information Hiding Workshop (IHW)*, pages 400–414, October 2002. Springer LNCS 2578.

[6] Christian Collberg, Ginger Myles, and Andrew Huntwork. SandMark — A tool for software protection research. *IEEE Magazine of Security and Privacy*, 1(?), August 2003.

[7] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages 1998, POPL'98*, San Diego, CA, January 1998.

[8] Christian S. Collberg and Clark Tomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. *IEEE Transactions on Software Engineering*, 8(8), 2002.

[9] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001*, Philadelphia, PA, USA, November 2001. Springer Verlag, LNCS 2320.

[10] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *12th USENIX Security Symposium*, pages 295–308, 2003.

[11] Rick Kennell and Leah H. Jamieson. An analysis of proposed attacks against genuinity tests. Technical Report 2004-27, Center for Education and Research in Information Assurance and Security, 2004.

[12] Nathaniel Nystrom. BLOAT – the Bytecode-Level Optimizer and Analysis Tool. www.cs.purdue.edu/s3/projects/bloat, February 2004.

[13] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Externally verifiable code execution. *Commun. ACM*, 49(9):45–49, 2006.

[14] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. *SIGOPS Oper. Syst. Rev.*, 39(5):1–16, 2005.

[15] Umesh Shankar, Monica Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[16] Umesh Shankar, Monica Chew, 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.

[17] Jeff Yan and Brian Randell. A systematic classification of cheating in online games. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Networkand system support for games*, pages 1–9, New York, NY, USA, 2005. ACM.

[18] Xiangyu Zhang and Rajiv 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.

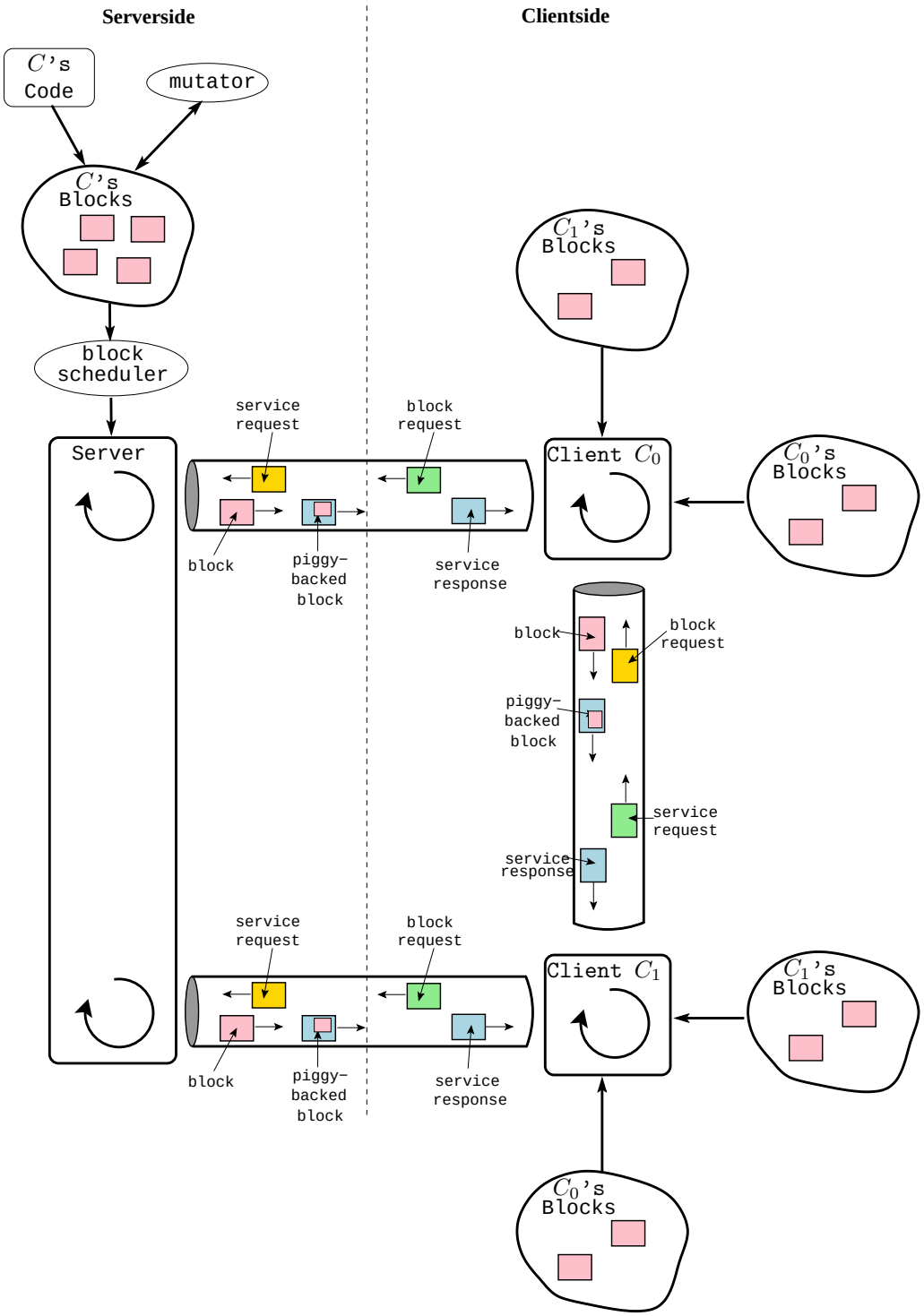


Figure 5: Overview of a distributed remote tamper-resistance architecture.