Proteus: Virtualization for Diversified Tamper-Resistance

Bertrand Anckaert
Bertrand.Anckaert@UGent.be
Electronics and Information Systems
Department
Ghent University
Sint-Pietersnieuwstraat 41
9000 Ghent, Belgium

Mariusz Jakubowski
Ramarathnam Venkatesan
[mariuszj,venkie]@microsoft.com
Cryptography and Anti-Piracy Group
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA

ABSTRACT

Despite huge efforts by software providers, software protection mechanisms are still broken on a regular basis. Due to the current distribution model, an attack against one copy of the software can be reused against any copy of the software. Diversity is an important tool to overcome this problem. It allows for renewable defenses in space, by giving every user a different copy, and renewable defenses in time when combined with tailored updates. This paper studies the possibilities and limitations of using virtualization to open a new set of opportunities to make diverse copies of a piece of software and to make individual copies more tamper-resistant. The performance impact is considerable and indicates that these techniques are best avoided in performance-critical parts of the code.

Categories and Subject Descriptors

K.5.1 [**Legal Aspects Of Computing**]: Hardware/Software Protection—*copyrights;licensing*; D.2.0 [**Software Engineering**]: General—*protection mechanisms*

General Terms

Design, Legal Aspects, Security

Keywords

Copyright Protection, Diversity, Intellectual Property, Obfuscation, Tamper-Resistance, Virtualization

1. INTRODUCTION

The value contained in and protected by software is huge. According to the Business Software Alliance and the International Data Corporation [9], $31 billion worth of software was installed illegally in 2004. Digital containers are increasingly used to provide controlled access to copyrighted materials. The value of virtual characters and assets in massively multi-player online games is becoming more and more real. For example, a virtual space resort in the game Entropia Universe sold for the equivalent of $100,000[1]. It is clear that the stakes in protecting software from tampering are rising, be it to protect copyrighted software or content or to prevent players from cheating.

When the incentive to tamper is that high, we should no longer expect to build one super-strong defense that will withstand attack for an extended period of time. Even hardware solutions are not safe [3]. In the arms race between software protectors and attackers, the one that makes the last move often has the winning hand. Acknowledging this might be a first important step for software providers. If we accept that a system will be broken, the remaining defense is to minimize the impact of a successful attack. We need to make sure that an attack has only local impact, both spatial and temporal. Reducing the impact of an attack might furthermore reduce the very incentive that leads to attacks.

Diversity is a key enabler in minimizing the impact of an attack. If we can sufficiently diversify the installed base, an attack against one instance will not compromise other instances (spatially renewable defense). If we can furthermore discriminate between legitimate and illegitimate copies when updating software (temporally renewable defense), an attacker will ultimately need to revert to time-consuming manual reverse engineering to craft a specific attack for every instance and every update [1]. This can be compared to the field of cryptography, where, despite major advances, most keys are short-lived. Conversely, in the domain of software, which is not as easily formalized as arbitrary messages, we may have to rely on renewable security.

The fundamental idea behind diversity is simple: In nature, genetic diversity provides protection against an entire species being wiped out by a single virus or disease. The same idea applies to software, with respect to resistance to the exploitation of software vulnerabilities and program-based attacks [36]. Existing applications of diversity include address space layout randomization as a defense against buffer-overflow attacks and other memory-error related exploits [8] (available on Linux through PaX and Windows Vista Beta 2) and instruction-set randomization against binary code injection attacks [7].

Initially, computer security research was mainly concerned with protecting the integrity of a benign host and its data from attacks from malicious code. In recent years, a number of techniques to defend code against a malicious host have been introduced, including watermarking [17], obfus-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DRM'06, October 30, 2006, Alexandria, Virginia, USA.
Copyright 2006 ACM 1-59593-555-X/06/0010 ...$5.00.

[1] http://news.bbc.co.uk/1/hi/technology/4953620.stm

cation [19] and tamper-resistance [4]. Note that we will use the term *software security* in the context of the malicious code paradigm, while we will use the term *software protection* in the context of the malicious host paradigm. The same evolution can be observed with diversity: Most suggested applications are security-oriented, while only recently diversity has been suggested as a valid approach for software protection [1, 41]. It is worth noting that DirecTV, the largest satellite digital broadcaster, is already fighting piracy through software updates, which provide a lower cost, software-based renewable security over traditional smartcard approaches [41].

While research in diversity for software security benefits research in diversity for software protection, different rules apply. For example, most of the run-time diversification introduced for security could easily be turned off when an attacker has physical access to the program and execution environment, unless additional measures are taken. This paper is focussed on the application of software-based diversity in the context of software protection.

Diversity for software protection presupposes defending against tampering: If an attacker only wants to gain knowledge about how a certain feature was implemented, it is sufficient to inspect one instance. This concept of physical tampering with the program itself is not required in the context of software security, where some security flaws can be exploited by crafting inputs using knowledge about the specific implementation. This is due mainly to the failure of certain programming languages to capture semantics fully.

Diversity can be introduced at any point between the specification of the desired user-observable behavior of the program and the execution of the program. Diversity for fault tolerance, for example, has typically been obtained by letting different groups of programmers create different implementations of a module given only its specifications. Diversity for security is typically introduced during the loading or execution of a program. This has the advantage that the same version of the software can be distributed, which allows software providers to leverage the near-zero marginal cost of digital duplication.

In this paper, we look at automated diversification of the distributed copies. As Internet distribution of software gains adoption, we believe this to be increasingly economically viable. However, many issues remain, including more complex debugging, more complex maintenance and updating procedures, etc. These aspects are beyond the scope of this paper.

We are developing a framework, Proteus, to study the practical limitations and possibilities of diversification. Proteus targets a managed, portable and verifiable environment. This stems from our belief that protection should not come at the cost of these productivity-increasing features. Moreover, we will show that managed code can be as protected as native code running in an emulatable environment.

In essence, Proteus compiles an existing bytecode binary into diverse, more tamper-resistant copies. The applied techniques rely heavily on virtualization. Virtualization provides us with more freedom, allowing us to design our own Instruction Set Architecture (ISA) and to abandon traditional execution models. This freedom is used for two main goals:

- To increase tamper-resistance and
- To introduce diversity.

High-level concepts and techniques are presented in Section 3, while more implementation specific details are given in Section 4. Section 5 provides an evaluation of the time implications of these techniques, and Section 6 concludes. We begin with an overview of related work to situate ours within the body of existing research.

2. RELATED WORK

2.1 Diversity

Diversity for Fault Tolerance:

Research in software diversity originates in the domain of software-engineering. Software diversity is used for the purpose of fault tolerance. As such, it is an extension of the idea to use redundant hardware to detect and tolerate physical faults. The two main directions here are recovery-block software [34] and N-version programming [5]. Recovery-block software requires an acceptance test and the implementation with the highest priority to pass the test wins. N-version programming compares the outputs produced by several versions and propagates only consensus results.

Unlike our approach, fault tolerance is not intended to protect against adversarial attacks.

Diversity in the Malicious Code Model:

Diversity as a security mechanism against malicious code attacks was proposed by Cohen [15] under the term *program evolution*. A number of code transformations which guarantee semantic equivalence are discussed, including equivalent instruction sequences, code reordering, variable substitutions, inlining and outlining, adding garbage, encoding and decoding, simulation and runtime code generation.

Pu et al. [33] describe a toolkit to manage diverse implementations of software modules. The idea is to have a set of specialized versions of a module. When an attack is detected, modules are replaced by different versions.

Forrest et al. [23] focus on diversifying data and code layout, with an emphasis on protection against buffer overflow attacks. Examples include padding stack frames, randomizing the location of local and global variables and treating the stack more like a heap instead of a contiguous stack.

Address space layout randomization, which randomizes the location of the stack, heap and shared libraries, has already made its way into widely distributed operating systems, mainly to fortify systems against the exploitation of memory-related errors. This form of protection is available on Linux via PaX and in Windows Vista Beta 2.

Chew et al. [12] have added diversification of system-call mappings, while Bhatkar et al. [8] extend the idea of address space layout randomization by randomizing the addresses of datastructures internal to a process through: (i) Permutations of code and data and (ii) The insertion of random gaps between objects.

Randomized instruction set emulation [7, 27] has been proposed as a defense against binary-code-injection attacks. The idea is to scramble the program using pseudo-random numbers when it is loaded and to descramble it prior to execution.

Diversity in the Malicious Host Model:

Diversity in the context of malicious code attacks has received less attention. In this context, different rules apply: One needs to make sure that the diversification cannot easily be turned off. This can be assumed in the malicious-code model, since an attacker has no physical access to the code.

Anckaert et al. [1] discuss the application of diversity in the context of software-piracy prevention. In their scheme, different users receive different versions to assure that users cannot share an attack. Tailored updates need to assure renewable defenses in time and protect against the case where a full cracked version is distributed.

Zhou et al. [41] discuss an application under the Next Generation Network Architecture and present code transformations based upon algebraic structures compatible with 32-bit operations to diversify code.

Secured Dimensions[2] and their .NET obfuscation system appear to have similar goals: Virtualization is used to make the code more tamper-resistant and they offer vendor specific virtual machines. However, their methods and implementation details have not been publicly disclosed.

2.2 Defenses in the Malicious Host Model

Note that we will limit our discussion to software-only approaches. We will not discuss defense mechanisms that rely on hardware, such as dongles, smart cards or secure (co)processors [24]. Likewise we omit techniques where critical parts of the application are executed by a server to prevent the client from having physical access.

Software Fingerprinting:

Watermarking embeds a secret message in a cover message. Fingerprinting is a specific form of watermarking where a different message is embedded in every distributed cover message. This should make it possible to detect and trace copyright violations. As such, software fingerprinting [17, 21, 38] is closely related to software diversity, as it requires unique versions. Collusion attacks are usually hindered by diversifying other parts of the program.

Software Obfuscation:

The goal of software obfuscation [6, 19, 39] is to defend against reverse engineering and program understanding. As such, it can be used as a first layer of defense against intelligent tampering. However, this is not the primary goal of these techniques. For example, obfuscation could be used to prevent the understanding and reuse of a proprietary algorithm as well. Many transformations used for obfuscation can be easily parametrized to generate different versions of a program and thus reused in the context of software diversity.

Tamper-Resistance:

The goal of tamper-resistance [4] is to prevent modifications in the behavior of the program. Techniques include checksumming segments of the code [10, 25]. However a generic attack against such schemes has been devised [40]. Other techniques [11] hash the execution trace of a piece of code.

We refer to other overview articles for a more extended discussion of related work [18, 30, 36].

3. VIRTUALIZATION

We are developing a framework to study the applicability of virtualization for diversification and tamper-resistance.

The framework is targeted at rewriting managed MicroSoft Intermediate Language (MSIL) binaries. We have chosen to rewrite binaries in information-rich intermediate languages because they are generally assumed to be more vulnerable to attacks than native binaries and because we feel that protection should not come at the cost of decreased portability or verifiability.

3.1 Design Principles for an ISA Targeted at Software Protection

Having the freedom to design our own ISA leaves us with many choices to make. We would like to use this freedom of choice to create a set of different versions of the program with the following properties: (i) Each version in the set has a reasonable level of defense against tampering and (ii) It is hard to retarget an existing attack against one version to work against another version.

We will use some of the freedom to achieve the first goal, while the remainder will be used to create sufficiently diverse versions. This involves a tradeoff: The many choices result in a large space of semantically equivalent programs that we can generate. We can consider this entire space to allow for more diversity. Alternatively, we can consider only parts of this space which we believe to be more tamper-resistant than other parts. The design principles to steer towards tamper-resistant properties are: (i) Prevent static analysis of the program; (ii) Prevent dynamic analysis of the program; (iii) Prevent local modifications and (iv) Prevent global modifications.

The first two are closely related to the problem of obfuscation, while the latter two are more tamper-resistance oriented. However, intelligent tampering requires at least some degree of program understanding, which is typically gained from observing the static binary, observing the running executable or a combination and/or repetition of the two previous techniques.

3.2 Evolution of Design Principles for ISAs

It seems as if these design principles conflict with the trend in general purpose ISA design: The concept of the Complex Instruction Set Computer (CISC) has lost ground to Reduced Instruction Set Computer (RISC). Because of the complexity of CISC instruction sets, often accompanied by fewer restrictions on the binary, CISC binaries are usually more complex to analyze than RISC binaries. The complexity of CISC architectures can complicate analysis and allows for a number of tricks that cannot be used as easily on RISC architectures. For example, Linn et al. [28] have exploited variable instruction lengths and intermixing of code and data to try and get the disassembler out of synchronization. Self-modifying code [2, 29] is facilitated on the IA32 because explicit cache flushes are unnecessary to communicate the modifications to the CPU.

Other research, such as control-flow-graph flattening [13, 39], does not rely on architecture or CISC-specific features, and can thus be applied to RISC architectures as well.

More recently, the advent of Java bytecode and managed MSIL has promoted ISAs that are even more easily analyzed. This is due to a number of reasons. First, binaries are typically not executed directly on hardware, but need to be emulated or translated into native code before execution. To enable this, boundaries between code and data need to be known and there can be no confusion between constant data and relocatable addresses. This, of course, comes with the advantage of portability. Besides portability, design principles include support for typed memory management and verifiability. To assure verifiability, pointer arithmetic is not allowed, control flow is restricted, etc. To enable typed memory management, a lot of information needs to be communicated to the executing environment about the types of objects.

[2] http://www.secureddimensions.com

All of these design principles have led to binaries that are easy to analyze by the executing environment, but are equally easy to analyze by an attacker. This has led to the creation of decompilers for both Java (e.g., DejaVu and Mocha) and managed MSIL binaries (e.g., Reflector).

As a result of this vulnerability, a vast body of valuable research can be found in the protection of Java bytecode [6, 16, 19, 20]. Note that most of the developed techniques are theoretically applicable to CISC and RISC binaries as well, while in practice their application is complicated by the absence of rich information.

One can clearly observe a trend where the design principles of ISAs are increasingly in conflict with the design principles that would facilitate software protection.

Surprisingly, one way to counter this trend is to add an additional layer of virtualization. We will emulate our own ISA on top of the portable, verifiable, managed Common Language Runtime (CLR) environment. This idea has been mentioned in academic literature as *table interpretation* [19].

3.3 Managed Binaries can be as Protected as Native Binaries Running on an Emulatable Environment

While this can be easily proved by the following construction, it is somewhat contrary to common beliefs. Furthermore, it might facilitate the adoption of a managed environment, which is now often avoided by software providers because of the fear of losing their investment if their code can easily be reverse engineered.

Consider the following construction: (i) Write an emulator for the environment which runs on top of the CLR; (ii) Take the binary representation of a binary and add it as data to the emulator and (iii) Have the main procedure start emulation at the entry point of the original executable.

Clearly, this is now a managed, portable and verifiable binary. Furthermore, it is as protected as a native binary, as the attacker of a native binary could easily take the native binary and follow the above construction himself.

3.4 Reuse Experience from CISC era

Experience and intuition tell us that the average IA32 binary is far more complex to understand and manipulate than the average managed binary. One might wonder what the cause is for the observed complexity. Clearly, there must be some underlying reasons. We believe that three key factors are in play: (i) Variable instruction length; (ii) No clear distinction between code and data and (iii) No clear distinction between constant data and relocatable addresses.

Since instructions (opcode + operands) can have variable length (1-17 bytes), instructions need only be byte-aligned, and can be mixed with padding data or regular data on the IA32, disassemblers can easily get out of synchronization. This has been studied in detail by Linn et al. [28].

As there is no explicit separation between code and data, both can be read and written transparently and used interchangeably. This allows for self-modifying code, a feature that is renowned for being difficult to analyze [14] and has previously been exploited to confuse attackers [26, 29].

The feature that the binary representation of the code can easily be read has been used to enable self-checking mechanisms [10, 25]. The absence of restrictions on control flow has enabled techniques such as control flow flattening [13, 39] and instruction overlapping [15].

While we know of no explicit publication which specifically exploits this feature, the fact that addresses can be computed, that they cannot easily distinguished from regular data, complicates the tampering with binaries: An attacker can only make local modifications, as he does not have sufficient information to relocate the entire binary.

These observations are an important source of ways to generate a hard to analyze ISA. So far, we have provisions for variable length of instructions (see Section 4.1.2). Introducing self-modifying code in the ISA is considered future work. A technique which uses the binary representation of parts of the program for increased tamper-resistance is discussed in Section 4.1.4 and could thus be considered related to some self-checking mechanisms. Clearly, the restrictions on the control flow for managed code are not present in the custom ISA, and we consider it future work to implement techniques related to control-flow flattening and instruction overlapping.

4. PROTEUS

Proteus, son of Oceanus and Thetys, is a sea-god from Greek and Roman mythology. He can foretell the future, but will change shape to avoid having to; he will answer only to someone who is capable of capturing him. Likewise, software often “knows” things it does not want to share in an uncontrolled manner. For example, trial versions may contain the functionality to perform a given task, but a time limitation might prevent from using it for too long. In digital containers, software is often used to provide controlled access to the contents. Mobile agents may contain cryptographic keys which need to remain secret. Our approach will be to change shapes as well, to make sure that the attacker has a hard time attacking the program. We will: (i) Make the program different for different installations; (ii) Different over time through tailored updates and (iii) Different for every execution through runtime randomizations (future work). Hence, we have named our framework PROTEUS.

PROTEUS is written on top of the Phoenix RDK[3], a software optimization and analysis framework. The overall design is given in Figure 1. The frontend reads a managed MSIL binary, runs a few times over the code to determine the ISA, and produces an XML description of the virtual machine. Once the ISA has been determined, it can rewrite the original binary into the custom bytecode language.

The backend of PROTEUS reads the XML description and creates a managed `dll` for the custom virtual machine. The separation in a backend and frontend is somewhat artificial, but it does allow for a more modular design and facilitates debugging. For example, the backend can be instructed to output `C#` code instead of compiling a `dll` directly, which can then be inspected and debugged separately.

In the current implementation, parts of the original binary are retained: We will simply rewrite every function into a wrapper which calls the VM, passing the necessary arguments. This has been illustrated in Figure 2. Note that we pass all arguments in an array of `Objects`. For instance functions, this includes the `this` pointer as well. As all functions will be in a single structure, we need to pass an identification of the entry point of the function. Finally, the returned Object may needs to be cast to the return type of the original function.

[3] http://research.microsoft.com/phoenix

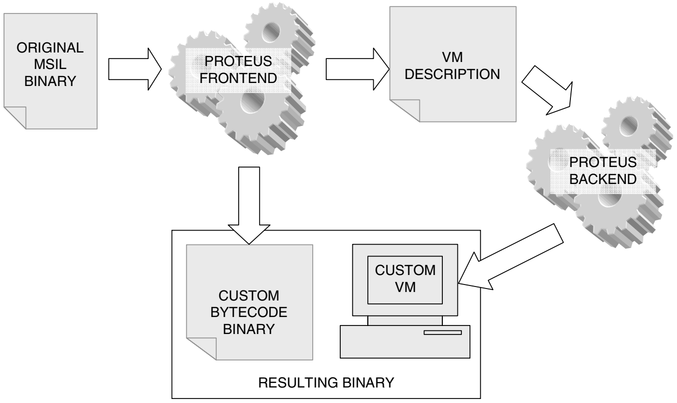
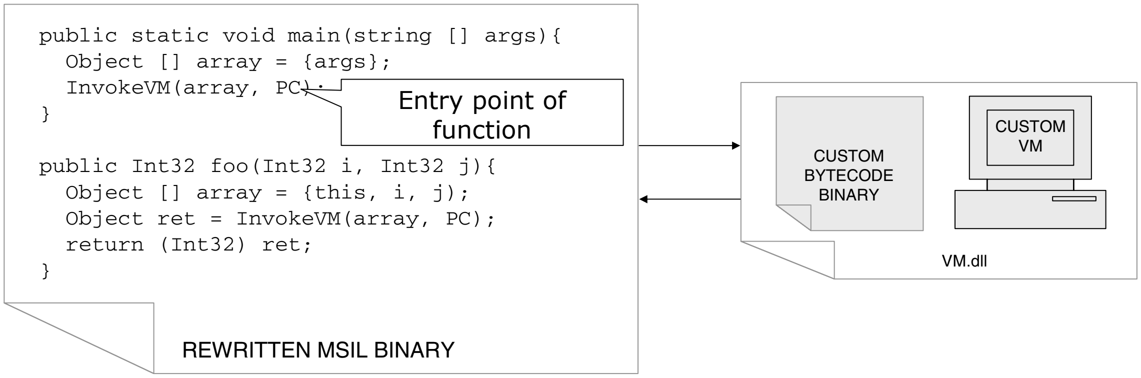


Figure 1: The Overall Design of Proteus



```
public static void main(string [] args){
  Object [] array = {args};
  InvokeVM(array, PC);
}

public Int32 foo(Int32 i, Int32 j){
  Object [] array = {this, i, j};
  Object ret = InvokeVM(array, PC);
  return (Int32) ret;
}
```

Figure 2: Converting Functions into Stubs

Rewriting the original program on a per-function basis has the advantage that we do not have to worry about things like garbage collection at this time, as datastructures are still treated as in the original program. Future work will include obfuscating, diversifying and making data more tamper-resistant as well.

We have already argued the choice for managed code. The choice between managed MSIL for the CLR and Java bytecode for the Java Runtime Environment is somewhat arbitrary. Managed MSIL is very similar to Java bytecode. As such, the results from this research can easily be transferred to that domain. Likewise, results from obfuscating Java bytecode can be applied to managed MSIL binaries. Therefore, we will focus our attention on the concepts that have not been studied in the context of Java bytecode. In particular, we will focus on the techniques that are a result of the added virtualization layer. We refer to the work of Collberg et al. [16] and the results of the Self-Protecting Mobile Agents project [6, 22] for other techniques for the protection of bytecode binaries.

4.1 Automated Protection

However much we want to protect software, we do not want it to complicate the software development cycle too much. Therefore, we introduce protection automatically at a point where human interaction is no longer required.

It is theoretically possible to generate an unmanageable number of diverse semantically equivalent programs: Consider a program with 300 instructions and choose for every instruction whether or not to prepend it with a no-op. This gives us 2^{300} different semantically equivalent programs and 2^{300} is larger than 10^{87}, the estimated number of particles in the universe.

However, uniqueness is not enough: The resulting programs should be diverse enough to complicate the mapping of information gained from one instance onto another instance. Furthermore, the resulting programs should preferably be non-trivial to break. While it is unreasonable to expect that the codomain of our diversifier will include every semantically equivalent program, we do want to maximize the codomain of the diversifier. The bigger the space is, the easier it will be to obtain internally different programs.

We need to start from an existing implementation of the semantics, rather than from the semantics itself. Through a number of parametrizable transformations we want to obtain different versions. In order to keep the diversity manageable, we have identified a number of components of the ISA that can be individualized independently. These components can be individualized in an orthogonal way, as long as the interfaces are respected. This allows for a modular design and independent development. These components are (see Figure 3):

1. Instruction semantics;
2. Instruction encoding;
3. Operand encoding;
4. Fetch cycle and
5. Program counter and program representation.

The above components are sufficient to generate a binary in the custom bytecode language; i.e., these determine the ISA. We can further diversify the virtual machine itself.

6. Diversifying the implementation of the VM.

The code in Figure 3 gives a high-level overview of the execution engine. The main internal datastructures of the VM are shown as well. The arrows indicate interface dependence. For example, DecodeOpcode expects to be able to fetch a number of bits. The diversifiable parts have been numbered in the order they will be discussed.

For each of these components, we will now discuss what the choices are and how these choices could be used to steer towards more tamper-resistant features.

4.1.1 Instruction Semantics

Freedom of choice:

To allow for the diversification of instruction semantics, we use the concept of micro-operations. An instruction in the custom bytecode language can be any sequence of a predetermined set of micro-operations. The set of micro-operations currently includes verifiable MSIL instructions and a number of additional instructions to: (i) Communicate meta-information required for proper execution and (ii) Enable additional features such as changing semantics (see Section 4.1.2). This can be compared to the concept of micro-operations (μops) in the P6 micro-architecture [31]. Each IA32 instruction is translated into a series of μops which are then executed by the pipeline. This could also be compared to the super-operators by Proebsting [32]. Super-operators are virtual machine operations automatically synthesized from smaller operations to avoid costly per-operation overheads and to reduce executable size.

We have provided stubs to emulate each of the micro-operations and these can simply be concatenated to emulate more expressive instructions in our custom bytecode language. Note that many of these emulation functions rely heavily upon reflection.

For example, consider the following MSIL instructions (addition, load argument and load constant) and their emulation stubs (simplified):

ldarg Int32:

```
 EvaluationStack.Push(
  ArgsIn.Peek(getArgSpec(insNr) );
```

ldc Int32:

```
 EvaluationStack.Push(
  getInt32Spec(insNr) );
```

add:

```
 EvaluationStack.Push(
  (Int32)EvaluationStack.Pop() +
  (Int32)EvaluationStack.Pop() );
```

Suppose that, during the instruction selection phase, we want to create a custom bytecode instruction with the following semantics: `CustomIns n i`: load the $\mathtt{n}^{th}$ argument, load the constant `i` and add these two values.

This instruction is then assigned to a `case`-statement, e.g. 1, in a large `switch`-statement. The `case`-statement is the concatenation of the different emulation stubs of the micro-operations:

```
switch(insNr){
 ...
 case 1:
   //Concatenation of stubs
   break;
 ...
}
```

Tamper-resistance:

Not knowing the semantics of an instruction will complicate program understanding, as opposed to having a manual in which semantics is specified. We can however go one step further and choose our instruction semantics to adhere to some design principles for a tamper-resistant ISA.

Conditional Execution:

To further promote merging slightly differing pieces of code, we use conditional execution. In the presence of conditional execution, instructions can be predicated by predicate registers. If the predicate register is set to false, the instruction is interpreted as a no-op, otherwise, it is emulated. The idea is to set these registers on or off along different execution paths to be able to outline slightly different pieces of code.

Limited Instruction Set:

The VM is tailored to a specific program. Therefore, we can make sure that the VM can only emulate operations that are required by that program. We can further limit the instruction set. A common way for an attacker to remove undesired functionality (e.g., a license check or decreasing the health of a wounded game character) is to overwrite that functionality with no-ops. There is little reason to include a no-op instruction in our custom ISA and not having this instruction will complicate padding out unwanted code.

Statistics furthermore show that, for example, of the integer literals from some 600 Java programs, 1.4 million lines in all, 80% are between 0-99, 95% are between 0 and 999 and 92% are powers of two or powers of two plus or minus 1 [21]. This allows us to limit the number of representable operands, again limiting the freedom of the attacker.

Another example can be found with conditional branches. Usually, there are two versions for each condition: Branch if condition is set and branch if condition is not set. Since this is redundant, we could rewrite the code so that only one version is used and not include its counterpart in the ISA. This may be useful, for example, when a license check branches conditionally depending on the validity of the serial number: It will prevent the attacker from simply flipping the branch condition.

4.1.2 Opcode Encoding

Freedom of choice:

Once instruction semantics has been determined, we need to determine an opcode encoding for those instructions. The size of all opcodes for traditional architectures is usually constant or slightly variable. For example, MSIL opcodes are typically one byte, with an escape value (0xfe) to enable two byte opcodes for less frequent instructions. The limited variability facilitates fast lookup through table interpretation. But, more generally, any prefix code (no code word is a prefix of any other code word) allows for unambiguous interpretation.

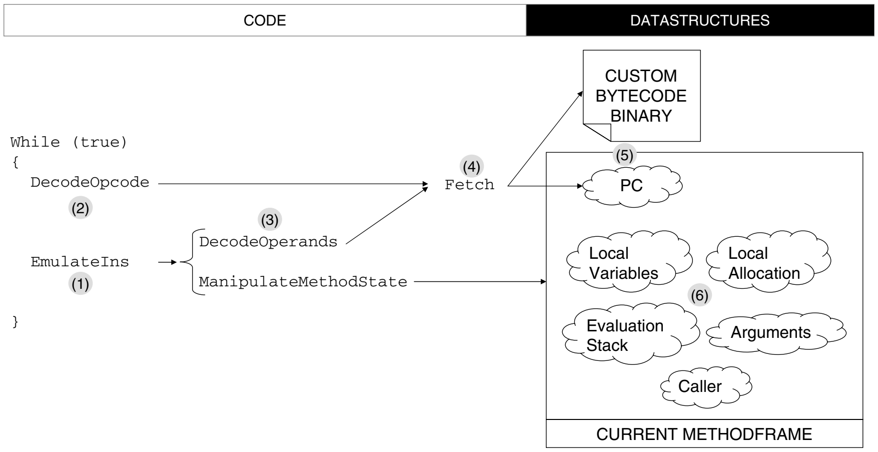


Figure 3: The Execution Model and the Interfaces

In its most general form, decoding opcodes to semantics can be done through a binary-tree traversal. Decoding starts in the root node; when a `’0’` bit is read, we move to the left child node; when a `’1’` bit is read, we move to the right child node. When a leaf node is reached, we have successfully decoded an opcode. This is illustrated in Figure 4. The leaf node contains a reference to the `case`-statement emulating the semantics of the instruction.

If we allow arbitrary opcode sizes, without illegal opcodes, the number of possible encodings for n instructions is given by the following equation:

$$\frac{\binom{2(n-1)}{n-1}}{n}n!.$$

The fraction represents the number of planar binary trees with n leaves (Catalan number), while the factorial represents the assignment of opcodes to leaves.

If we choose fixed opcode sizes with the shortest possible encoding, i.e. $\lceil log_2(n) \rceil$ bit, we might introduce illegal opcodes. In this case, the number of possible encodings is given by:

$$\binom{2^{\lceil log_2(n) \rceil}}{n}.$$

Many more possibilities would arise if we allowed illegal opcodes for other reasons than minimal fixed opcode sizes. However, this increases the size of a binary written in the custom ISA without any clear advantages. Therefore we do not consider this option.

We currently support the following modes: (i) Fixed length opcodes with table lookup; (ii) Multi-level table encoding to enable slightly variable instruction sizes (escape codes are used for longer opcodes) and (iii) Arbitrary-length opcodes with binary-tree traversal for decoding.

Tamper-resistance:

Again, not knowing the mapping from bit sequences to semantics introduces a learning curve for the attacker, as opposed to having that information in a manual. Again, there are a number of additional tricks to choose this mapping in such a way that it allows for tamper-resistance properties.

Variable Instruction Sizes:

We already know that variable instruction sizes introduce complexity in disassembling CISC binaries. When designing our own ISA, we can introduce even more variance in the length of opcodes.

Variable instruction sizes can also be used to make local modifications more complicated. It is easy to see how a larger instruction cannot simply replace a smaller instruction, because it would overwrite the next instruction. We can also make sure that smaller non-control-transfer instructions cannot replace larger instructions. This can be done by making sure that they cannot be padded out to let control flow to the next instruction.

For example, if we have 64 instructions, we could assign each of them a unique size between 64 and 127 bits. Clearly, larger instructions do not fit into the space of smaller instructions. Smaller instructions do fit in the space of larger instructions, but when control falls through to the next bit, a problem arises: There is no instruction available to pad out the remaining bits with no-ops to make sure that control flows to the next instruction. Under this scheme it is useful to make control-transfer instructions the longest, to keep an attacker from escaping to another location where he can do what he wants.

Unary Encoding:

To entangle the program further, we could try and maximize physical overlap. We want to be able to jump into the middle of another instruction, and start decoding another instruction. We could facilitate this by choosing a good encoding. For example, we could use unary encoding to encode the opcodes $(0, 01, 001, ..., 0^{63}1)$; there is a good chance that we find another instruction when we jump one bit after the beginning of an instruction. This has been illustrated in Figure 5. Four instructions have been assigned an opcode using unary encoding. We can see that if decoding is started at the second bit of the ’divide’ instruction, the ’subtract’ instruction is revealed. Likewise, looking at the last bit of the ’divide’, ’subtract’ and ’multiply’ instruction reveals the ’add’ instruction.

Non-Local Semantics:

Having a unique bytecode language for every distributed copy is clearly a major barrier for attackers. There is no documentation available on: (i) The mapping from bit pat-

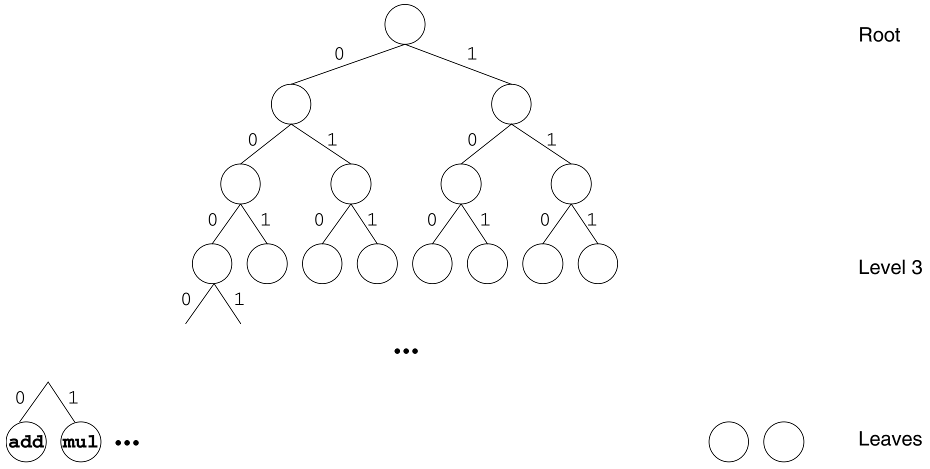


Figure 4: Prefix Code Decoding with Binary Tree

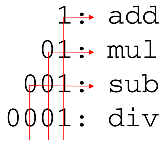


Figure 5: Unary encoding to promote physical overlap

terns to instructions; (ii) The semantics of instructions; (iii) The mapping from bit patterns to operands; (iv) The representation of data-structures; Etc.

However, this can eventually be learned through static or dynamic inspection. We can further complicate this process by making sure that a bit pattern has different meaning along different execution paths.

A binary program is just a sequence of `'1'`s and `'0'`s, which is given meaning by the processor. The meaning between bit patterns and interpretation is typically fixed by the ISA. On traditional architectures, if the opcode of a certain instruction is represented by a given bit pattern, this pattern is constant for every binary, everywhere it occurs. We want to make this variable.

A bit pattern should only be assigned meaning depending on previously executed code. The first observation we need to make if we want the interpretation to depend on previously executed code is that, depending on the (fully specified) input, we can get to a program point along different execution paths. However, we still want to have control over the interpretation of bits at a given program point. To accommodate this variability, we have chosen to make interpretation changes explicit in the ISA, rather than implicit as a side effect of some other event.

Another consideration that we need to make is that it should not be overly complex to get the executing environment in a specific interpretation state. That is to make sure that, if we can get to a program point from different execution paths in different interpretation states, we can relatively easily migrate to a single target interpretation state no matter what those different interpretation states are.

The approach we have taken is the result of the following observation: If we look back at Figure 4, it is easy to see that changing interpretation is nothing more than rearranging the decoding tree.

Taking into account the previous observations, we can only allow a limited form of diversification. To this end, we have chosen a level at which subtrees can be moved around. This choice is a trade-off between how many different interpretations are possible and how easy it is to go to a fixed interpretation from a set of possibly different interpretation states. We have chosen the third level. Assuming that the shortest opcode is 3 bit, this allows for 8! interpretation states, while any interpretation state is reachable in at most 8 micro-operations.

The micro-operations we have added to the set of MSIL micro-operations to enable this are:

- `Swap(UInt3 position1, UInt3 position2)`, which exchanges the nodes at position `position1` and `position2` and
- `Set(UInt3 label, UInt3 position)`, which exchanges the node with label `label` (wherever it may be) and the node at position `position`.

In the case of table interpretation, this is implemented as a two-level table interpretation. The first level simply refers to other tables which can be swapped.

4.1.3 Operand Encoding

As our micro-operations largely correspond to MSIL instructions, the operand types correspond largely to MSIL operand types. Micro-operation emulation stubs that use operands use function calls to ensure that opcode encoding can be diversified orthogonally to what we have previously discussed. These callbacks furthermore pass an argument `insNr` identifying the custom VM instruction from which it was called (see example Section 4.1.1). This allows us to encode operands differently for different custom VM instructions. Note that due to the concatenation of stubs, an arbitrary number of operands can follow the opcode.

Similar observations on diversifying the opcode encoding can be made as for instruction encoding.

4.1.4 Fetch Cycle

Diversifying the fetch cycle is really an artificial form of diversification. In its most simple form, the fetch cycle simply gets a number of bits from the custom bytecode binary, depending on the current Program Counter (PC). However, we will allow a number of filters to be inserted into this phase to allow for improved tamper-resistance. Basically, they will transform the actual bits in the binary to the bits that will be interpreted by the VM.

These filters will typically combine the requested bits with other information. For example, the actual requested bits may be combined with other parts of the program. This way, the program becomes more inter-dependent as changing one part of the program may impact other parts as well. Other applications include combining it with a random value derived from a secret key, or combining it with the program counter to complicate pattern matching techniques.

4.1.5 Program Representation and Program Pointer

We are very familiar with the traditional representation of the code as a linear sequence of bytes. The program counter then simply points to the next byte to execute, and control transfers typically specify the byte to continue execution at as a relative offset or an absolute address. This could be seen as representing the code as an array of bytes.

However, it is worth noting that an array is not the only datastructure that can be used to represent code. In fact, almost any datastructure will do. We could represent the code as a hash table, as a linked list, as a tree structure, etc.

So far, we have implemented representing the code as a splay tree [35]. Related research includes keeping data in a splay tree [37]. Splay trees have a number of advantages: They are self-balancing, which will allow for automatic relocation of code. Furthermore, they are nearly optimal in terms of amortized cost for arbitrary sequences. Finally, recently accessed nodes tend to be near the root of the tree, which will allow us to partially leverage temporal locality present in most executables.

Because of the self-balancing property, a piece of code could be in many different locations in memory, depending on the execution path that led to a certain code fragment. Code fragments can be moved around, as long as there is a way to refer to them for control-flow transfers, and we can retrieve them when control is transferred to them. We will use the keys of the nodes in the splay tree to make this possible: Control transfers specify the key of the node to which control needs to be transferred.

As such, it is required that targets of control flow be nodes. We cannot jump into the middle of the code contained within a node. In practice this means that we start a new node for each basic block. We deal with fall-through paths by making all control flow explicit. All control flow targets are specified as the keys of the node containing the target code. The size of the code in a node is constant. If a node is to small to contain an entire basic block, it can overflow to another node and continue execution there.

This has been illustrated in Figure 6 for the factorial function. When, for example, the function is called for the first time, the node with key ***1*** will be referenced and percolated to the root, as shown in part (2). Another thing that is worth noting in this example is that calls no longer need to specify the function signature, as this code will not be subject to verification.

We also want to note that if this technique is implemented naively: Only pointers will be moved around, and the actual code will remain at the same place on the heap. To overcome this, we can explicitly exchange the actual contents (of primitive types) of the nodes, or alternatively, we can allocate a new code buffer and copy the code buffer there, possibly with re-encryption with different garbage padding.

4.1.6 Diversifying the Implementation of the VM

4.1.6.1 Hand-Written Exchangeable Modules.

The internal implementation of, e.g., the evaluation stack is not determined by the ISA. The emulation stubs for the micro-operations rely only on an interface which supports a number of operations such as `pop` and `push`. The internal implementation of this stack datastructure can be diversified independently: It could be an array, a linked list, etc. We could provide a number of different implementations of these interfaces. Currently, we have made no efforts to diversify the VM in this way, because it requires a lot of manual coding and is not that interesting from a research perspective.

4.1.6.2 VM Generation.

Once the parameters for the above specified forms of diversification are fully specified, the PROTEUS backend will combine code snippets from various locations along with some auto-generated code to assemble a managed `C#` representation for the implementation of the custom VM. Alternatively, it can directly output a `dll`.

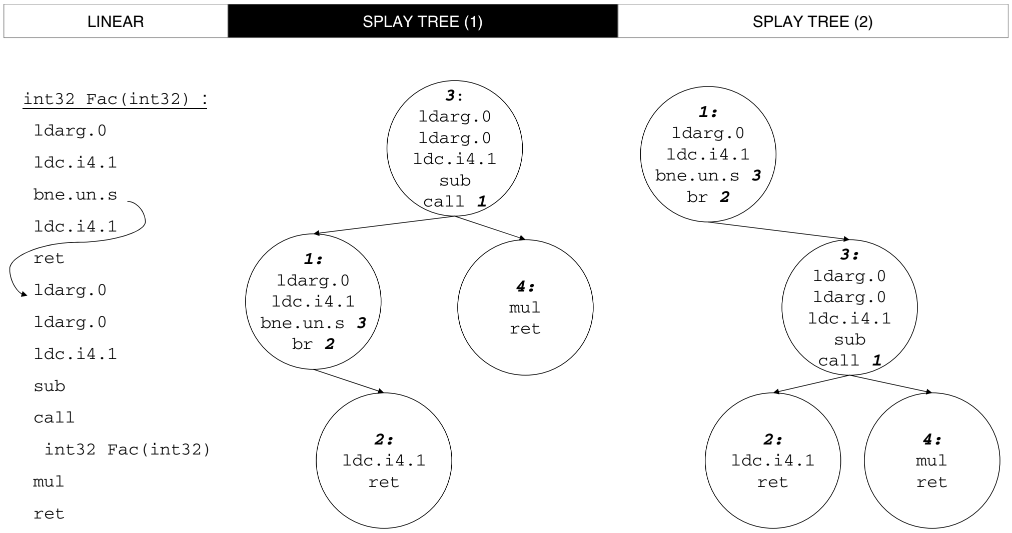
4.1.6.3 Automatic Diversification.

The implementation of the VM has been generated by combining many code snippets which have been hand-coded and are therefore limited in diversity. While not currently implemented, we suggest to diversify the resulting `dll` using randomizable versions of existing code transformations from various domains such as software optimization, software obfuscation, (non-virtualization-based approaches to) software diversification, etc.

5. EVALUATION

The slowdown for some `C#` versions of benchmarks of the Java Grande Benchmark suite is given in Table 1. The overhead of the techniques is considerable, ranging between a factor 50 and 3500. There are several reasons for this overhead. Firstly, this slowdown is a worst-case slowdown. Every function in the program has been transformed. The tool has been configured to use binary tree decoding as opposed to more optimal table interpretation to leave the instruction length completely randomizable. This results in a significant overhead for every decoded bit. Furthermore, the code is represented as a splay tree, which is clearly less optimal than an array representation. Secondly, PROTEUS is a recently developed proof-of-concept evaluation framework for research in software protection. As such, it hasn't been optimized in any way. The framework is under continuing development and we expect it to be possible to significantly reduce the overhead if these concepts were to be commercialized. Thirdly, there is an inherent overhead involved with adding an extra layer of virtualization.

There are applications in DRM and license systems where this kind of slowdown could be acceptable. These applica-



```
int32 Fac(int32) :
 ldarg.0
 ldc.i4.1
 bne.un.s
 ldc.i4.1
 ret
 ldarg.0
 ldarg.0
 ldc.i4.1
 sub
 call
  int32 Fac(int32)
 mul
 ret
```

Figure 6: Linear versus splay-tree representation for the factorial function.

Benchmark	ArithBench	CastBench	CreateBench	LoopBench	FFT	SparseMatMult
Slowdown (factor)	50.38	284.02	1557.39	253.80	1823.39	3516.39

Table 1: Slowdown as measured for C# versions of the JGrande Benchmark Suite

tions typically boil down to some computations followed by one or more Boolean checks. Computations that are too time-consuming (e.g., asymmetric cryptography with large keys) would need to be omitted from virtualization. Both run-time profiling and programmer input could be used to determine which parts are practically virtualizable.

The significant slowdown does show us that this level of transformation will typically not be acceptable for performance-critical parts of the program.

6. CONCLUSION

Virtualization opens up a wide range of possibilities for both diversity and tamper-resistance. Controlling our own execution environment gives us a lot of leverage to complicate the task of the attacker. In this paper, we have outlined the design of a framework for research in software protection based on the concept of virtualization, and we have identified a number of locations in which diversity and/or tamper-resistant features can be introduced in a largely independent way. This separation allows for modular development. We have discussed a number of techniques which intuitively lead to a more tamper-resistant ISA.

The significant overhead indicates that these kinds of techniques are to be used sparingly. Nevertheless, there are situations in which such an overhead can be tolerated.

7. REFERENCES

[1] Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Software piracy prevention through diversity. In *The 4th ACM Workshop on Digital Rights Management*, pages 63–71, 2004.

[2] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. A model for self-modifying code. In *The 8th Information Hiding Conference*, LNCS (to appear), 2006.

[3] Ross Anderson and Markus Kuhn. Tamper Resistance - a Cautionary Note. In *The 2nd Usenix Workshop on Electronic Commerce*, pages 1–11, 1996.

[4] David Aucsmith. Tamper resistant software: an implementation. In *The 1st Information Hiding Conference*, volume 1174 of *LNCS*, pages 317–333, 1996.

[5] Algirdas Avizienis and L Chen. On the implementation of n-version programming for software fault tolerance during execution. In *The IEEE Computer Software and Applications Conference*, pages 149–155, 1977.

[6] Lee Badger, Larry D'Anna, Doug Kilpatrick, Brian Matt, Andrew Reisse, and Tom Van Vleck. Self-protecting mobile agents obfuscation evaluation report, 2001.

[7] Elena Gabriela Barrantes, David Ackley, Stephanie Forrest, and Darko Stefanovi. Randomized instruction set emulation. *ACM Transactions on Information and System Security*, 8(1):3–40, 2005.

[8] Sandeep Bhatkar, Daniel DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *The 12th USENIX Security Symposium*, pages 105–120, 2003.

[9] Business Software Alliance and International Data Corporation. *Second Annual BSA and IDC Global Software Piracy Study*, 2005.

[10] Hoi Chang and Mikhail Atallah. Protecting software code by guards. In *The 1st ACM Workshop on Digital Rights Management*, volume 2320 of *LNCS*, pages 160–175, 2002.

[11] Yuqun Chen, Ramarathnam Venkatesan, Matthew Cary, Ruoming Pang, Saurabh Sinha, and Mariusz Jakubowski. Oblivious hashing: a stealthy software integrity verification primitive. In *The 5th Information Hiding Conference*, volume 2578 of *LNCS*, pages 400–414, 2002.

[12] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, 2002.

[13] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *The 4th Information Security Conference*, volume 2200 of *LNCS*, pages 144– 155, 2001.

[14] Cristina Cifuentes and John Gough. Decompilation of binary programs. *Software - Practice & Experience*, 25(7):811–829, 1995.

[15] Frederick Cohen. Operating system evolution through program evolution. *Computers and Security*, 12(6):565–584, 1993.

[16] Christian Collberg, Ginger Myles, and Andrew Huntwork. Sandmark - a tool for software protection research. *IEEE Security and Privacy*, 1(4):40–49, 2003.

[17] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *The 26th Conference on Principles of Programming Languages*, pages 311–324, 1999.

[18] Christian Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.

[19] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *The International Conference on Computer Languages*, pages 28–38, 1998.

[20] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *The 25th Conference on Principles of Programming Languages*, pages 184–196, 1998.

[21] Patrick Cousot and Radhia Cousot. An abstract interpretation-based framework for software watermarking. In *The 30th Conference on Principles of Programming Languages*, pages 311–324, 2003.

[22] Larry D'Anna, Brian Matt, Andrew Reisse, Tom Van Vleck, Steve Schwab, and Patric LeBlanc. Self-protecting mobile agents obfuscation report, 2003.

[23] Stephanie Forrest, Anil Somayaji, and David Ackley. Building diverse computer systems. In *The 6th Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[24] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[25] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *The 1st ACM Workshop on Digital Rights Management*, volume 2320 of *LNCS*, pages 141–159, 2002.

[26] Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken ichi Matsumoto. Exploiting self-modification mechanism for program protection. In *The 27th Annual International Computer Software and Applications Conference*, pages 170–181, 2003.

[27] Gaurav Kc, Angelos Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *The 10th ACM Conference on Computer and Communications Security*, pages 272–280, 2003.

[28] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *The 10th ACM Conference on Computer and Communications Security*, pages 290–299, 2003.

[29] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. Software protection through dynamic code mutation. In *The 6th International Workshop on Information Security Applications*, volume 3786 of *LNCS*, pages 194–206, 2005.

[30] Gleb Naumovich and Nasir Memon. Preventing piracy, reverse engineering, and tampering. *Computer*, 36(7):64–71, 2003.

[31] David Patterson and John Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[32] Todd Proebsting. Optimizing an ANSI C interpreter with superoperators. In *The 22nd Conference on Principles of Programming Languages*, pages 322–332, 1995.

[33] Calton Pu, Andrew Black, Crispin Cowan, and Jonathan Walpole. A specialization toolkit to increase the diversity of operating systems. In *The ICMAS Workshop on Immunity-Based Systems*, 1996.

[34] Brian Randell. System structure for software fault tolerance. *SIGPLAN Notices*, 10(6):437–449, 1975.

[35] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[36] Paul van Oorschot. Revisiting software protection. In *The 6th Conference on Information Security*, volume 2851 of *LNCS*, pages 1–13, 2003.

[37] Avinash Varadarajan and Ramarathnam Venkatesan. Limited obliviousness for data structures and efficient execution of programs. Technical report, Microsoft Research, 2006.

[38] Ramarathnam Venkatesan, Vijay Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *The 4th Information Hiding Conference*, volume 2137 of *LNCS*, pages 157–168, 2001.

[39] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *The 2nd International Conference of Dependable Systems and Networks*, pages 193–202, 2001.

[40] Glen Wurster, Paul van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. In *The 26th IEEE Symposium on Security and Privacy*, pages 127–138, 2005.

[41] Yongxin Zhou and Alec Main. Diversity via code transformations: A solution for NGNA renewable security. In *NCTA - The National Show*, 2006.