Run-Time Randomization to Mitigate Tampering*

Bertrand Anckaert[1], Mariusz Jakubowski[2], Ramarathnam Venkatesan[2], and Koen De Bosschere[1]

[1] Ghent University, Electronics and Information Systems Department
Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
{banckaer, kdb}@elis.UGent.be
[2] Microsoft Research, Cryptography and Anti-Piracy Group
One Microsoft Way, Redmond, WA 98052, USA
{mariuszj, venkie}@microsoft.com

Abstract. The problem of defending software against tampering by a malicious host is not expected to be solved soon. Rather than trying to defend against the first attack, randomization tries to minimize the impact of a successful attack. Unfortunately, widespread adoption of this technique is hampered by its incompatibility with the current software distribution model, which requires identical physical copies. The ideas presented in this paper are a compromise between distributing identical copies and unique executions by diversifying at run time, based upon additional chaff input and variable program state. This makes it harder to zoom in on a point of interest and may fool an attacker into believing that he has succeeded, while the attack will work only for a short period of time, a small number of computers, or a subset of the input space.

1 Introduction

Protecting software against attacks from the outside is a problem that has been largely solved in theory. In practice, however, vulnerabilities continue to be discovered at an astonishing rate. Buffer overflows, for example, were a solved problem as early as the 1960s, yet continue to be the most common type of security issue [19].

Due to the complexity of modern software and the increasing body of legacy code, this and other types of vulnerabilities continue to exist. Run-time randomization acknowledges this and tries to mitigate attacks at a different level: by removing predictability and consistency between different executions. Address space layout randomization (ASLR), for example, is an acknowledgment that buffer overflows and related types of attack will continue to emerge. ASLR is available for mainstream operating systems such as Linux (PaX) and Windows Vista.

* This work is partially funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT).

A. Miyaji, H. Kikuchi, and K. Rannenberg (Eds.): IWSEC 2007, LNCS 4752, pp. 153–168, 2007.

Protecting software against a malicious host is sometimes a theoretically unsolvable problem [3]. Intuitively, any protection scheme other than a physical one depends on the operation of a finite-state machine. Ultimately, given physical access, any finite-state machine can be examined and modified at will, given enough time and effort [8]. This intuition is confirmed by many examples: Users cheat in games, DRM systems are compromised, software is installed and used without the proper license, pay-TV suffers from piracy, etc.

Most defenses against malicious hosts are about delaying the first attack. The success of these techniques varies in terms of the additional time and effort required by a tamperer. However, no actively attacked protection has remained unbroken for an extended period of time. Randomization is a promising additional layer of defense: Rather than trying to postpone the first attack, it is about limiting the impact of a successful attack in space and time. However, surprisingly little research is publicly available on randomization against malicious-host attacks.

Existing proposals for randomization against malicious hosts randomize the program before distribution. Unfortunately, diverse copies are in conflict with the current software-distribution model, which requires identical copies to leverage the near-zero marginal cost of duplication. Not surprisingly, commercial implementations of this technique can be found in situations where a network connection can be assumed to distribute the copies digitally: DRM implementations for on-line music stores and digital broadcasters [27]. We suggest combining the best of both worlds by introducing diversity after distribution.

The ideas are discussed against a specific model of a tamperer's behavior: the locate-alter-test cycle (Section 2). It has been long understood implicitly that there are many similarities between tampering and debugging. In this model, we make these similarities explicit. As a result, the techniques presented to counter tampering leverage known difficulties from the domain of debugging: non-deterministic behavior (from the viewpoint of the program) and the fundamental limitations of testing (for every input and every environment). Despite originating from a specific model, the techniques increase the workload to create a fully functional patched version in a more general attack model, which assumes only that behavioral changes are made by modifying the program itself.

In this line of work, we make the run-time execution of the code unique, based upon additional chaff inputs (such as time, hardware identifiers, etc.) and variable program state, including additional fake input dependencies. The goal is twofold: (i) To make it harder for an attacker to zoom in on a point of failure and (ii) to limit the impact of a successful attack to a short period of time, a particular computer, a subset of the input space, etc. The underlying ideas are that (i) an attacker typically repeats the execution of the program with a particular input and slowly zooms in on the part where he thinks a vulnerability may occur. This becomes harder if the execution cannot be replayed at will, and (ii) if we can fool an attacker into believing that he has succeeded for a longer period of time, we can delay the feedback-loop of software tampering. These goals and high-level ideas are motivated in Section 3.

On a lower level, the technique requires a number of basic operations. Some of these operations have been dealt with extensively in academic literature. This paper contains a discussion of operations which have received less attention: (i) a concealed way to augment the user-observed input with chaff input, (ii) a criterion to select fake state and input dependencies and (iii) a diversifier to generate syntactically different, yet semantically equivalent pieces of code. These operations are discussed in Section 4-5. An experimental evaluation of the diversity that can be achieved by a practical implementation and the cost of these techniques in terms of code size and execution time is given in Section 6. Related work is the topic of Section 7, and conclusions are drawn in Section 8.

2 Low-Level Debugging Versus Tampering

Debugging and tampering are similar in many respects: many of the same techniques and tools are used in both disciplines. Debugging software is about finding and reducing the number of defects in a computer program to make it behave as the software provider intends. Likewise, tampering is about finding and reducing the number of undesired features to make it behave as the user desires.

The incentive to tamper with software thus originates from the difference between the behavior intended by the software provider and the behavior desired by the user. This difference can take on many forms; e.g.:

- Some software does not want to install without a valid license key. To some users, this is undesired behavior.
- Software may prohibit the printing of certain documents if a user does not have the right privileges. Many users find this cumbersome.
- Gamers may find it annoying that, e.g., they cannot see through walls, or that their health decreases when they get shot.
- Many users do not want their evaluation version to stop working after the evaluation period.
- Some people find it annoying that their credit card gets charged when they listen to music in a digital container, or when they watch pay TV.

Put another way, debugging is about transforming the semantics encoded in the program to the semantics intended by the software provider. Tampering is about transforming the semantics encoded in the program to the semantics desired by the user. Therefore, it should be no surprise that both disciplines are alike. Many tools, such as IDAPro and SoftICE, and many techniques, such as breakpoints and slicing, have been originally designed for debugging, but are heavily used in tampering. The main difference is that during debugging, a higher-level representation of the program is often available (source code, specification, etc.), while tampering typically starts from machine code or bytecode.

Similar to the edit-compile-test cycle of debugging, tampering is typically a cyclic process. Since tampering is usually done at a low level, the compile phase can be eliminated. Furthermore, we can split up the edit phase, leading to the following cycle:

1. **Locate the origin:** To turn the observed undesired behavior into desired behavior, a tamperer first needs to find the origin of the undesired behavior. For example, the displayed health of a gamer is only a manifestation of the internal state. Locally changing the code that displays his health will not result in the desired behavior: He needs to trace it back to where the internal representation of his health actually gets decreased.
2. **Alter the behavior:** Once the origin is determined, a tamperer needs to determine and apply a set of changes that will alter the undesired behavior into desired behavior.
3. **Test:** In this phase, the tamperer checks if the behavior of the software is as desired. If so, his work is done. Otherwise, more cycles are required.

3 Slowing Down the Locate-Alter-Test Cycle

If tampering is similar to debugging, we can argue along the same lines that making tampering harder is the opposite of making debugging easier.

One of the key concepts in making software easier to debug and maintain is modular design. Such design facilitates local changes and thus minimizes the need to verify the impact of a local change on other parts of the program. Most tamper-resistance techniques [6,7,17] have focused on doing the opposite: making the program more inter-dependent. Existing techniques are thus about **slowing down the alter phase** by requiring an understanding of a larger portion of the program and more binary changes to possibly unrelated sections of the program to effect a small change in the behavior of the program.

In this paper, we focus on slowing down the locate and test phases.

Slowing down the Locate Phase. Looking again at debugging, the first task when dealing with a bug report is to reproduce the problem. This is vital, since one cannot observe a problem and learn new facts if one cannot reproduce it. Furthermore, it is essential to find out if the problem is actually fixed. Reproduction is one of the toughest problems in debugging. One must recreate the environment and the steps that led to the problem [26].

Similarly, reproducing undesired behavior is indispensable for tampering. The manifestation of undesired behavior needs to be traced back to its origin. Typically, a tamperer repeatedly executes the application with a particular input and slowly zooms in on a part where he thinks the undesired behavior may originate.

This requires that execution can be replayed at will. We try to hamper this process by choosing between different control paths based on pseudo-random numbers, timing results, thread scheduling, etc.

In software tamper-resistance, the "bugs" are features that we want to manifest every time, so it seems illogical to make their appearance non-deterministic. We can, however, make sure that these features manifest themselves in different ways by duplicating parts of the program, diversifying them and choosing more or less randomly among the alternatives at run time. This makes it harder for a tamperer to zoom in on the vulnerable part of the program, since the semantics of the program may be constant, but the execution paths will not be identical.

Slowing down the Test Phase. Testing is also a major issue in debugging and software maintenance. It is very hard to foresee every input, every environment, every usage scenario and every combination of applications [12]. Testing can show only the presence of undesired behavior, not its absence.

The techniques discussed in this section increase the number of tests required to manifest all occurrences of the undesired behavior. The underlying idea is that the impact of a successful patch for a small subset of the input space, for a limited number of computers or for a short period of time does not pose a great threat to the software or content provider.

The time required to create a fully functional tampered version of the software is increased by letting the tamperer believe that he has succeeded, while it works only for a subset of the input space, or for a short period of time. Tamperers often work by trial and error. Using incomplete knowledge about the program, they change parts, hoping that the desired results will arise. When it is easy to evaluate whether these results have been obtained, this process can be repeated many times. If this evaluation takes longer (e.g., because it works for most of the input sets most of the time), the workload increases.

Furthermore, the credibility of the tamperer in the cracker community may decrease if he claims to have successfully patched a program, while it still behaves as intended by the software provider on other computers.

We could for example use one type of license check in 90% of the cases and another one in the remainder. This way, the tamperer may be fooled into believing that he has succeeded for a longer period of time. In this case, the tamperer has done a good job if the undesired behavior appears randomly: he can just restart the program and hope that it will work next time. However, if it is linked to certain input patterns or hardware identifiers, the usability of the tampered version is decreased significantly.

4 Tools of the Trade

The core mechanism behind the discussed techniques is illustrated in Figure 1. In its simplest incarnation, a piece of code $c \in C$ is duplicated, both copies are diversified and one of them is selected at run time more or less randomly. Note that C represents the set of syntactically correct pieces of code in whatever language it is written.

This section provides more detail on two aspects related to the input of the opaque predicates. Firstly, we present techniques to augment the user-observed input with chaff input as a source of randomness. Secondly, we discuss the usage of variable program state at a program point – e.g., as fake input dependencies. Finally, we also look into more detail on how to generate diverse copies of a piece of code, as we believe that this has not been discussed in sufficient detail in other publications. Due to the extent of the discussion, it is in a separate section (5). Other aspects, such as the creation of the opaque predicates and the rewriting of software, have been discussed elaborately elsewhere.

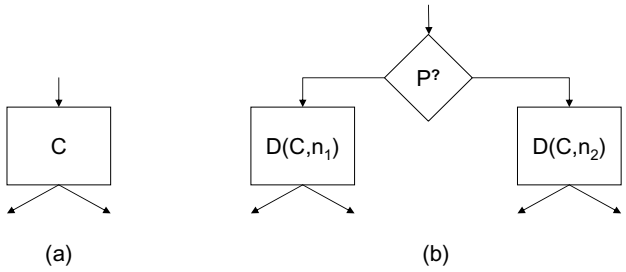

Fig. 1. The basic mechanism behind the implementation

4.1 Chaff Input

We say that run-time randomization delays the locate phase if it introduces diversity during a single "tamper session" – i.e., if the randomization takes place even on a single computer, for the same user-observed input, and for a limited period of time (one or a few days). Conversely, run-time randomization is said to delay the test phase if it requires multiple tamper sessions – i.e., the tampering itself needs to be repeated for different computers, for different user-observed inputs, and for different periods of time.

Under these specifications, chaff input is needed to delay the locate phase. This will be used as a source of pseudo-randomness, which will then serve as input to the opaque predicates. Note that chaff input is likely to stand out in command-line applications, as there is typically little difference between the user-observed and the fully specified input. However, real-life interactive applications already make use of threads, timing information, information about mouse movement, on-line content, etc., making them more suited for this technique.

We now discuss some sources of chaff input.

- **Scheduling of Threads.** In multi-threaded applications, several threads may interact with one another in a non-deterministic manner (from the viewpoint of the application). The actual scheduling depends on the operating system (and the virtual machine, if applicable), and is influenced by asynchronous events such as user interaction, other processes, thread priority, etc. Therefore, the actual scheduling is an excellent source of pseudo-randomness. If necessary, additional threads can be created to perform part of the original functionality, or to perform other software-protection tasks.
- **Return Values of System Calls.** System calls also provide a source of randomness from the viewpoint of the application. Many system (or library) calls return information that is changeable over different runs: system time, unallocated memory, network traffic, load of the machine, file system, etc. The Underhanded C Code Contest 2005 (www.brainhz.com/underhanded/) contains examples on how to obtain pseudo-randomness in a covert way. One of the entries leaves a matrix partially uninitialized, as a result of which it still contains information from a previous `stat()`-call (`stat` returns file

info, including time of last access). This type of call is common in regular programs and will thus not quickly raise suspicion.

An interesting way of randomizing the program is to change the code executed (not the behavior) based on the presence of a debugger. This way an attacker could spend much time making the program behave as desired in the debugger, only to find that it behaves differently without the debugger.

– **External Service.** Alternatively, we may require access to an external service, which provides a source of randomness. Such an external service could be a piece of trusted hardware or an on-line service.

Record/replay mechanisms and omniscient debuggers. Clearly, given a fully specified input, the behavior will be deterministic. While the fully specified input is often a superset of what the user perceives as input, a tamperer could ultimately use a perfect record/replay system [22] to make the fully specified input (including data, user interaction, communication, system calls, schedules, etc. [26]) repeatable, thus making the execution repeatable. This way he can track down and tamper with one of the copies of the origins of the undesired behavior. Alternatively, he could use an omniscient debugger [5], such as the Simics Hindsight Debugger, to back-track to the origin. Note that the general application of these techniques can be very expensive in terms of memory requirements. Therefore, a potential defense against such capabilities is to increase the amount of state necessary for the debugger to be able to trace backwards. This can be accomplished by maximizing the number of irreversible operations in the program.

In any case, there will be more origins of the undesired behavior. Unless the tamperer finds a way to automate detecting copies of that specific origin, which is undecidable in general, he must either (i) repeat this labor-intensive method for every copy of the origin, or (ii) make the choice between the different copies fixed. As a result, the workload of the tamperer increases.

Fixing the choice between different copies may be complicated as well. It may be easy automatically to find points where the different executions digress, but some of these points may be part of the original functionality of the program.

4.2 Variable Program State and Fake Input Dependencies

The internal state at a program point is itself highly variable, and therefore serves as an excellent source of input for the opaque variables. Furthermore, it is less suspicious to select different execution paths based upon the internal state.

Through profiling [20], we can easily spot tuples (p, s), for which either (i) the state s is constant at program point p for a fixed input, but variable for different inputs; or (ii) the state s is variable at program point p even for a fixed input. Note that, due to the nature of profiling, we cannot be certain that a state s is fixed; we can conclude only that a state s is fixed for the tested inputs.

Tuples for which the first property holds are then candidates for introducing fake input dependencies. As a result, execution for different inputs will differ at places where it originally overlapped. This can thus delay the test phase.

Tuples for which the second property holds are useful to delay the locate phase, because they will increase the amount of information in the static representation of the program and the number of different instructions in a trace of a particular execution. As a result, the trace will be less "foldable," by which we mean that constructing a Control-Flow Graph (CFG) from the trace will result in a larger CFG than from the original program.

Using the same argument as earlier, an attacker needs to patch at least one of the copies, and needs to patch additional ones or remove the fake dependencies on the program state. The latter may be harder than in the earlier case. These kinds of choices during execution of the program are bound to be less suspicious, since this type of choice occurs regularly during normal operation.

5 Diversity Systems

Diversity (also referred to as individualization or randomization) can be applied in a number of different ways. The most heard-of form of randomization is probably Address Space Layout Randomization (ASLR). ASLR is a specific form of randomization that requires no changes to the program itself. Instead, the operating system positions key code and data areas in a random way to make it harder to predict target addresses.

This type of defense is less viable in the malicious-host model, since we cannot rely on the environment. The only aspect that we can control is the program itself; thus, the randomization needs to be an integral part of it. Under these circumstances, there are still a number of possibilities on when and where to randomize:

- **Before distribution:** The static representation of the program is randomized before distribution.
- **During installation:** The static representation of the program is randomized when it is installed (e.g., based upon hardware, the license key, etc.)
- **Between runs:** The static representation of the program is randomized between executions, comparable to metamorphic viruses.
- **During execution:** The dynamic execution trace of the program is randomized, as discussed in this paper.

All of these types have in common that some system is needed to generate semantically equivalent, but syntactically different versions of a piece of code.

A schematic diagram of such a "diversity system" is given in Figure 2. A diversifier D takes as input a piece of code c and a set of nonces (numbers used once) $\{1, \ldots, k\}$, and produces a set of code pieces $\{D(c,1), \ldots, D(c,_k)\}$ so that $\forall i \in [1,k] : D(c,i)$ has the same functionality as c, yet $\forall (i,j) \in [1,k]^2$, $D(c,i)$ is syntactically different from $D(c,j)$.

Similar to Kerckhoffs' principle for cryptography (a cryptosystem should be secure even if everything about the system, except the key, is public knowledge), we must assume that everything about the system is public knowledge. We

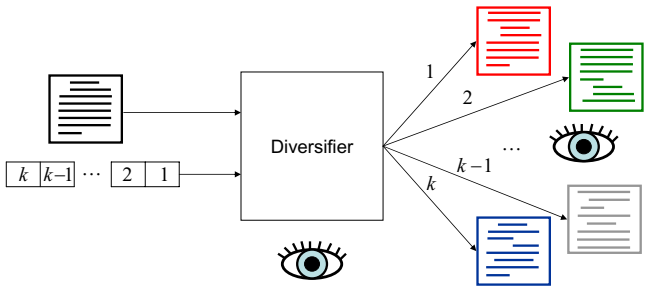

Fig. 2. Schematic of a diversity scheme

should also assume that an attacker will have access to one or more of the diversified versions. Note that this is obvious in the case of run-time randomization, as everything is embedded in the program.

5.1 Combining Diversity Systems

Rather than trying to build a single monolithic diversity system from scratch, we chose to build upon the vast body of existing research on semantics-preserving program transformations. Semantics-preserving transformations have been developed for a wide range of applications, such as refactoring, optimization for size and speed, obfuscation, watermarking, instrumentation, etc.

Combining many small transformations also makes it easier to prove (or debug) their semantics-preserving nature independently. If every transformation is semantics-preserving, the combination is guaranteed to be semantics-preserving.

The goal of combining these transformations is to enlarge the set of semantically equivalent pieces of code that can be generated by the resulting diversity system D from a piece of code $c \in C$. As usual, this is referred to as the "range" property, $ran(D, c)$.

The cardinality of the range of a diversity system can easily become very large. Consider the diversity system which chooses for every instruction in the original code whether or not to precede it by a nop-instruction. If that piece of code consists of n instructions, then the cardinality of the range is 2^n. This diversity system has a big range, yet can be easily circumvented in most applications of diversity. Therefore, the cardinality of the range is not a good indication of the quality of a diversity system.

On the other hand, a diversifier E of which the range is a superset of the range of another diversifier D ($\forall c \in C : ran(D, c) \subseteq ran(E, c)$) will typically be preferred, as this indicates that more diversity can be achieved. We will abbreviate this relation as follows: $D \subseteq E$.

Choice operation. Fortunately, given two diversity systems D and E, it is easy to create a third diversifier F for which $ran(D) \subseteq ran(F)$ and $ran(E) \subseteq ran(F)$ through the choice operation (Figure 3a): $F = D \vee E$. This corresponds to making

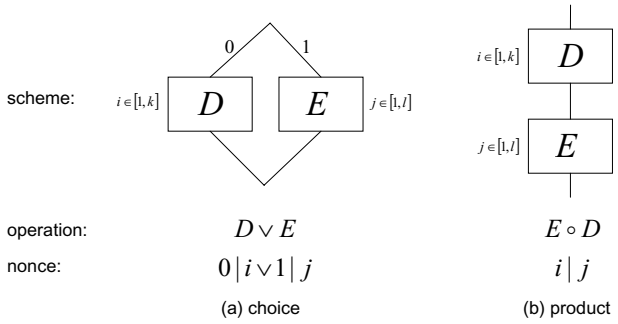

Fig. 3. Two combining operations for diversity schemes

a preliminary choice as to whether system D or E is to be used. When this is done, D or E is used as originally defined. Note that $D \vee E = E \vee D$.

Product operation. A second combining operation (Figure 3b): $F = E \circ D$, corresponds to diversifying the program with the first diversifier D and diversifying the resulting program with the second diversifier E, the nonces for D and E being chosen independently. This total operation is a diversifier whose transformations consist of all the products (in the usual sense of products of transformations) of transformations in E with transformations in D.

Logging the applied transformations. Keeping track of the applied transformations is important, since some applications may need to be able to recreate the diversified copies. For example, consider randomizing programs before distribution. When updates are needed afterwards, the software provider may need to tailor them to a specific copy. Maintaining a database of nonces requires less storage than keeping a copy of every distributed version.

This choice operation is recorded in the resulting nonce as follows: If $[1, k]$ (respectively $[1, l]$) is the range of nonces accepted by D (E respectively), then for $i \in [1, k], j \in [1, l]$, the nonce becomes $0|i \vee 1|j$. ('|' is the concatenation operator). The nonce of the product operation then becomes $i|j$.

5.2 Injective Properties of a Diversity System

Nonce-injective. An important property of a diversity system is that the resulting programs be in fact diverse. Therefore, we want different nonces to lead to different programs – i.e.,

$$\forall c \in C, \forall i, j \in [1, k], i \neq j : D(c, i) \neq D(c, j)$$

Note that the relation $=: P \times P$ denotes syntactical equivalence. We call this property *nonce-injective* – i.e., $\forall c \in C, D(c, .)$ is injective.

Typically, this will not be a problem for basic transformations (not a composition of other transformations). However, it may become an issue when many

transformations are combined using the combination operations described earlier, since the product of two nonce-injective transformations is not necessarily nonce-injective.

Injective. A transformation is injective in the traditional sense if:

$$\forall(c_1, c_2) \in C^2, \forall(i, j) \in [1, k]^2, c_1 \neq c_2 \vee i \neq j : D(c_1, i) \neq D(c_2, j)$$

Clearly, the composition of two injective transformations is injective. Furthermore, if E is an injective transformation and D is a nonce-injective transformation, then $E \circ D$ is nonce-injective. Note that if c_1 and c_2 have different semantics, c_1 cannot be syntactically equal to c_2. This definition is useful when c_1 and c_1 are two semantically equivalent, but syntactically different versions of a piece of code (e.g., after applying a diversity system).

Disjoint diversity systems. We say that two diversity systems D and E are *"disjoint"* if and only if

$$\forall c \in C, \forall i \in [1, k], \nexists j \in [1, l] : D(c, i) = E(c, j)$$

The choice of two disjoint injective transformations $(D \vee E)$ is injective.

5.3 Diversity Systems in Practice

A practical diversity system may be composed of a number of transformations: $(D_1 \vee D_2 \vee ... \vee D_n) \circ (D_1 \vee D_2 \vee ... \vee D_n) \circ (D_1 \vee D_2 \vee ... \vee D_n)$.... The probabilities that determine which transformation to choose in every iteration are assignable, and may change as the result of earlier transformations. For example, it may be useless to apply the same transformation twice. This can be recorded by setting its probability to zero for subsequent iterations once it is selected. For a more elaborate discussion on the selection of transformations with dependencies, we refer to closely related work on selecting transformations in the domain of obfuscation by Heffner and Collberg [16].

Iterated transformations. To increase the range and complexity of randomization, a tool can iterate and recombine a number of diversifying operators. Each such primitive can be quite simple – e.g., referencing variables through newly created pointers or duplicating a program statement, along with a new obfuscated predicate to choose one of the individualized copies. While such operators may be insecure when used alone, iterated application can create complexity, including emergent properties due to interaction among various transformations. This is similar to behavior found in complex systems such as cellular automata, and also helps to create confusion and diffusion, as in iterated application of rounds in block ciphers and hash functions.

Selecting transformations for the composed diversity system. The injective property and related properties discussed earlier prove to be a useful

guideline in the selection of transformations to add to the mix. For example, it is not useful to add a transformation D to a diversity system E if the range is not increased as a result ($D \subseteq E$).

Clearly, injective transformations disjoint with the already present diversity system are preferred. In practice, however, this requirement is not so stringent: Due to the large range, the chance of actually obtaining two identical code pieces after a number of transformations is small. If required, a hash can be computed of every generated code piece, and newly generated code pieces can simply be discarded if their hash matches one of the earlier ones.

Selecting nonces. In practice, it proves to be complicated to determine the range of nonces accepted by a composed diversity system. The application of one transformation will lead to more or fewer possibilities for the next transformation in a way that is hard to predict without actually applying the transformation. As the range quickly becomes unmanageable, generating all possibilities to determine the range in advance is also not practically viable. Therefore, we cannot predetermine a uniform range of nonces from which to choose in advance. Rather, every transformation will return its range once it is selected as the next transformation (and all previous transformations have been applied), after which an element from its range is selected. The nonces are thus built dynamically during the randomization, as shown in Figure 3, and can have variable lengths.

6 Evaluation

To get an idea of the achievable range of practical diversity systems, we have implemented a number of diversifying transformations in the binary rewriting framework Diablo [11]:

1. Splitting basic blocks by a two-way opaque predicate (as shown in Figure 1).
2. Inlining basic blocks with multiple incoming edges (as shown in Figure 4).
3. Inlining functions.
4. Replacing instructions by semantically equivalent instructions.
5. Reordering instructions within a basic block.
6. Inverting the condition of branches.
7. Reordering chains of basic blocks.

We have evaluated these transformatiosn on the C programs of the SPEC 2006 benchmark suite, compiled with gcc 3.2.2 and statically linked against glibc 3.2.2. The number of choices that need to be made for the transformations when applied to the entire benchmark is given in Table 1, normalized to choices between 10 options. For perlbmk, e.g., we can choose independently for $83,759$ basic blocks whether or not to split them with a two-way opaque predicate. This leads to $2^{83,759}$ possible output programs (assuming we use the same predicate every time), or a range of about $10^{25,214}$; hence the value $25,214$ in the table. The second transformation has been limited to one round, meaning that the candidates for inlining, namely (basic-block, edge) pairs, are all taken from the

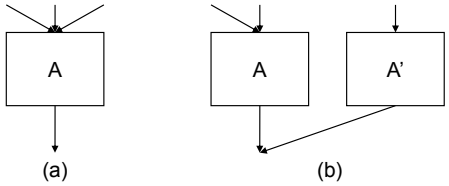

Fig. 4. Inlining basic blocks with multiple incoming edges

Table 1. Number of available choices x, normalized to 10 options. The range is 10^x.

	400.perlbench	401.bzip2	403.gcc	429.mcf	433.milc	445.gobmk	456.hmmer	458.sjeng	462.libquantum	464.h264ref	470.lbm	482.sphinx
1. split bbl	25214	6704	63948	6093	7289	16519	8355	7679	6380	12150	6146	8066
2. inline bbl	16434	3908	46098	3541	4313	10384	5037	4561	3732	6947	3585	4834
3. inline fun	2734	594	6800	508	803	2254	944	697	545	1039	513	999
4. select ins	20630	7340	51413	6603	7464	16203	8495	8251	6876	13210	7301	8228
5. schedule	14330	5679	31032	4741	5972	15462	6518	5985	5161	13617	4813	6450
6. flip branch	11516	3106	29803	2781	3184	7171	3676	3540	2888	5547	2803	3572
7. layout	76379	19319	183010	17954	21160	51065	23913	22655	18646	34457	18159	22777

original control-flow graph. If more rounds are allowed, the transformation can be reapplied endlessly for constructs such as loops (a loop can be unrolled infinitely).

The bars in the table aim intuitively to indicate the per-benchmark relation among the available choices for the different transformations.

In order to evaluate the cost of the techniques discussed in this paper, we have evaluated the impact on the code size and execution time resulting from the following setup: Transformations 1-3 are applied with a probability drawn from a Bernoulli distribution with $p = 0.05$. As a result, the transformations are applied about 5% of the times they could be applied. Both the original and copied version are then diversified by randomly applying transformations 4-7.

Over the entire benchmark suite, we notice an increase of about a quarter in code size. This is slightly higher than what one may expect at first (about 15,8% from three times 5% increase), because the candidates for transformation 2 are (basic-block, edge) pairs, which is more than just basic blocks. The same holds for transformation 3 where the candidates are (function, call-site) pairs.

The slowdown is on average 7%. This slowdown results from (i) the evaluation of the opaque predicates, (ii) additional control-flow instructions, and (iii) worse cache behavior due to less code locality and increased code size. Note that the slowdown is more variable than the code-size increase, as it depends on the execution count of the transformed code.

7 Related Work

Software Diversity. Software diversity was first used for *fault tolerance* as an extension of the idea of using redundant hardware to mitigate physical faults.

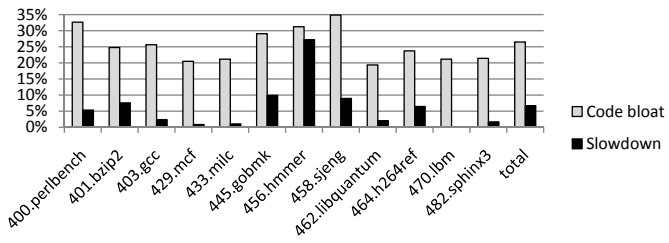

Fig. 5. Code bloat and slowdown for an exemplary transformation

The two main directions are recovery-block software [21] and N-version programming [2]. Both rely on hand-written, semantically equivalent modules. 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.

Software diversity or individualization as a security mechanism *against malicious code* attacks was proposed by Cohen [8] under the term "program evolution." Since then, numerous transformation techniques have been presented, including memory-layout randomization [13] and instruction-set randomization [4]. It has been shown that these techniques are also vulnerable to attacks [23,24].

Other research assumes the presence of diversity and studies the assignment of distinct software packages to individual systems in a network [18], or uses different versions in a framework for detection and disruption of attacks, similar to N-version programming for fault tolerance [10].

Software diversity as a protection mechanism *against a malicious host* seems to have received less attention. Existing work is focused on randomization before distribution. Anckaert et al. [1] propose to rewrite the program in a custom instruction set and to ship it with a matching virtual machine. Zhou et al. [27] present code transformations based upon algebraic structures compatible with 32-bit operations commonly present in code.

Software diversity has also been used to *hide malicious code*, such as viruses. Self-modifying viruses will typically change their binary representation before propagation. Early implementations simply encrypt the body of the virus with a different key, leaving the decryption routine vulnerable to signature-based detection. More recent viruses diversify the decryption routine as well, or contain a metamorphic engine to rewrite themselves completely (e.g., W32.simile[1]).

Tamper-Resistance. Most techniques to protect the integrity of a program are based on checksumming segments of the code [6,17]. A generic attack against such schemes has been devised for the x86 through the manipulation of processor-level segments, and for the UltraSparc through a special translation look-aside buffer load mechanism [25]. A countermeasure against this type of attack relies on self-modifying code [15]. Related techniques [7] hash the execution of a piece

[1] http://securityresponse.symantec.com/avcenter/venc/data/w32.simile.html

of code, while others have looked at the reaction mechanism in more detail. Once tampering is detected, appropriate action needs to be taken. If the manifestation of this action is too obvious, it can be easily tracked down. Delayed and controlled failures [14] are a way to make it harder to locate the reaction mechanism.

Obfuscation. Software obfuscation [3,9] aims to make programs harder to understand and has many parallels with software diversity. While the goals are different, many of the techniques developed for obfuscation can be parameterized for diversity purposes. Typically, the versions of a piece of code generated by a diversity system will be obfuscated. If the different versions are too easy to understand, it may be easy to match or find semantically equivalent code.

8 Conclusion

We modeled the tamperer's behavior starting from parallels between debugging and tampering. As such, the techniques presented to mitigate tampering leverage known difficulties from debugging: non-deterministic behavior and the fundamental limitations of testing. An experimental evaluation shows that diversity systems can generate many different semantically equivalent code sequences and that the cost of the applied techniques is acceptable for most applications.

References

1. Anckaert, B., Jakubowski, M., Venkatesan, R.: Proteus: virtualization for diversified tamper-resistance. In: The 6th ACM workshop on Digital Rights Management, pp. 47–58 (2006)
2. Avizienis, A., Chen, L.: On the implementation of N-version programming for software fault tolerance during execution. In: The 1st IEEE Computer Software and Applications Conference, pp. 149–155 (1977)
3. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001)
4. Barrantes, E., Ackley, D., Forrest, S., Stefanovi, D.: Randomized instruction set emulation. ACM Trans. on Information and System Security 8(1), 3–40 (2005)
5. Bhansali, S., Chen, W., de Jong, S., Edwards, A., Murray, R., Drinic, M., Mihocka, D., Chau, J.: Framework for instruction-level tracing and analysis of program executions. In: Virtual Execution Environments Conference (2006)
6. Chang, H., Atallah, M.: Protecting software code by guards. In: Sander, T. (ed.) DRM 2001. LNCS, vol. 2320, pp. 160–175. Springer, Heidelberg (2002)
7. Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., Jakubowski, M.: Oblivious hashing: a stealthy software integrity verification primitive. In: Petitcolas, F.A.P. (ed.) IH 2002. LNCS, vol. 2578, pp. 400–414. Springer, Heidelberg (2003)
8. Cohen, F.: Operating system evolution through program evolution. Computers and Security 12(6), 565–584 (1993)
9. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: The 25th Conference on Principles of Programming Languages, pp. 184–196 (1998)

10. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-variant systems: A secretless framework for security through diversity. In: The 15th USENIX Security Symposium, pp. 105–120 (2006)
11. De Sutter, B., De Bus, B., De Bosschere, K.: Link-time binary rewriting techniques for program compaction. ACM Trans. on Programming Languages and Systems 27(5), 882–945 (2005)
12. DiMarzio, J.F.: The Debugger's Handbook. Auerbach Publications (2007)
13. Forrest, S., Somayaji, A., Ackley, D.: Building diverse computer systems. In: The Workshop on Hot Topics in Operating Systems, pp. 67–72 (1997)
14. Gang, T., Yuqun, C., Jakubowski, M.: Delayed and controlled failures in tamper-resistant systems. In: The 8th Information Hiding Conference (2006)
15. Giffin, J., Christodorescu, M., Kruger, L.: Strengthening software self-checksumming via self-modifying code. In: The 21st Annual Computer Security Applications Conference, pp. 23–32 (2005)
16. Heffner, K., Collberg, C.: The obfuscation executive. In: Zhang, K., Zheng, Y. (eds.) ISC 2004. LNCS, vol. 3225, pp. 428–440. Springer, Heidelberg (2004)
17. Horne, B., Matheson, L., Sheehan, C., Tarjan, R.: Dynamic self-checking techniques for improved tamper resistance. In: Sander, T. (ed.) DRM 2001. LNCS, vol. 2320, pp. 141–159. Springer, Heidelberg (2002)
18. O'Donnell, A., Sethu, H.: On achieving software diversity for improved network security using distributed coloring algorithms. In: The 11th ACM conference on Computer and Communications Security, pp. 121–131 (2004)
19. Park, Y., Lee, G.: Repairing return address stack for buffer overflow protection. In: The 1st conference on Computing frontiers, pp. 335–342 (2004)
20. Pettis, K., Hansen, R.: Profile guided code positioning. In: The ACM conference on Programming Language Design and Implementation, pp. 16–27 (1990)
21. Randell, B.: System structure for software fault tolerance. SIGPLAN Notices 10(6), 437–449 (1975)
22. Ronsse, M., De Bosschere, K.: Recplay: a fully integrated practical record/replay system. ACM Transactions Computer Systems 17(2), 133–152 (1999)
23. Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: The 11th ACM conference on Computer and communications security, pp. 298–307 (2004)
24. Sovarel, A., Evans, D., Paul, N.: Where is the FEEB? The effectiveness of instruction set randomization. In: The 14th USENIX Security Symposium, pp. 145–160 (2005)
25. Wurster, G., van Oorschot, P., Somayaji, A.: A generic attack on checksumming-based software tamper resistance. In: The 26th IEEE Symposium on Security and Privacy, pp. 127–138 (2005)
26. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann, San Francisco (2005)
27. Zhou, Y., Main, A.: Diversity via code transformations: A solution for NGNA renewable security. In: NCTA - The National Show (2006)