ABSTRACT

We present a new general technique for protecting clients in distributed systems against Remote Man-at-the-end (R-MATE) attacks. Such attacks occur in settings where an adversary has physical access to an untrusted client device and can obtain an advantage from tampering with the hardware itself or the software it contains.

In our system, the trusted server overcomes the analytical abilities of the untrusted client by continuously and automatically generating and pushing to him diverse client code variants. The diversity subsystem employs a set of primitive code transformations that provide an ever-changing attack target for the adversary, making tampering difficult without this being detected by the server.

Categories and Subject Descriptors
K.6.5 [Security and Protection]
C.4.2 [Computer Systems Organization]: Distributed systems
D.4.1 [Software]: Design tools and techniques
D.4.6 [Software]: Analysis

General Terms
Renewability, Defense-in-depth, Diversity, Obfuscation, Tamper-proofing

Keywords
Distributed Systems, Security, Software Protection

1. INTRODUCTION

Man-at-the-end (MATE) attacks occur in settings where an adversary has physical access to a device and compromises it by tampering with its hardware or software. Remote man-at-the-end (R-MATE) attacks occur in distributed systems where untrusted clients are in frequent communication with trusted servers over a network, and malicious users can get an advantage by compromising an untrusted device.

To illustrate the ubiquity of R-MATE vulnerabilities, consider the following four scenarios. First, in the Advanced Metering Infrastructure (AMI) for controlling the electrical power grid, networked devices (“smart meters”) are installed at individual households to allow two-way communication with control servers of the utility company. In an R-MATE attack against the AMI, a malicious consumer tampers with the meter to emulate an imminent blackout, or to trick a control server to send disconnect commands to other customers [7, 22]. Second, massive multiplayer online games are susceptible to R-MATE attacks since a malicious player who tampers with the game client can get an advantage over other players [17]. Third, wireless sensors are often deployed in unsecured environments (such as theaters of war) where they are vulnerable to tampering attacks. A compromised sensor could be forced into supplying the wrong observations to a base station, causing real-world damage. Finally, while electronic health records (EHR) are typically protected by encryption while stored in databases and in transit to doctors’ offices, they are vulnerable to R-MATE attack if an individual doctor’s client machine is compromised.

1.1 Overview

In each of the scenarios above, the adversary’s goal is to tamper with the client code and data under his control. The trusted server’s goal is to detect any such integrity violations, after which countermeasures (such as severing connections, legal remedies, etc.) can be launched.

Security mechanisms. In this paper we present a system that achieves protection against R-MATE attacks through the extensive use of code diversity and continuous code replacement. In our system, the trusted server continuously and automatically generates diverse variants of client code, pushes these code updates to the untrusted clients, and installs them as the client is running. The intention is to force the client to constantly analyze and re-analyze incoming code variants, thereby overwhelming his analytical abilities, and making it difficult for him to tamper with the continuously changing code without this being detected by the trusted server.

Limitations. Our system specifically targets distributed applications which have frequent client-server communication, since client tampering can only be detected at client-server interaction events. Furthermore, while our use of code diversity can delay an attack, it cannot completely prevent it. Our goal is therefore the rapid detection of attacks: applications which need to completely prevent any tampering of client code, for even the shortest length of time, are not suitable targets for our system. To see this, consider the following timeline in the history of a distributed application running under our system:
The $e_i$’s are interaction events, points in time when clients communicate with servers either to exchange application data or to perform code updates. At time $t_1$ the client tampers with the code under his control. Until the next interaction event, during interval $I_1$, the client runs autonomously, and the server cannot detect the attack. At time $t_2$, after an interval $I_2$ consisting of a few interaction events, the client’s tampering has caused it to display anomalous behavior, perhaps through the use of an outdated communication protocol, and the server detects this. At time $t_3$, finally, the server issues a response, perhaps by shutting down future interactions. Programs in which the client does not need to frequently communicate with the server are not suitable to our system.

Adversarial model. We place very few limiting assumptions on the adversary’s abilities. First of all, we expect that, through reverse engineering, he has achieved a complete understanding of our system. We also expect him to have at his disposal all the common static and dynamic analysis techniques that would be helpful in analyzing and tampering with the code under his control, including debuggers, tracers, disassemblers, decompilers, control- and data-flow analyses, code slicers, etc. The level of protection against tampering that our system affords thus does not rely on security-by-obscurity but rather on the server’s ability to deliver to the client a steady stream of random code mutations, at a rate that exceeds the client’s ability to reverse engineer. To ensure this, all of our transformations are randomized, and many are based on hard static analysis problems, such as the need to perform pointer analysis.

Security vs. performance trade-offs. Some of our code transformations incur much overhead on the client side (we can, for example, transform a function by applying multiple levels of randomized interpretation), and some incur overhead on the server side (generating streams of code variants for multiple clients). Our system is therefore completely configurable, allowing the server to trade-off between its ability to detect client tampering, client performance, and server performance.

2. BACKGROUND AND RELATED WORK

Fred Cohen was the first to suggest that code transformations could be used to create a diverse set of programs which would be less vulnerable to attack [8]. The intuition is that if every installation of, say, an operating system, is different, adversaries (human intruders as well as automated malware) will find it more difficult to reuse an attack that was successful against one variant when attacking another. Fritz Hohl may have been the first to suggest (in the context of protecting mobile agents) time-limited protection against MATE attacks [18]. The intuition is that an adversary who is in complete control over his attack target may still be kept at bay if his window of opportunity is sufficiently short; i.e. if the time needed to analyze and tamper with the target is strictly longer than the time the target is of value to the attacker. In this paper we combine these two ideas: we protect vulnerable code running on an untrusted client in a distributed system by diversifying the code and, by continuously and automatically replacing that code, we ensure that the client has a short time window to analyze and tamper with it.

Code diversity. Since Cohen’s original paper, several systems have been developed to randomize various aspects of computer systems to protect against attacks: opcodes can be randomized and then executed under emulation/interpretation [5, 1]; modules, stacks, and heaps can be placed in random location in memory (Address Space Layout Randomization, ASLR [31]); and code can be reordered and stack frame layouts can be randomized [15]. Other than ASLR, these ideas have not found their way into production systems. The reasons are many, but include unacceptable performance overhead and challenging software engineering problems, such as how one would approach software testing, error reporting, and perform updates on widely divergent programs.

Various forms of code randomization have also been used to protect programs against MATE attacks, i.e. to make programs more difficult to reverse engineer, tamper with, and redistribute illegally [9]. Common techniques include obfuscating code transformations that reorder code to make it difficult to analyze, tamper-proofing transformations that reorder code at runtime to make it difficult to modify [4], and software watermarking techniques that embed unique identifiers in code, using compilers that randomize code-generation choices [2]. Such MATE protection techniques have mainly found applications in intellectual property protection of computer programs and digital media.

MATE attack models. The fundamental difference between MATE and other computer security scenarios is that a true MATE adversarial model cannot assume an unassailable root-of-trust. Because the adversary has physical access to the software and hardware of the device he tampers with, we have to assume that, given enough time, all defenses will be compromised. This is true even if the device is equipped with tamper-resistant hardware—experience has shown that such devices are eminently vulnerable to invasive as well as non-invasive attacks [3, 12]. This makes the problem of defending against MATE attacks significantly more challenging than for more traditional security scenarios.

In analogy with Kerckhoffs’s principles for cryptography, in designing defenses against MATE attacks we must assume everything about our system is available to the attacker for study, including primitive code transformations, strategies for combining such transformations, and in fact, the source code of the entire system. What is not available to the attacker, however, and the only leverage we as defenders have, are the actual randomization seeds used in a particular run of the system. This means that the attacker cannot easily predict the order in which transformations are applied or the location in the code where they are applied.

R-MATE defense strategies. The Trusted Platform Module (TPM) chip has been suggested as a solution to the R-MATE problem, by ensuring that the untrusted client’s software and hardware are approved and untampered [23]. This solution, however, has been shown to be impractical, requiring comprehensive white-lists of trusted software to be kept up-to-date at all times.

The CodeBender system [6] is the most closely related to our work, in that it uses code updates for security. Unlike our system, however, they rely on a single code transformation and require clients and servers to both be completely shut down and restarted for every update. For many distributed applications this is unacceptable.

Also related to the work presented here is the work of Stoyles et al. [29] who do run-time code updates, but in their case for servers, not clients, and not for security. Haebeler et al. [16] also detect code tampering in distributed scenarios, but with a much more significant time delay until detection. The Pioneer system by Seshadri et al. [27] detects code tampering in a system where exact system specifications and latency of the network must be fully known, whereas our system works on general systems and networks such as the Internet.
The work by Scandarici et al. [24] also attempts to solve the issue of remote tamper-detection via software. They generate proofs that the code has been untampered, potentially by checksums. Their work doesn’t make modifications to the code of the target application and obfuscations are not utilized. Falcarin et al. [14] break a program into pieces and send each over individually when the target requires them. The program is loaded into random memory locations, but no obfuscation is done at the function level.

### 3. CORE MECHANISMS

Our system (shown in Figure 1) consists of a trusted server and (one or more) untrusted clients. The input to the server is the code to be executed on the client side (client.c), the server code itself that services requests from the client (server.c), and profiling data for the client. The server contains a set of code and data blocks extracted from client.c, a set of primitives (block transformations), a set of diversity strategies for combining such primitives, and a diversity graph that expresses the relationships between every generated block variant. The system can easily be extended with new primitives.

The client holds a subset of the blocks that make up client.c in its block bag. The adversary’s intent is to analyze and modify some of the blocks in the bag to meet his attack goal. The server’s intent is to detect when it happens, and then to punish the client accordingly. The server’s strategy is to accomplish this by making the block bag change continuously at runtime, forcing the client to invalidate old block variants and add new ones, thereby stressing his analytical ability.

Over the course of execution the client’s block bag will be filled with a sequence of block working sets, \(\{w_0, w_1, \ldots\}\). Each \(w_i = \{... f_{j,k} ...\}\) is a set of blocks which, taken together, forms a complete variant of the original client.c. A block \(f_{j,k}\) is a variant of a function \(f\) with protocol version number \(j\) and implementation version number \(k\). At any one point in time, the client may only see a subset of the current working set, namely the set of blocks it actually needs for execution. Moving from working set \(w_i\) to \(w_{i+1}\) normally only changes a subset of the blocks, i.e. it is the result of applying a few primitive code transformations to a few of the blocks in \(w_i\). In Figure 1, for example, it appears that an initial working set was \(w_0 = \{\text{main0}, \text{foo0}, \text{g0}\}\) followed by a transformation of \(\text{foo}\), yielding a second generation working set \(w_1 = \{\text{main0}, \text{foo0}, \text{g0}\}\).

There are three concurrent tasks occurring at runtime. The first task (① in Figure 1), is the “normal” client-server interaction that would occur in an unprotected distributed application, where the client executes code out if its block bag and makes remote procedure calls (RPC) to the server whenever this is needed in order to make progress. The code in server.c continuously listens for and services such requests. In the second task (②), the server’s block scheduler supplies the client with blocks it requests or pushes blocks to the client that it predicts the client may need. In the third task (③), finally, the server’s diversity scheduler informs the client’s overseer process that a new update cycle is about to begin, the client returns its current state (blocks on the current call stack, which we will refer to as the active set) to the server and suspends itself. The server then generates new block variants by selecting appropriate primitives and strategies, invokes the code transformer to apply these transformations, and, once done, wakes up the client to inform it of invalidated blocks.

The code transformer and the primitives are built on top of the CIL [20] system for C code transformations. CIL’s facilities for static analysis are needed to implement the primitive code transformations. At system startup time, client.c is parsed by CIL and each function and global variable (also called blocks since they are one-to-one with the blocks in the client’s block bag) is extracted and kept in memory in CIL’s intermediate code format. Blocks are compiled into dynamically linked libraries (.so files) which the block scheduler transfers to the client for execution. Each block (④ in Figure 1) contains four sections, each of which can be empty: zero or more global variables, zero or more unprotected (level-0) functions, an initialization routine (init) which is executed when a block first arrives in the client’s block bag, and a main routine exec which is executed when a function gets called.

![Figure 1: Overview of the diversity system.](image-url)
3.1 Client-Server Protocol

To illustrate the interaction between a server and a client, consider the timeline in Figure 2. Execution begins at ① where, after proper authentication, the client receives the initial block, containing global variables, level-0 functions, and main itself. Subsequently, the client can either explicitly request a block (②) that it needs to execute, or receive blocks in the background (③) that the server predicts that it will need. Whenever a block request is received by the server it performs a validation step: given what is known of the client’s current state (information gathered from active sets received during the update cycle, the sequence of block requests, and remote procedure calls the client has made), the server determines whether the client is in compliance.

At ④, the server initiates an update cycle. At ⑤, the client suspends execution and responds with the active set of blocks (the current call stack). At ⑥, the server checks that the active set forms a valid sequence of function calls, computes the forbidden set (blocks which cannot be obfuscated), selects blocks to be transformed and the transformations to be applied, applies the transformations and sends the invalid set (out-of-date blocks which the client should no longer execute) to the client. At ⑦, the client resumes execution, concluding the update.

For every incoming remote procedure call (⑧), the server also verifies the validity of RPC numbers and argument values.

3.2 Transformation Primitives

A primitive in our system is a code transformation that transforms a function or variable into a different form. The primitives serve three goals: to add confusion to the client code thereby taxing the adversary’s analytical abilities (obfuscation); to make it more difficult for the adversary to modify the client code (tamper-proofing); and to make it easier for the trusted server to detect any tampering of the client code (tamper-detection). These goals are not independent: adding confusion to client code makes static and dynamic analysis harder, and this in turn makes the adversary’s goal of modifying the client code while maintaining correctness more difficult.

3.2.1 Preserving Protocols

Transformations that only add confusion are, by themselves, not sufficient for our purposes. Consider, for example, a situation where, in accordance with Kerckhoff’s principles, the adversary has full access to our system and, through analysis, has realized that our primitives only add non-functional code to functions while otherwise preserving their behavior. He can then simply reverse engineer a function the first time he sees it, tamper with it at will, and then ignore any further code updates! Therefore, some of our primitives generate variants that are compatible with each other, while some do not. Our goal is for the system to add enough confusion such that the adversary cannot easily determine whether a code update pushed to it can be safely ignored or not. If our primitives are able to satisfy this goal, we will force the adversary to expend resources on fully analyzing (automatically, or ideally, manually) each new block that gets sent to him. Then, providing security against R-MATE attacks becomes a question of the system being able to sustain a block replacement rate that is high enough to over-tax the adversary’s analytical abilities, without causing too much computational overhead.

Thus, some of our transformations are protocol-preserving. This means they only change the body of a function, but the interface through which the function is called is preserved. Other transformations are non-protocol-preserving, meaning any code that calls a transformed function must be updated to use the new interface. Similarly, a primitive that modifies the encoding of a global variable does not maintain protocol compatibility—any function which references that variable needs to be updated to use the new encoding. We get the following definitions:

**Definition 1 (Protocol Compatibility).** $T$ is known as a protocol-preserving transformation if $p$ is a function or a remote procedure and $T(p)$ preserves $p$’s external behavior, its signature and the encoding of arguments and return values. $T$ is a protocol-preserving variable transformation if $p$ is a variable and $T(p)$ preserves $p$’s type and encoding.

Block variants are numbered with a protocol and implementation version number: `block_prot_impl`. Two blocks with identical protocol numbers (such as `foo_prot_0` and `foo_prot_1` in Figure 1) are protocol compatible.

3.2.2 Diversifying Primitives

Some of the primitives described here are adapted from obfuscating code transformations found in the literature. The adaptations parameterize the primitives such that a given transformation can generate multiple different variants from the same function. Transformations take a random number seed as input and use a PRNG to randomize choices they make.

It should be noted that all our transformations can be combined ad infinitum: a function can first be flattened and then turned into an interpreter, a second layer of interpretation can be added (using a different dispatch method and instruction set), etc. This is important for transformations which in themselves do not have a whole lot of opportunities for diversification, but which can be combined to generate a potentially infinite sequence of variants. It is the duty of the strategies in the diversification scheduler (see Section 3.4) to decide when and how to combine primitives.

3.2.3 Protocol-Preserving Primitives

Our current implementation supports four protocol-preserving transformations which only modify the body of functions, not the way in which they are invoked. The `interpret(f, seed)` transformation turns a function $f$ into a specialized interpreter for $f$. The seed is used to select a random method of instruction dispatch (currently supported are call, switch, direct and indirect threading [13]), to select a fraction of instruction pairs to merge into superoperators [21], to randomize opcode
assignment, and to select the mix of addressing modes (register and stack arguments).

The flatten($f$, seed) transformation removes nested control flow from a function $f$ by making every basic block a case in a switch statement nested inside an infinite loop [30]. The seed is used to randomize the order of the basic blocks inside the switch statement.

The split($f$, seed) transformation converts a function $f$ into two functions $f_1$ and $f_2$, now called from $f$. The seed randomizes the point in $f$ where the split is made. $f$ passes along its local environment (formal arguments and local variables) to $f_1$ and $f_2$:

```c
void f1 (int x) {
  int y;
  f1 (x);
  y;
  return;
}
```

The opaque($f$, seed) transformation inserts non-functional code protected by an opaque predicate [10]. This code can range from code that causes crashes, code that invalidates client data, or calls to fake functions which the server knows should never be requested. The seed randomizes the point in $f$ where the insertion is made, and also the type of transformation. This example shows four of the kinds of transformations currently supported ($S^n_{log}$ is an obfuscated version of $S_1$ into which a bug has been inserted, bogus is a randomly selected function, and ⟨args⟩ a list of random expressions):

```c
if (p \neq q) S_{1,1}
if (p \neq NULL) S_{1,2}
if (p \neq q)\{ bogus ⟨args⟩ \}
if (p \neq q) RPC ⟨args⟩
p\rightarrow q
q\rightarrow \text{next}.
```

Here, $P^n$ represents an opaque true predicate, i.e. a boolean expression that is difficult for an adversary to analyze. In our current implementation, opaque predicates are manufactured by creating and dynamically modifying linked structures. In the example above, $p$ and $q$ are pointers being moved around in a circular linked list subject to the invariant $p \neq q$.

### 3.2.4 Non-Protocol-Preserving Primitives

Our system currently supports four primitives that do not preserve protocol compatibility, namely rnd_args, RPC_encode, merge, and var_encode.

The merge($f_1$, $f_2$, seed) transformation combines two functions $f_1$ (args$_1$) and $f_2$ (args$_2$) into a new function with a signature of $f_1$ (args$_1$) | args$_2$, in which sel is used at the call site to distinguish between calls to the two functions.

The rnd_args($f$, seed) primitive randomly reorders $f$’s formal parameters and adds extra, bogus, forms.

We have extended C with a remote procedure call mechanism that allows for simple client-server communication. Remote procedure calls are identified by number and carries with them a sequence of scalar data items and return a scalar result. The transformation of RPC_encode($n$, seed) assigns a new random encoding of the $n$’th remote procedure call RPC($n$, args) by assigning it a new random RPC number, randomly reordering its arguments, and inserting bogus arguments. This is an important transformation since, should an adversary choose to ignore block updates, he may inadvertently execute an invalid block containing an RPC with an obsolete encoding, thereby alerting the server of the tampering.

The var_encode($g$, seed) transformation selects a random new encoding for a variable $g$ [11]. As a trivial example, a global integer variable $x$ could be replaced with a long variable $x_{new}$ whose encoding is $x + 1$. Any reference throughout the program to $x$ in the original code has to be updated in order to preserve correctness. When the var_encode($x$, seed) transformation is selected, not only do all blocks containing $x$ need to be updated, but we also need to allocate the new variable on the client side and initialize its value to the current value of $x$, plus one. This is done by sending the client an init block like 42.so in Figure 1.

The var_encode transformation is important in that an adversary who attempts to use an obsolete block may inadvertently use an old variable encoding. This could lead to client failure or, when the server is sent the old variable’s value in an RPC, detection by the server.

### 3.3 Diversity Graph

Our transformations generate a complex set of relationships between blocks: function signatures change, RPC arguments are permuted, global variables change in type and encoding, and so on, and these changes, in turn, can force changes to other blocks. While this complexity is good for creating confusion on the client side, it unfortunately also results in generations of block working sets with varying levels of compatibility.

To allow us to reason intelligently about the relationships between various block variants, we use an abstraction we call a diversity graph representing dependencies between blocks and protocols. The scheduler uses the diversity graph to determine which blocks to transform next and how a transformation applied to one block will force updates to other blocks.

In the initial program, the diversity graph is nearly identical to a conventional call graph, with the addition of nodes for global variables and edges between a function and the global variables it references. In addition to the nodes that represent function implementations, we also add nodes to represent function protocols. As transformations are applied the graph will grow with nodes for all new variable/function variants.

**Definition 2 (Diversity Graph).** A diversity graph $G$ for a program $P$ consists of the following protocol and implementation nodes, where $m$ is a protocol and $n$ an implementation number:

- an implementation node $f^i_m,n$ for each function variant;
- a protocol node $f^i_m$ for each function protocol;
- a protocol node RPC$(a)^{\nu}_{m}$ for each RPC protocol;
- a protocol node $v^n_{m}$ for each protocol of a variable $v$.

$G$ contains the following directed edges:

- $f^i_m \rightarrow f^i_m,n$ from a function protocol node to its variant implementations;
- $f^i_m,n \rightarrow g^n_{v}$ if $f^i_m,n$ calls one of $g^n_{v}$’s implementations;
- $f^i_m,n \rightarrow$ RPC$(v)^{\nu}_{m}$ if $f^i_m,n$ makes remote procedure call number $\nu$;
- $f^i_m,n \rightarrow v^n_{m}$ if $f^i_m,n$ references global variable $v^n_{m}$.

**Forward update cycles.** Figure 3 shows the effect on the diversity graph from a sequence of forward update cycles, where each new graph is the result of applying one primitive. Figure 3 (a) shows the initial graph for a program consisting of a global variable $g$ and two functions main and foo. Note that each function results in a single protocol node and a single implementation node. A global variable is represented by a single protocol node. Figure 3 (b) shows the graph after an application of the protocol-preserving primitive flatten, which adds a new implementation.
node \texttt{foo}_0^1. Next, we obfuscate \texttt{g}_0^i using the \texttt{encode_var} primitive, which generates a protocol-incompatible variant \texttt{g}_1^i. We are forced to furthermore generate a new variant \texttt{foo}_0^2 of \texttt{foo}_0^1 compatible with \texttt{g}_1^i. In Figure 3 (d), finally, \texttt{foo}_1^0 is created by applying \texttt{rnd_args} to \texttt{foo}_0^p, resulting in a new protocol node \texttt{foo}_1^p.

Rollback update cycles. For performance reasons it is not always possible to transform a block working set \texttt{w}_i into a new generation \texttt{w}_{i+1} by simply applying obfuscating primitives. Every such transformation incurs overhead and compounding transformations would eventually lead to unacceptable client performance. The diversity scheduler supports a rollback operation, in which working set \texttt{w}_{i+1} is the result of replacing some of the blocks in \texttt{w}_i with variants from previous generations. Since our primitives are parameterized, this will allow the system to first roll back a few blocks to previous variants, then roll forward by applying new transformations, and to do so without incurring any extra performance penalty.

3.4 Diversity Strategies

It is well known that software protection techniques (such as the obfuscating primitives in Section 3.2) deployed in the field have a limited survival time. Any static target will fall to a motivated adversary equipped with standard reverse engineering tools such as debuggers, tracers, disassemblers, and decompilers. We therefore make use of strategies, means of combining primitive transformations that gives the adversary not a static, but a constantly changing, attack target. We identify three strategies, based on the notion of diversity, essential to making effective use of software protection:

DEFINITION 3 (DIVERSITY). A program \textit{p} is temporally diverse if it is delivered to the user, over time, as an infinite and non-repeating sequence of variants \(\{v_0, v_1, \ldots\}\) where \(v_i \neq v_j\) if \(i \neq j\). A program \textit{p} is spatially diverse if a variant \(v\) is constructed by compounding multiple layers of interchangeable primitive transformations. A program \textit{p} is semantically diverse if two variants \(v_i\) and \(v_j\) of \(p\) cannot be used interchangeably.

Temporal diversity is sometimes known as renewability or software aging [19], and spatial diversity as defense-in-depth.

The diversity scheduler (see Figure 1) decides on an appropriate sequence of strategies to employ in order to best protect the client program from tampering. It employs temporal diversity by producing generations of block variant working sets, spatial diversity by compounding protocol-preserving primitives from Section 3.2.3, and semantic diversity by applying the non-protocol-preserving primitives from Section 3.2.4.

The input to the scheduler is the current working set \(w_i\), the current diversity graph \(G_i\), the performance profile of the client code, the security requirements of different parts of the code (the level attributes provided as program annotations), the set of primitives and their effect on diversity and performance, and the active set (functions currently on the client’s call stack). The output of the scheduling operation is a new working set \(w_{i+1}\), the kill set of blocks that need to be invalidated on the client side, and an updated diversity graph \(G_{i+1}\).

3.4.1 The Diversity Scheduler

The scheduler can choose between forward and rollback updates. In a forward update it applies primitive transformations to the blocks in the current working set, resulting in a new working set with more confusion but also more performance overhead. In a rollback update, one or more of the blocks in the current working set are made to revert back to previous variants.

Depending on the state that the client is currently in, not all blocks can be transformed by all primitives, nor can all blocks be reverted to previous variants. Fundamentally, the reason is that our system is designed not to replace running code, i.e. functions currently on the activation stack. We will next examine the restrictions necessary in the case of forward cycles.

3.4.2 Restrictions on Forward Update Cycles

To see the restrictions on which transformations can be applied to which blocks in case of a forward update cycle, consider the example in Figure 4 (a), where \texttt{main}_0^0 is on the stack. Since it would provide no further diversity to transform blocks not in the current generation, any block in \(w_{j}, j < i\) is forbidden. Active blocks (here, only \texttt{main}_0^0) are also forbidden since in our system we cannot replace running code. Furthermore, a block called directly by an active block is forbidden for non-protocol-preserving transformations. To see why, consider \texttt{foo}_0^0, above. If we were to change \texttt{foo}_0^0’s protocol, the call site in \texttt{main}_0^0 would have to be updated, and this is not possible since this block is currently active.
Thus, to perform a forward transformation, the scheduler computes forbidden nodes in the diversity graph, selects a permissible block \( B \) and a transformation \( T \), performs the transformation, updates the diversity graph, and sends the kill set to the client. If there is more than one possible candidate block and transformation, we choose heuristically to maximize diversity and confusion while minimizing additional overhead.

### 3.5 Linking

During execution blocks will continuously come and go in the client's block bag. Therefore, it is not possible to refer to code blocks directly, and we are forced to add a level of indirection. We thus keep an array of function pointers \( \text{funPtrArr} \) indexed by block numbers. A function call \( \text{bar}(42) \), where \( \text{bar} \) is block number 22, turns into the following code, where the function named _loadBlock(blockNo) loads a missing block over the network and fills in \( \text{funPtrArr}[\text{blockNo}] \) with the new address:

```c
float (*tmp)(int) = funPtrArr[22];
if (tmp == NULL) {tmp = _loadBlock(22);} (*tmp)(42);
```

### 4. SECURITY ANALYSIS

In this section we evaluate the security generated by our system. In Section 4.1 we first discuss the server's ability to detect tampering during interaction-events with the client. In Section 4.2 we enumerate the methods of attack available to the client and how each of these attacks can be blocked. In Section 4.3 we describe the results of attacks we implemented and executed ourselves. In Section 4.4, finally, as a measure of the difficulty of reusing analyses (the most serious attack described in Section 4.2) we measure the diversity created by our various transformations.

#### 4.1 Attack Detection

There are several ways in which the server can check that the client is playing by the rules and ways in which it can punish him if he does not. Referring back to Figure 1, we see that there are three client-server interaction events which give the server the opportunity to detect foul play.

In Figure 1 ①, the block scheduler receives a block request from the client, at which time it will verify that the block belongs to the client's current working set. If it does not, this means that the client has refused a previous block update and, as a result, is now requesting obsolete blocks belonging to a previous working set. If fake function calls have been added into the program, a request for a fake function (which has been inserted by means of the \text{opaque} primitive) will also alert the server that tampering has occurred.

In Figure 1 ②, the server initiates an update cycle and the client responds with its active set, a list of the functions on its current call stack (or call stacks, in case of a multi-threaded program). Clearly, it is in the client's best interest to lie about the active set: if he can convince us that the entire working set is active, i.e. that every function in the program is currently on the call stack, he will prevent us from transforming any blocks! The block scheduler can detect such degenerate cases by using the call graph portion of the diversity graph to verify that the active sets represent realizable call sequences.

In Figure 1 ③, the server verifies that a remote procedure call to \text{primary} is our main way of verifying that the client is executing blocks from an untampered and current working set.

We can also add application specific checks to server.c. For example, we may know that certain sequences of remote procedure calls are illegal, or we may be able to keep track of enough of the client's current local state to be able to detect calls that could never happen at the current point in the execution.

#### 4.2 Enumeration of the Attack Space

The attack tree in Figure 5 shows the attack space available to a malicious user. The root of the tree represents the attacker's goal, namely to tamper with a particular asset in the program without being detected. An asset could be a security check, code that updates a particular global variable, the integrity of a control-flow path, global data, etc. An effective attack proceeds in three steps: successfully find the asset blocks \( \mathcal{A} = \{ A_1, \ldots, A_k \} \), the set of blocks that need to be modified among the blocks in the block bag (node 1); successfully tamper with these blocks (node 2), and, finally, avoid detection by the server (node 3). Once the asset blocks have been located, the adversary is free to modify them to reach his attack goals.

Figure 5 shows four ways for the attacker to avoid the server detecting the modification to \( \mathcal{A} \), ordered from easiest to hardest to carry out. First (node 3.1), if the attacker is lucky the asset blocks are badly protected, i.e. not connected to the rest of the program through use of global variables, calls to other functions, or calls to RPCs, in any meaningful way. Then he can modify \( \mathcal{A} \) without the risk of detection.

Node 3.2 shows that if, during every update cycle, the client can convince the server that the blocks in \( \mathcal{A} \) are on the active set, then the server will never be able to update \( \mathcal{A} \). The active set must still be plausible, requiring the client to analyze the blocks, build the call graph, and report an active set that does not arouse suspicion.

Node 3.3 shows that the client can trick the server into only making trivial changes to \( \mathcal{A} \), allowing him to simply ignore updates. For this attack to work the client reports an active set that contains blocks that reference the RPCs and variables that \( \mathcal{A} \) also references. As a result, those RPCs and variables cannot be modified, which prevents the server from performing major changes to \( \mathcal{A} \).

Finally, node 3.4 shows an attack where the adversary finds ancestor blocks \( N_0, \ldots, N_{k-1} \) of a new block variant \( N_k \), extracts any new variable or RPC encodings, and patches \( \mathcal{A} \) to use these new encodings. Thus, \( \mathcal{A} \) remains tampered but conforms to any new protocols.

#### 4.2.1 Countermeasures

The primitives in Section 3.2 have been designed to counter the attacks above.

Nodes 3.1.1, 3.1, 3.2.1.1, 3.3.1.1, and 3.4.1 in Figure 5 show that block analysis is an integral part of every attack. Depending on the information needed to carry out the attack, this will involve various forms of static and/or dynamic analyses, such as disassembly, de-compilation, control-flow analysis, data-flow analysis, slicing, etc. We use the primitives, especially the protocol-preserving ones in Section 3.2.3, to make block analysis more difficult. Any reverse engineering effort will eventually succeed, but because blocks are constantly updated it is sufficient for the transformations to slow down the attacks between updates.

If the asset is contained in one or more orphan blocks (node 3.1), such as the unlikely situation where the asset is a function containing a single \text{printf("hello world")} statement, the adversary can modify it at will. In such situations the \text{opaque} primitive will allow us to connect the orphan block to the rest of the program.
by inserting bogus function calls, RPCs, and variable references.

To counter the attacks in node 3.2 and 3.3, we can again use the opaque primitive, inserting calls to non-existing functions. If the adversary reports an active set containing such a function, we know that he is cheating.

## 4.3 Empirical Tests

We implemented three attacks and evaluated our system’s ability to detect them. In all cases, the same target C program was used, a simple Tetris game with some functionality moved server-side.

An unsophisticated attacker could simply ignore block updates, continuing to execute the tampered program (node 3.3.2 in Figure 5). We simulated this attack by turning off client updates. SinceRPCs are frequent in our test program, the server reliably detected the malicious behavior shortly after the first RPC_encode update.

A malicious client may attempt to build a snapshot of the entire program, in order to analyze it off-line. This can help with attack nodes 1.1.2, 3.2.1.1, and 3.3.1.1. To simulate this attack we engineered a client that disassembles its blocks, looks for blocks which are referenced but not yet held, and requests them from the server. However, this malicious client quickly requested nonexistent blocks to which our system had inserted references; blocks a benign client would never ask for because they were protected by opaque predicates. By adding such bogus calls in just 50% of the blocks in our test program we identified the malicious client after they had successfully requested and received 16 of the 24 real blocks. When we added a call to a bogus function into every block the malicious client received only 6 of 24 real blocks (25%) before the attack was detected.

In order to prevent the server from updating blocks a client can report the entire contents of the block bag to the server as his active set. This can be seen as a crude example of the attacks in nodes 3.2.1 and 3.3.1 of Figure 5. We implemented such a client and the server was able to use the program call graph to reliably identify the malicious behavior after the first update cycle.

## 4.4 Diversity of Primitive Transformations

The most sophisticated attack in the attack tree is rooted at node 3.4. The idea is for the attacker to try to identify the history of a new block $N_k$, i.e. all its ancestors $N_0, \ldots, N_{k-1}$, and to use that information to help him reverse-engineer the transformations that were used to generate $N_k$. This, in turn, could allow him to intelligently update blocks he has previously tampered with, without having to re-analyze them from scratch.

Determining the ancestor $N_{k-1}$ of a newly updated block $N_k$ can be accomplished in one of two ways: by comparing the call graphs before and after the update (3.4.1.1) or by comparing $N_k$ against all blocks in the block bag, looking for a similar block (3.4.1.2). We claim that because our primitives generate high entropy in both the call graph and the content of the blocks, recognizing ancestry should be very difficult.

To demonstrate the call graph diversity generated by our system, we repeatedly applied the merge and split obfuscations to functions from the gzip SPEC benchmark. We compared call-graph similarity between the obfuscated and unobfuscated program using
the algorithm from Shang et al. [28]. The data series in Figure 6 (b)
shows the similarity scores after each iteration of the transforma-
tions. For comparison, the horizontal lines in the graph show simi-
arity scores of gzip to other unrelated SPEC programs; this result
demonstrates that the call graph of the modified program is often
less similar to its ancestor than to that of unrelated code.

Failing to use the call graph to identify \( N_{k-1} \), the attacker may
attempt to scan all blocks in its bag and measure their similarity to
\( N_k \). We defend against this attack by ensuring that we can gen-
erate significantly diverse implementations of a block via obfusca-
tions. To demonstrate this block diversity, we measured \( n \)-gram
similarity [25] of obfuscated blocks to their ancestors (\( N_k \) to \( N_{k-1} \)
pairings) and compared these scores with the \( n \)-gram similarity of
unrelated code (\( N_k \) to \( N_j \) pairings). To simulate the process of a
successful decompilation, we first converted block contents with
normalized source code, replacing literals and variable names with
place-holders and converting the program to high-level intermedi-
ate code. The gray bars in Figure 6 (a) show the distribution of
\( n \)-gram similarities of randomly-chosen unrelated blocks (\( N_k \) to
\( N_j \) pairings). The black bars show the distribution of \( n \)-gram simi-
larities of obfuscated blocks to their unobfuscated ancestors (\( N_k \)
to \( N_{k-1} \) pairings). Obfuscations used were interpreter and flatten.
The average \( n \)-gram similarity of unrelated blocks was .57; the av-
average similarity of a block to its ancestor was .66. Though some
obfuscated functions show high similarity to their ancestors, many
other block-to-ancestor pairings show low similarity, and would be
clearly buried in the noise of all possible block-to-block pairings,
making them impractical to find via \( n \)-gram analysis.

5. PERFORMANCE EVALUATION

With respect to performance, we evaluate the overhead of our
infrastructure (without any primitives applied), the overhead added
by individual primitives, and the delay the client experiences as the
result of an update. While it would seem interesting to measure the
"typical" or average overhead a program might suffer, this is
not possible, since it depends as much on the mix of primitives and
strategies the scheduler chooses as on the nature of the program
itself. For many applications it is more interesting, in fact, to mea-
sure worst case performance, for example the longest delay a user
might suffer as the result of an update. For all measurements we
therefore attempt to evaluate the worst case performance overhead
for a few of the SPEC benchmarks.

Client measurements were taken with our client running on a lap-
top with 4 GB of memory and an Intel 2.9 GHz Core i5 processor,
running Ubuntu Linux. Server measurements were taken with our
server running on an Amazon EC2 m1.small instance, providing
1.7 GB of memory and one EC2 compute unit.

Our infrastructure has multiple potential sources of overhead.
First of all, our client code client.c is a C program that origi-
nally might have been compiled as a single large file. Our system
instead breaks this into multiple files consisting of a single function,
each one compiled separately, and this will impede inter-procedural
optimization. Furthermore, an intelligent linker might place related
functions of client.c on the same physical page, but the client
in our system does not see the entire program at once, and cannot
make the same choices as easily. A further consequence of continu-
ous updates is that we add a level of indirection to all function calls.
To evaluate the overhead that our infrastructure adds, we measured
the change in runtime for bzip, gzip, mcf and crafty. We
found the cost as typically near a five to ten percent runtime in-
crease, though crafty was closer to a twenty percent increase.

The latency of an update is the sum of network delay, the time
to transform blocks using our primitives, compilation and linking
time (since our system is based on CIL which generates C code as
output), and dynamic load time on the client. Transformation of
blocks is usually on the order of a tenth of a second (even with our
most complex primitives). Compilation currently takes roughly a
second per function that needed to be changed and transfer across
the network takes roughly half of a second per function. The compi-
tilation time dominates the cost in our current model, but this could
be remedied by compiling multiple files concurrently on separate
computation units or by using a system such as LLVM that does
not require a final compilation stage.

6. FUTURE WORK

The latency of updates clients suffer in our system is acceptable
for some applications (say, medical records databases) but not for
those with real-time constraints, such as multiplayer online games.
We are investigating multiple ways to reduce the delays, both those
cased by network overhead and by code transformations.

Currently, we are working on supporting background generation
of working sets. The idea is simple and easily realized using our
diversity graph: (1) determine (through profiling or static analy-
sis) active sets that will be commonly occurring on the client and
generate block working sets compatible with these active sets in
the background; (2) during an update cycle, serve the client with
a pre-generated working set if one matches his current active set,
otherwise generate one on the fly.

Previous work that has suggested diversification-for-security has
typically ignored the correctness issues that follow with randomiz-
ning code, or have assumed the existence of extensive test suites
that could be run after diversification. Our situation is more difficult
than most since we generate code continuously at runtime, under
time constraints, meaning comprehensive testing after each trans-
formation is infeasible. However, we do keep track of the history
of each generated block through our diversity graph, and we hope
to use this information to integrate project unit tests in the system.

7. CONCLUSIONS

We have described a system for detecting tampering of clients
running on untrusted nodes in a distributed system. Our system
continuously updates client code, keeping it in a state of constant
flux, giving the adversary a limited time-window for analyzing and
tampering with the code. We employ protocol-preserving code
transformations which provide diversity to slow down the adver-
sary’s analysis and tampering of the code, and transformations that
are not protocol-preserving to make it harder for him to tamper with
the code without modifying its expected behavior, thereby making
it easier for the trusted server to detect the tampering.

The security afforded by the system is a function of the frequency
of code updates and the complexity and variability generated by in-
dividual code transformations. This, in turn, is related to the com-
putational overhead a particular program can afford to suffer. To
manage overhead, our diversity scheduler is designed to allow de-
tailed control over which parts of the program to transform, which
transformations to apply, and uses a rollback technique to avoid
compounding too many transformations.

We have shown that our infrastructure itself causes a perfor-
ance overhead ranging from 4% to 23% and that, for several
easy-to-implement attacks, the server reliably and quickly detects
the malicious behavior.
8. REFERENCES


