The Integrity Enforcer: Ensuring Program Data and Code Security

Kevin Hock
Miami University
Computer Science and Systems Analysis Department
hockkn@muohio.edu

Dr. Ann Sobel, Professor
Miami University
Computer Science and Systems Analysis Department
sobelae@muohio.edu

Abstract
Ensuring operational integrity of program code and data would make it virtually impossible for a program to be corrupted, either by a malicious party or through a software or hardware failure, without detection. The Integrity Enforcer is a way to provide this high level of assurance, and the Enforcer can be automatically added to any existing program, by recompiling that program's existing code using the Enforcer Builder. Further, the Enforcer is able to provide this protection, which can even be adjusted to provide an ideal balance of security, with minimal performance degradation with respect to execution time and memory usage. Several real-world examples are shown where the use of the Integrity Enforcer would have prevented large security holes, one so large that the CERT was involved and hundreds of applications needed repair. We present a detailed discussion of the criteria used to evaluate the Integrity Enforcer, along with a discussion of the inner workings of the Integrity Enforcer, which is the element of our solution that automatically adds the Enforcer to any program with a simple recompile. The construction of an Enforcer Builder is also discussed, including the issues that would have to be addressed in order to build a complete Enforcer Builder. Several limitations of the Integrity Enforcer are also discussed, along with the future work that can be done for both the Builder and the Enforcer.

Introduction and Motivation
As we move further and further into the digital age, the number of critical information systems is rapidly increasing, and such systems are playing larger and larger roles in our everyday lives. Such critical systems run the very backbones of our lives, including power networks, telecommunications networks, banking systems, nuclear facility operations, air traffic control systems, and many others. Protecting the integrity and uptime of these critical systems is essential, and there has not yet been a way to guarantee that the operating environment of any program or system has not been corrupted, either by a failure, or a malicious intrusion (by an insider or an outsider). Further, these systems, by their very nature, must almost always be exposed to the outside world (online banking, for example, exposes a bank's information systems to the world via the Internet), and any computer running in a non-isolated environment is inherently in danger of being attacked and compromised.

Further, as we become a more networked society, many programs have become decentralized, and the appearance of "mobile agents" has greatly changed the magnitude of some security issues, raising the question of how to protect such agents from a potentially malicious host, or even other malicious agents. Much like with critical systems, it is imperative that mobile agents can operate without being modified or corrupted by the host system or programs on the host system. Many attempts have been made at protecting mobile agents, but none of them have been totally satisfactory. In fact, the problem of protecting mobile agents is even more complicated than that of protecting a stand-alone system, since a stand-alone system does not have to protect itself from its operating environment. Further, a standalone machine can have its operating environment verified to be safe, while a remote machine may not afford such a luxury.

As described above, we have two unique situations, each requiring the same level of guaranteed program code and data integrity. One common approach is able to solve both of these problems; whether running on a known safe system, or the host of another system, our Integrity Enforcer can guarantee that a program has not been modified either intentionally, by a malicious user or host, or unintentionally, by a failure or glitch.

In this paper, we will first present our solution, the Integrity Enforcer, that solves the problems just described, followed with a discussion of some related work already done attempting to solve these problems. Next, we will present
the paths of discovery that led to our solution, through some code transformations and dynamic analysis of that code. A detailed presentation of the Integrity Enforcer follows, including an analysis of the cost and benefits of its use. Code fragments will be used to highlight the most important aspects of the code transformations needed by the Integrity Enforcer, and generated by the Integrity Builder, which builds the Integrity Enforcer into a program’s code. Further, the reasoning behind each implementation decision will be fully justified. Real-world applications of the Integrity Enforcer are shown through some discussions or recent security problems in well-known applications that the Integrity Enforcer could have prevented. A brief discussion of the Enforcer Builder will also discuss how the Integrity Enforcer can be easily integrated into existing software. Lastly, some limitation of the Integrity Enforcer and some final remarks about the current status of the Integrity Enforcer and the Enforcer Builder will conclude the paper.

Our Integrity Enforcer

The security solution that we present in this paper will be able to provide a guarantee of program code and data memory integrity, thus lending itself to its name, the Integrity Enforcer. If either the code segment or the data segment of a program is corrupted or altered, whether by an internal failure or a malicious intruder, the Integrity Enforcer will be able to detect that the program’s execution or memory space has been compromised. Thus the Integrity Enforcer can protect against the subtle errors that may not manifest themselves until after minutes or hours of unknown-state operations, which compounded over that time may lead to the system operating in bizarre, unexpected and possibly damaging manner. Lastly, the Enforcer Builder provides a completely automatic and completely transparent way to add the Integrity Enforcer to any program; a program only need be recompiled to take advantage of all of the benefits offered by the Integrity Enforcer. By working at the precompiler level, the Integrity Enforcer is development environment independent.

What the Integrity Enforcer cannot do is prevent an application crash; if the execution space becomes so corrupted that a program protected with the Integrity Enforcer crashes, the Enforcer will also crash. However, any sufficiently well-planned critical system will have built-in redundancy, and an outright crash will cause a rollover of service, which should result in minimal downtime. This is not viewed as a disadvantage of the Integrity Enforcer, because application crashes are likely already handled via an alternate mechanism, such as rollover to another node.

Comparison to Related Works

Many disparaging comments have been made about the solubility of the problem of ensuring code and data integrity. "Apart from using trusted hardware, few approaches exist so far to solve the problem of malicious hosts. To make the problem worse, the solubility of this problem besides the use of trusted hardware is estimated in the literature as very small or even zero." [5] This comment seems a bit premature, and the Integrity Enforcer takes steps towards proving this assertion incorrect. While the Integrity Enforcer does not solve all the security concerns for secure execution (the Integrity Enforcer does not, for example, provide security for verifying that data received from a remote host is accurate), used in tandem with other security measures, the Integrity Enforcer can significantly improve any application’s security.

There have been attempts to solve this problem of protecting programs and systems from outright failure. Two approaches that have been applied to both critical systems and mobile agents both have showed promise, but have been fundamentally weak in ways that make those solutions’ real merit questionable. Our Integrity Enforcer can be universally applied to both critical systems and mobile agents, and, depending on the level of security needs, could be combined with certain aspects of already existing solutions to provide the absolute highest levels of security.

One approach for ensuring program integrity used code obfuscations in order to make it more difficult for an intruder in the system to find specific items in memory and corrupt them. However, this solution is fundamentally flawed, although Wang was very up-front about the flaws with code obfuscations. When the code is obfuscated, it is bloated in size and its execution time is significantly affected, and obfuscations generally cripple the compiler’s optimizer because the code has become impossibly complex. Wang even admits, "Compiler optimizations are essentially ineffective when the code transformations are applied." [9] Further, random attacks changing memory contents could cause significant non-fatal problems that might cause the program to misbehave, and obfuscating the code does nothing to protect against hardware or software errors corrupting the execution space.

The second flawed solution uses a network of acceptable system states in order to detect when the program has strayed from its expected execution. However, this requires an elaborate analysis of the program, and leaving out any of the states will create an unknown situation where the program may halt when it should continue, or continue when it should halt. For extremely complex software, a complete analysis of all the program states could be prohibitively expensive, and even the author of the states approach admits this [7]. The Integrity Enforcer that we present is not bound
by program states because all information that it uses is created dynamically, so that states do not need to be identified ahead of time. Identifying valid execution states is valuable for preventing logic errors that corrupt program execution, but is not very useful from a security point of view. Elder devoted an entire thesis to the study of using a state model for fault tolerance in [3], but ultimately the practicality of such a solution diminished as the programs requiring such state analysis become larger.

Fundamentally, what these other solutions lack is protection of the program code and data memory. Ultimately, if we could guarantee that a program's code and data memory has been untouched by outside agents, we could at least ensure that the program's execution is as the programmer intended, rather than having the program work through various unplanned states or process garbage data. Further, even if we protect against program corruption caused by intruders, there is still the possibility that hardware or other software failures will cause data or program corruption. Such protections would be invaluable in any critical system, mobile agent, and valuable even in ordinary software for preventing small-scale data corruption. Detecting changes in the program or data can maintain the integrity of the process by immediately stopping the program and restoring integrity.

Other papers that provided interesting background reading during our research, but did not end up directly relating to the Integrity Enforcer, were [8], [6] and [4]. These papers provided valuable insights into the broader security issues at hand and what elements would be necessary to create a tamper-detecting program.

**Defining Criteria and Finding the Solution**

From the beginning, we knew that a good solution to protect critical systems would have four attributes: automated integration with existing code (automation), a high level of protection, minimal memory usage, and minimal performance overhead. These criteria are listed in order from most important to least important. Automation is essential; a security solution that is not almost completely automated will be rejected for use because of the time required to integrate it with the existing code base. Obviously, there must also be a high degree of protection provided, and so this is also a high priority. The last two criteria emphasize that the performance degradation as a result of the Integrity Enforcer must not prohibit timely execution of the program. Further, memory usage is considered a slightly lower priority than execution speed because memory is inexpensive, and requiring some additional memory would not be a reason for not implementing the Integrity Enforcer, except in extreme circumstances. Even in the worst case, a program's memory requirements would only double.

The complete “Integrity Solution” that we ultimately selected is the combination of three smaller solutions that together create comprehensive protection. These three pieces are code checksums, memory checksums, and active variable backups, each of which play a unique role in providing complete protection while at the same time minimizing the memory and execution overhead. Further, automation is achieved by having a set of pre-compiler code transformations (the Enforcer Builder) automatically insert the code necessary for the Integrity Enforcer to work on a given program.

The code checksums are used to ensure that blocks of code have not been altered before and after they are executed. As part of the Enforcer Builder transformation process, the program is divided into small blocks, and checksums are computed via a hash for each block and stored in the program file. During execution, when a new code block is about to run, the same hash will be executed, and the result compared to the hash checksum stored when the program was first run. If there is any difference, then corrective actions need to be taken because the program code has somehow been altered. The program could be reloaded, or halted and service rolled over to a backup node.

The second aspect of the solution protects "active" variables, those variables that can be modified within a specified code block. Inside of a code block, for each variable that is used in that block, a duplicate variable is created, and any changes to the original variable are also stored in that duplicate variable. Each time that variable is accessed, the original and the duplicate are compared, and any difference indicates that the program data memory has been altered. Only active variables are duplicated in order to prevent the backup structure from becoming unwieldy by always backing up every program variable. The active variable list for each section of code would be defined at the time of the code transformations. The definition of active variables could be changed; these changes will be discussed later with the Enforcer Builder.

The "non-active" variables, those that cannot change during the execution, are hashed at the beginning of the code block, and the checksum stored. At the end of the execution of that block, and possibly at intervals during the execution of that block (for a large block), the non-active variables will be hashed again, and any difference in the checksum indicates a compromise of the program data integrity. Like the active variable list, the non-active variable list for each block would be determined at the time when the Enforcer Builder is making the code transformations.
Simple Integrity Enforcer for C

The Integrity Enforcer was first developed statically (by hard-coding the source program, checksums, and memory backups) in order to determine its feasibility, and once the feasibility was established, a simple Enforcer Builder was written in order to demonstrate that the transformations could be made automatically. This section will describe the process used to implement each aspect of the code transformations, as well as discuss the benefits and limitations of each chosen approach. The Integrity Builder example code is shown in C++, and targets protecting C programs.

All of the hashes and variable checks are done in a separate thread, using semaphores to manage execution control and alternate execution between the running program and the Integrity Enforcer thread that verifies that the program code and data have not been modified. The benefit to using this threaded approach is that the execution is divided and can be easily controlled, and could at some later time be run on a separate CPU or even a separate machine in a more dynamic manner. Halting the execution during the Integrity Enforcer's checks also ensures that there are no synchronization problems, such as variables changing while a hash or verification is being run (such as in the incorrect summation problem from database theory).

In order to back up the active memory contents, an appropriate structure was needed that would allow dynamic sizing, since memory requirements cannot be analyzed statically at compile-time (using the new operator to allocate an array based on the size of an integer variable). Thus the structure that we have implemented allows for dynamic, unbounded growth, completely alleviating this problem. The growth factor could be changed, thus allowing a tradeoff between memory usage and execution speed; growing the memory allocated to a variable backup in larger chunks saves on execution time, but may significantly over-allocate memory.

The backup data structure implemented inside of the Integrity Enforcer is shown below. A class will be created for each type of variable used in the program, and each class object will encapsulate the backup of a variable of that class's type in the main program. The variable backup itself is stored in memory allocated to the single pointer in the class, and the size of the memory available in stored as well, so that the amount of memory allocated is always known. This is only important for array variables.

```cpp
class varint {
public:
    int datasize;
    int* data;

    varint()
    { datasize = 1;
      data = new int[datasize];
    }

    ~varint()
    { delete [] data;
    }

    int* address(int index)
    { return &data[index];
    }

    void newInfo(int newdata, int off)
    { if (off >= datasize)
      { grow(off * 2);
      }
      data[off] = newdata;
    }

private:
    void grow(int newsize)
    { int* old = data;
      data = new int[newsize];
      for (int i = 0; i < datasize; i++)
      { data[i] = old[i];
      }
      datasize = newsize;
      delete [] old;
    }
};
```

A variable verification class then wraps all of these backup variable data structures for a single variable type into one structure with all of the method calls needed to update and retrieve the information stored in the individual data structures; these methods are the interface between the main program and backup variables. A variable verification class is made for each variable type, and each variable verification class receives all of the structures and variables shown in the example code below; this code shows the backup elements for a single int data type. `typeint` will store an integer value that is assigned to the integer type; this value is used by the verification algorithm to determine what type of variable needs to be verified. `paddrint` is a pointer to the variable currently being checked in the main memory, and `curaddrint` is a pointer to the corresponding backup variable. The last few items hold the actual backup values of that data type; each `varint` object will store the duplicate information from its namesake variable in the main code. The `pint_` array structure is initialized in the class...

Variable Backup Data Structure
constructor to hold the addresses of each backup variable, which allows each variable to be referenced by number. In the example code, \( x \) would be \( \text{pint}[0] \) and \( y \) would be \( \text{pint}[1] \). This makes it very easy to identify the backup variables from the main program through integer parameters.

**Single Variable Information in Verification Class**

```cpp
class VerifyInfo {
    const int typeint;
    const int* paddrint;
    int* pcuraddrint;
    const static int numint = 2;
    varint* pint_[numint];
    varint x_;  // Address of variable \( x \)
    varint y_;  // Address of variable \( y \)

    void NewInfoint(int newval, int pos, int off) {
        pint_[pos]->newInfo(newval, off);
    }

    void OldInfoint(int addr, int pos, int off) {
        type = typeint;
        paddrint = &addr[off];
        pcuraddrint = pint_[pos]->address(off);
    }
};
```

The two methods, `NewInfoint` and `OldInfoint`, provide the interface from the main program; two example uses are shown at right. `OldInfoint` is used to set the appropriate flags for a new variable verification, and takes the address of the variable that needs to be verified, its integer position in the `pint_` array, as well as the offset position into the array (0 in the case of a non-array variable). The `OldInfoint` method sets the flag indicating what type of variable will need to be checked, sets the `paddrint` pointer to the address of the variable in main memory, and sets the `pcuraddrint` pointer to the address of the backup variable in memory. These two pointers can then be dereferenced and the actual variables compared when the Integrity Enforcer checks the variables for inconsistencies.

The `NewInfoint` method takes a new variable value, and passes it to the `NewInfo` function of the correct backup variable structure. Notice how all of the parameters needed for the `OldInfo` and `NewInfo` method calls can be read out of the source code or determined prior to the transformations while generating the verification structures. This static analysis is a critical element of the Enforcer Builder's automation.

The non-active variable hash (for variables not being modified in the current Integrity Enforcer logical code scope) and the memory hash work quite similarly. After passing in all of the necessary memory blocks, a hash is computed on the memory and the result stored. The hash is then re-run on the same memory pointers at the end of the execution, and if the checksum computed is different, then the memory has been modified unexpectedly, due to either an error or a malicious intruder. Further, no matter what memory is altered (assuming there is an alteration), whether the first checksum, the second checksum, the memory itself, or the hash algorithm, the hashes and checksums will always detect any changes.

One of the main arguments against the code obfuscations mentioned earlier is that it makes static analysis of a program after being decomplied much more difficult. With the Integrity Enforcer, there is no need for obfuscations because a static analysis would be inherently more difficult given the nature of the backups as well as the secret nature of the hash function. Unless the hash could be re-run on the code manually, and then the result changed in the code, there would be no way to corrupt the program data or program code. Changing the hash function "randomly" would also render the program inoperable. Locating the verification code and modifying it would also be something of a challenge because it would be different for each program. There really is no need for any obfuscations in the Integrity Enforcer because of its inherent structure; the added code and its dynamic nature would be confusing enough to an intruder.

**Practical Applications of the Integrity Enforcer**

While the Integrity Enforcer was designed with the intent of being able to detect direct, malicious attacks against a running program, it also can protect against other types of more casual attacks, attacks that represent just as greater on perhaps an even greater threat. This makes the Integrity Enforcer that much more versatile.

McGraw and Morriset assert that the inherent complexities of today's systems make malicious code even more of a problem, especially in C, the language that our Integrity Enforcer currently targets: "This problem [of rising complexity] is exacerbated by the use of unsafe programming languages (e.g., C or C++) that do not protect against simple kinds of attacks, such as buffer overflows."

[7] One particular attack that the Integrity Enforcer can very effectively combat is any of the buffer overflow attacks, a type of attack that is simple to execute but yet poses a significant threat. Many servers have been crashed by buffer overflows occurring in various services where executable code is appended to the end of a string sent to the server, which then overwrites memory with malicious executable code. Any program that has the Integrity
Enforcer would detect such attacks and would not execute any such code unless it was lucky enough to overwrite the currently executing code, which is unlikely due to the separation of code and data; overflowing a buffer would simply write code to a data memory area, and changes to that data memory would be caught the next time a hash was run or a variable check occurs.

The dangers of buffer overflows have been brought to the forefront of security due to a recent buffer overflow vulnerability discovered in March 2002 in the zlib lossless data-compression libraries, which are free for use. The enormity of the vulnerability was so significant that the CERT (Computer Emergency Response Team) was asked to aid in handling the exposed threat, which they described as follows: "In most circumstances, this influence will be limited to denial of service or information leakage, but it is theoretically possible for an attacker to insert arbitrary code into a running program. This code would be executed with the permissions of the vulnerable program." [2] The buffer overflow aspect was the result of a chunk of memory being freed twice, so in this case the Integrity Enforcer would be able to prevent a program against an error caused by the programmer. Clearly, the Integrity Enforcer provides an unsurpassed level of protection, sometimes even against certain logic errors created by the programmers.

How exactly would the Integrity Enforcer have been able to prevent this vulnerability in zlib from occurring? Quite simply, the Integrity Enforcer would have assured that malicious code could not have been executed, since the inserted malicious code would have been detected by one of the hashes or the active variable verification. So while the program using zlib may have terminated due to an integrity violation after the compromise was detected, at least malicious code could not have been executed--malicious code that could have resulted in data destruction or data corruption.

This vulnerability in zlib was particularly significant because zlib is known to be used in over 500 applications, and is probably used in even more than that. A press release from the authors of zlib states "The use of zlib has apparently reached pandemic proportions. :-) It is not clear that even the CERT advisory will be seen by every application author that has used zlib. You can find a partial list of zlib applications at http://www.gzip.org/zlib/apps.html and you can find vendor statements in the CERT advisory. Those represent zlib applications that we know about." [1] It is possible that thousands or millions of installed applications using zlib will continue to have this vulnerability, which is even worse now that the vulnerability has been announced all over the Internet.

The vulnerability in zlib wouldn't be particularly dangerous if all the applications that used zlib linked to it dynamically, but there are many applications that used static linking of the zlib library, meaning that any such application will have to be recompiled with a newer version of zlib linked with the executable in order to remove the vulnerability. The combined factors of the wide install base of zlib, the unknown number of applications affected, and zlib's static linking to some applications led some analysts to call into question the validity of the entire open-source movement, especially with regards to Linux, since some Linux distributions were heavy users of the zlib compression libraries. Clearly, the impact of this vulnerability was significant.

In late April and early May 2002, several vulnerabilities were also discovered in AOL's popular instant messenger software, both of which were the result of buffer overflow errors. While the exact details of the vulnerabilities or their source were withheld, nevertheless again it was a buffer overflow problem, a problem that could have been prevented by using the Integrity Enforcer. AOL's popular AIM client is used by over 100 million people, so a security vulnerability in their AIM software represents a potential public relations disaster. Further, with these two recent vulnerabilities discovered, the security of the AIM client has been completely thrown into question. Had AIM been written with the Integrity Enforcer, it would be impossible for any buffer overflow attack to cause any damage, even if the buffer overflow vulnerability was created by the programmer, as in the case of zlib.

The number of viruses in the wild has also increased significantly in the past few years. Every month brings the birth of a few additional extremely dangerous viruses, many of which work by infecting executable files with their own virus code, which then in turn can infect a computer when they are run. If the Integrity Enforcer had been built in to such a program that was modified by a virus, the code would halt at the portion that was modified since the checksum from the hash would be incorrect. Further, a virus that affects programs running in memory could also be caught and stopped. While there certainly are other virus payloads, such as wiping the BIOS or deleting the FAT / MBR on a hard drive, preventing the corruption of individual programs would be a defense against many common viruses.
The Enforcer Builder

The code transformations that make up the Integrity Enforcer are designed to be highly automated; the Enforcer Builder works on the compiler pre-processor level to generate all the code transformations, which will then be compiled into the program. A complete Enforcer Builder will identify the variables used in the program, divide the code into code blocks, identify the active and non-active variables for each block, and build all of the necessary data and hash structures to support the Integrity Enforcer. With a complete Enforcer Builder, the goal of 100% automation could be achieved, which makes adding the Integrity Enforcer to any existing software as straightforward as recompiling it. A fully implemented Enforcer Builder would have to use parsing algorithms similar to those found in a compiler, as well as several complicated data structures of its own in order to store and track all of the information needed through the code transformation process.

When building and testing the simple Integrity Enforcer for this paper ("simple" by being able to parse only a limited subset of the C language), many time tests were run in order to ensure that the overhead of the security was not becoming excessively large, especially in terms of processing time. By changing when (and if) variables were backed up, the performance of the Integrity Enforcer was drastically affected, and a block of code that ran almost instantly without any security had varying results with the Integrity Enforcer based on the settings. By backing up each variable with each assignment, even in the innermost nested loop (of a triply-nested loop), performance was decreased by a factor of about 20. By removing only a few variable backups from inner loops, which were mostly temp variables in our test program, we were able to make the performance difference almost unnoticeable. Overall, from our experience gathered to date, we have found that the overhead in execution time has been linear. Any more specific assertions are not possible given the small nature of our test programs.

The Enforcer Builder would also incorporate many options that would allow the programmer to tweak the transformations made to their program, in order to prevent performance degradation like just described. Options could include maximum code block sizes, what types of variables to back up, or the maximum memory usage permitted for the backup data structures. The memory usage would be one of the most important settings; by changing the memory available to the backup structures the programmer could easily and effectively make tradeoffs between security and the resources required to attain that security. Further, maximum security may not always be needed, so it is only appropriate to provide some control for the level of integrated security. Bear in mind that as security is increased, so is the computational overhead.

In order to demonstrate the need for memory usage settings, consider a simple loop of 100 variable assignments (initializing an array, for example). Further, consider several such loops nested inside of each other. If security is the highest concern, then clearly each assignment needs to be backed up, no matter how deeply nested it is, but if that is not necessary, then the execution time of the loop could be drastically reduced by not backing up inner variables. The implications of nested loops are shown in the table given several different scenarios of loops and backing up different levels of variables. While small numbers of variable backups will not impact program performance, nesting a few "small" loops could quickly bog down a program, especially given the indirection and the object method calls used in making the backups in the Integrity Enforcer.

Assignments in Embedded Loops

<table>
<thead>
<tr>
<th>Loop Levels</th>
<th>Backups in levels...</th>
<th>Total assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>1, 2</td>
<td>10,100</td>
</tr>
<tr>
<td>3</td>
<td>1, 2, 3</td>
<td>1,010,100</td>
</tr>
<tr>
<td>3</td>
<td>1, 3</td>
<td>1,000,100</td>
</tr>
<tr>
<td>4</td>
<td>1, 2, 3, 4</td>
<td>101,010,100</td>
</tr>
<tr>
<td>4</td>
<td>3, 4</td>
<td>101,000,000</td>
</tr>
<tr>
<td>4</td>
<td>2, 3</td>
<td>1,010,000</td>
</tr>
</tbody>
</table>

Other selection criteria could be used for determining which variables should be duplicated. For example, a statistical analysis could be done on the variable access frequency, and only the most frequently accessed variables could be backed up, assuming that any modifications to the lesser-used variables are less dangerous to the proper functioning of the program. This, of course, may be a bad assumption, and may not be appropriate for all applications. Another criteria could be variable lifespan; a global variable or instance variable would always be backed up, but a variable declared inside of a very narrow scope may be ignored, figuring that its short instantiation might be hard to notice, let alone modify its value. Yet another selection criteria could be the size of a variable, and if the memory needed is larger than a certain threshold then that variable will not be duplicated.
**Complexities of the Enforcer Builder**

Implementing a complete Enforcer Builder would be a significant challenge, and would also require some stretching of the code transformation themselves. This section will address some of the major complexities that would need to be resolved before a complete Enforcer Builder could be constructed.

The C/C++ language, or any high-level language, is admittedly complex, and since the Enforcer Builder is essentially another type of compiler, it must be able to parse almost the entire C/C++ language, including custom variable and class types. A fully featured Enforcer Builder would have to be written by someone with extensive background in compilers; in fact, beginning with the code for a compiler's parser may not be a bad place to start.

New data types and classes would create a particularly tricky situation for the Enforcer Builder, since it must be able to keep track of all new data types defined, and then recognize them later, even scattered across (potentially) multiple files! Further, all defined data types and classes must have certain methods implemented, especially the operator== and a working copy constructor, since the Enforcer Builder will need these for dealing with the objects created and comparing the original data to the duplicated data. Fortunately, default copy constructors are provided, and any class using dynamic memory should have a programmer-defined copy constructor.

Individual instances of objects would also require their own Integrity Enforcer, consistent with the object-oriented principle of data hiding, data encapsulation and data protection. In this way, each object would be responsible for its own integrity, upholding the object-oriented paradigm. Since the Integrity Enforcer thus far has been written only for C code, it would be necessary to define exactly how the Enforcer Builder could create and manage the threads associated with each object. Fortunately, from the standpoint of the Enforcer Builder, the actual modifications required in the code to add the Integrity Enforcer should remain reasonable.

The current implementation of the Enforcer Builder is designed to handle only single levels of indirection, and while double or higher levels of indirection are less common, they certainly are still easily found in programs. Keeping track of what variables are pointers of what type through "x" levels of indirection presents a significant challenge, especially when considering what dereferencing must be done in order to extract the actual value to be backed up. Discovering these layers of indirection may be difficult, especially where parameters are concerned, because a multiple-indirection pointer may be passed to a function with a single level of indirection, and any changes to the data referenced by that pointer would still have to be properly updated.

Multi-threaded applications, including any GUI-based application, could cause significant synchronization issues if not carefully planned. Each thread of execution would have to have its own verification thread, otherwise two running processes could overwrite each other's backup data in a situation similar to the Lost Update or Dirty Read problems in a database without any concurrency control. At the same time, two threads may have to share a global variable, requiring multiple variable updates, or a shared pointer. Implementing such a parser is certainly beyond the scope of this paper, although it certainly would be possible to augment the transformations with concurrency control code through timestamping or lock-based protocols.

**Limitations of the Integrity Enforcer**

The Integrity Enforcer described herein is not intended to be a complete countermeasure against all security breaches, nor can it prevent program logic errors (although it can detect logic errors that cause corrupted memory space, as in the case of the buffer overflow vulnerabilities discussed earlier); rather, the protection provided by the Integrity Enforcer guarantees that the program code and data has not been unexpectedly modified during execution. This protection is valuable, protecting against modification by any other user or program in the system, as well as hardware or software failures that may change program code or data memory unexpectedly.

Because the verifications of data and code are not always continuous, but rather done at checkpoints, it is possible that corruption of the in-process code or variables could pass undetected until the next code block begins, and, in the meantime, the program could process some amount of garbled data or incorrectly process correct data. However, this continued execution with corrupted data will only last as long as the code block, and the administrator of the system in question will at least know that the system was compromised and that the most recent set of transactions was invalid. If the program in question were a web application working with a database, commits could be done only after a checkpoint has passed, otherwise, a rollback could be done to undue the garbled transactions.

One extremely minor loophole in the protection provided by the code transformations would allow a simultaneous change of a variable and its duplicate value to pass undetected. However, the chance of this happening, whether due to malicious intent or a hardware error, is extremely slim. For an intruder to run the necessary analysis to locate such similar memory segments and then corrupt their values would simply be unreasonable, especially since the contents of the memory could be
changing very rapidly, and an intruder is not likely to look for duplicated memory segments in the first place, which was the original reason behind the variable backups.

**Future Work**

The Enforcer Builder developed in conjunction with this research is limited to demonstrating only the basic concepts of the automated code transformations, and only for a very limited feature subset of C. The parser included in the Enforcer Builder would need to be significantly overhauled, and support for classes would have to be added, which in itself would be a major undertaking; inheritance and polymorphism would drastically change the level of static analysis that would have to be done and the additional methods that may be needed for the code transformations. Even without a complete Enforcer Builder, the prototypes developed during the course of this research show proof of concept, and that the basic foundations of the Integrity Enforcer and Enforcer Builder are solid.

Other work could analyze the applicability of the Integrity Enforcer into other languages, such as Java or C#. Obviously, adding the Integrity Enforcer to interpreted code would be impossible, thus limiting its use in certain environments such as BASIC. Further, some languages, such as Pascal, which lack the powerful memory features of newer languages like C, may not be able to implement an Integrity Enforcer. It would seem reasonable to assume, however, that these older, less powerful languages are less likely to be used in security-critical applications, especially since such applications are likely to be more recently engineered programs.

Java and C# seem to be viable candidates for use with an Integrity Enforcer and Enforcer Builder. While Java and C# lack the infamous pointers of C/C++, Java and C# have other dynamic memory structures that could be used to maintain backup variables. These structures may not be as fast as using direct memory access, but they should be just as feasible. Some other major elements, such as threads and semaphores, would also have to be reevaluated as well. While certainly not without issues, the jump to Java or C# from a C++ Enforcer Builder and Integrity Enforcer would be only a minor change, rather than a complete overhaul.

**Conclusions**

Growing numbers of exposed computing environments require a way to protect applications and critical systems. Until now, no solution has been presented that guarantees that a program's code and memory space haven't been altered. With the Integrity Enforcer, we have devised a starting-point solution that can ensure untouched operation. Combined with a full Enforcer Builder, the Integrity Enforcer can be integrated with only a simple recompile, realizing the ideal of being fully automated.

While the Enforcer Builder and Integrity Enforcer were not fully implemented, the initial implementation that we created show that they are feasible, although substantial additional work will need to be done. This paper serves as a foundation for a complete implemented solution that can guarantee program execution integrity.

**References**


