Tamper-Tolerant Software: Modeling and Implementation

Mariusz H. Jakubowski[1], Chit Wei (Nick) Saw[1],
and Ramarathnam Venkatesan[1,2]

[1] Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA
{mariuszj,chitsaw}@microsoft.com
[2] Microsoft Research India
196/36 2nd Main, Sadashivnagar, Bangalore 560 080, India
venkie@microsoft.com

Abstract. Common software-protection systems attempt to detect malicious observation and modification of protected applications. Upon tamper detection, anti-hacking code may produce a crash or gradual failure, rendering the application unusable or troublesome. Such a response is designed to complicate attacks, but has also caused problems for developers and end users, particularly when bugs or other problems invoke anti-tampering measures accidentally. To address these issues, an alternative approach is to detect and fix malicious changes. This paper presents a scheme to transform programs into tamper-tolerant versions that use self-correcting operation as a response against attacks. Combining techniques from the fields of fault tolerance and software security, the approach transforms programs via code individualization and redundancy. We also describe security enhancements through error correction, delayed responses and checkpointing. For security analysis, we adapt a graph-based model of attacks and defenses in the context of software tamper-resistance. This helps to estimate the difficulty of breaking our scheme in practical scenarios.

1 Introduction

On modern computing systems, certain software requires protection against malicious tampering and unauthorized usage. For example, DRM (Digital Rights Management) systems attempt to prevent software piracy, as well as illegal distribution of music, video and other content. Running on open PCs, however, such security-sensitive applications are subject to observation and modification by hackers. Consequently, developers have employed *tamper-resistant software (TRS)* [5,9,18,19], which involves a variety of program obfuscation and hardening tactics to complicate hacker eavesdropping and tampering [12,32,4,29]. While no provably secure and practical methods have been deployed, various TRS heuristics extend the time and effort required to break protection.

Among the most popular protection techniques is *integrity checking*, or verifying that a program and its execution are tamper-free. Specific methods

T. Takagi and M. Mambo (Eds.): IWSEC 2009, LNCS 5824, pp. 125–139, 2009.

include computation of hashes over program code and data, along with periodic checks for mismatches between pre-computed and runtime values [10,19]. Upon detection of incorrect program code or behavior, a protection system typically responds by crashing or degrading the application (e.g., via slowdown and erratic operation) [29]. Often obfuscated, this response mechanism serves both to delay hackers and deny illegitimate usage of the application.

This typical "pessimistic" response to tampering has caused issues with application development, including testing and debugging, as well as with end-user experience. For example, application bugs sometimes manifest themselves only in tamper-protected instances of applications, forcing developers to face their own (or third-party) protection measures. Bugs in the actual protection system can be especially troublesome, particularly when interacting with protected applications. Given random application failures and erratic behavior, legitimate end users may find it difficult or impractical to file bug reports and receive support. These and other problems have contributed to general unpopularity of software protection.

A more constructive response to attacks is not to render an application unusable, but to correct the effects of tampering and allow the program to continue. The basic notion of such *tamper-tolerant software (TTS)* is appealing from the perspectives of both developers and end users, since TTS works actively to keep a program running correctly despite attacks – much like fault-tolerant systems protect against system breakdown due to malfunctioning components. Along these lines, some earlier protection schemes have used multiple copies of code to guard against tampering [9,11].

Fault tolerance [16,21,27] is a rich area that has seen much theoretical and practical work, but aims mainly to defend against "random" or unintentional failures, not against intelligent malicious attackers. However, TTS can derive from the basic concepts of fault tolerance, including redundancy, failover, and checkpoints with rollback. Likewise, error-correction methods [24] are geared mainly towards addressing noisy data transmission, but are useful in TTS as well. TTS can be considered as an adaptation and extension of fault tolerance and error correction to the intelligent-attacker scenario in software protection.

Evaluating the real-life effectiveness of software protection has been a traditionally problematic task. Most implementations in practice tend to use "ad hoc" techniques that offer only heuristic security assessments, if any. Even schemes that reduce to solving "difficult" problems can often be broken when attacks violate their idealized models or assumptions. Nonetheless, a recent line of work on graph-based modeling of tamper-resistance [15] offers some promise. In this framework, execution is modeled as a walk on program graphs, while attacks are analyzed as a "graph game" between hackers and defenders. We provide a simple adaptation of this model to our tamper-tolerance framework. As a step towards security analysis, this approach estimates the number of runtime observations and modifications required by any successful tampering attack.

The rest of this paper is structured as follows. Section 2 describes our basic TTS approach, including background on fault tolerance and software protection.

In Section 3, we present a graph-based security model for evaluating the strength of TTS. Section 4 presents a test implementation and experimental results on SPEC benchmarks. We provide a final assessment in Section 5.

2 Tamper-Tolerant Software

The essential idea of tamper-tolerant software is to detect tampering and fix its effects at runtime. This is distinct from traditional anti-tampering responses, which use techniques such as delayed crashes and graceful degradation [29,15] to block illegitimate usage and hinder attackers. Much the same effects are achieved if the program silently corrects its operation instead of failing. However, clear practical advantages come into play when applications continue to function as intended despite attacks. As with delayed responses in TRS, TTS may actually allow some tampered execution, but only temporarily.

To construct TTS schemes, our approach relies on several well known techniques from the fields of fault tolerance and software protection. Below we briefly review these techniques in the context of software security.

2.1 Building Blocks

Fault-tolerant systems typically use redundant and diversified components to resist random or non-malicious failures. With some extensions, such methods also help against intelligent attackers. We leverage the following main elements of fault tolerance:

- **Redundancy**: Duplication of system components into distinct, independently functioning units. This is a means of implementing *failover*, or switching to a fresh component if one stops working correctly. In addition, *voting* schemes compute results from multiple redundant components and select the most frequently occurring values. For example, the well known TMR/V scheme (triple modular redundancy with voting) uses three duplicate components and picks the majority answer [28].
- **Individualization**: Alteration of software code without affecting its functionality (synonymous with *diversification* [4]). Such transformations can implement the same operations in different ways, leading to potentially more robust systems. This can also make the same code appear different to adversaries, ideally forcing them to duplicate analysis efforts. Both code and data may be individualized statically (i.e., prior to runtime [1]) or dynamically (i.e., periodically during runtime [3]).
- **Checkpointing**: Upon tamper detection, rollback of execution to an earlier point. Checkpoints (i.e., summaries of program state sufficient to restart execution) are saved periodically for this purpose. This is the essential idea behind *recovery-block schemes* [26,30]. One motivation for this is attacks that alter program state without patching code, so that canceling and redoing operations can effectively fix the tampering. Alternately, a different redundant

component can redo the computation. Such checkpointing is also used by so-called "time-travel" (or "omniscient") debuggers to execute code backwards or roll execution back to some arbitrary point [8]. We may take advantage of existing checkpoint technology from such debuggers and simulators.

TTS also uses a number of techniques from the field of software protection:

- **Integrity checks**: Runtime verification of correct program operation. A traditional method is to compute checksums or hashes of code bytes, comparing at runtime against precomputed correct values [18,9]. Without reading instruction bytes, the technique of *oblivious hashing* computes and verifies hashes based on runtime variable values and control-flow transfers (e.g., by updating a hash value upon every variable assignment and branch [10,19]). Similarly, *integrity-checking expressions (ICEs)* can be used to verify integrity of execution by computing predicates over runtime state [20].
- **Delayed responses**: Separation of tamper detection and correction in space and time. This is to prevent easy identification of the TTS mechanism, mainly by disguising and hiding corrective response mechanisms [29,15]. One example technique is corrupting pointers so that future dereferencing will result in an invalid access, crashing the program [29].
- **Result correction**: Combination of several (possibly encoded) outputs from individualized copies of a code block, and output of the block's correct result despite tampering with one or more of the copies. This is a generalization of the idea of majority decoding in error correction (e.g., TMR/V, as described earlier [24]).
- **Data encoding and shuffling**: Encryption or scrambling of a program's working data, including usage of standard authentication (e.g., hashes or MACs) and error-correction codes. Variables may be continually transformed and moved in memory to prevent easy dataflow analysis and tracking [31,2].

2.2 Tamper-Tolerance Schemes

Following the core principles and terminology of fault tolerance, our basic TTS approach uses the notion of *individualized modular redundancy (IMR)*. In essence, the methodology duplicates code blocks at various granularities (e.g., basic blocks or entire functions), transforming the copies into diversified but functionally equivalent versions. We treat these code blocks as deterministic functions that map inputs to outputs and have no side effects. At runtime, the different copies may execute at various times or in parallel, producing individual intermediate outputs (all of which should be the same if no tampering occurs). Fig. 1 shows the basic conception of tamper-tolerant software.

This general IMR framework may be specialized in various ways. For concreteness, we present three specific practical schemes. We use acronyms and terminology derived from literature on fault tolerance [28,27,16,21]:

- **IMR with voting (IMR/V)**: This is IMR combined with a baseline form of result correction (similar to N-version programming [6]). From among the

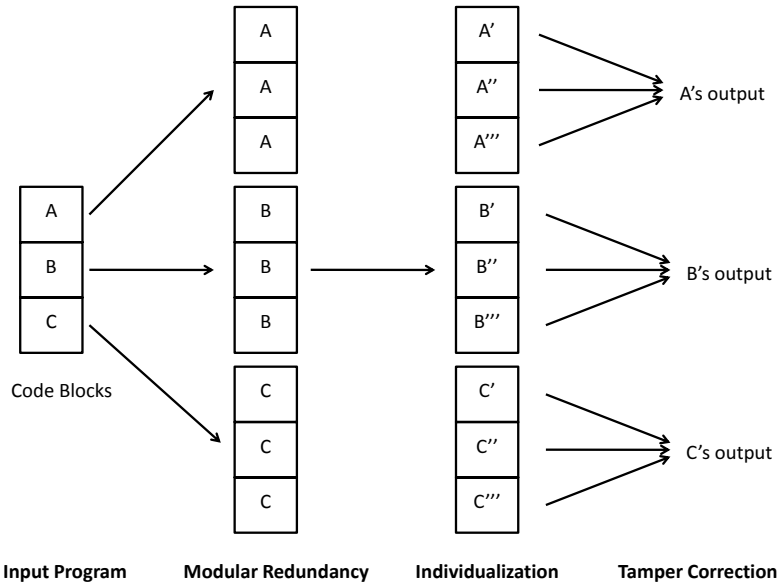

Fig. 1. High-level overview of tamper-tolerant software

results computed by individualized code blocks, a voting mechanism selects the final output. Given no tampering, the vote will be unanimous – i.e., all redundant copies generate the same output. This voting scheme can be considered as a simple means of implementing result correction.

For more secure result correction, we use the general concept of a *tamper-correcting transform* that accepts a number of encrypted (or scrambled) and potentially corrupted intermediate results. Individualized copies of a code block compute such results; the transform then decrypts (or unscrambles) these values and uses error correction on them to produce one final answer. As described above, the simplest form of this transform is equivalent to error correction via majority voting – i.e., the intermediate results are in plaintext, and the most commonly appearing value is selected as the final output. In general, however, the decryption (or unscrambling) and error correction (or voting) may be coalesced into a single transform that performs these distinct operations implicitly.

– **IMR with detection and correction (IMR/DC):** In this scheme, the protection system resorts to redundant execution only if tampering is detected. Using the integrity-verification techniques described earlier, the system checks the execution of each code block for correctness (e.g., via verifying code-byte checksums or oblivious hashes of execution). Upon tamper detection, the system selects and executes an individualized copy of the block, also verifying its runtime integrity. The system may simply call another individualized version of the block, or overwrite the tampered code with new code from a repository of possibly encrypted redundant blocks. This process

repeats until some copy of the block executes successfully without tampering (or until no more individualized blocks are available). Checkpointing may be used to roll back execution and provide correct program state before a copied block executes. If side effects are part of block execution, checkpointing with rollback may be necessary for proper program operation.

IMR/DC has some parallels with the notion of *recovery blocks* in standard fault tolerance [26]. The basic idea is to detect and recover from failures by rolling back to a "last known good" execution state. Fault detection may be accomplished via algorithm-specific checks [30], which are analogous to runtime tamper detection via techniques like integrity-checking expressions [20]. Also related is recent work on the Rx system, which survives software bugs via checkpoint-based rollback and re-execution in a modified environment [25]. However, these systems are geared against random or non-malicious faults, while IMR/DC and TTS in general aim to resist intelligent attackers.

- **IMR with randomized execution (IMR/RE):** This method selects individualized code blocks randomly for execution. For example, given three redundant, functionally equivalent code blocks A, B, and C, the system chooses and executes one with some probability (e.g., 1/3 for each of A, B, and C). If an attacker tampers with only A, execution will still be correct with probability 2/3, since B and C may be picked. Controlled by opaque predicates or other possibly obfuscated means [14,13], block selection may vary during runtime and between runs of a program. Such an approach cannot be used standalone to ensure tamper tolerance, but may be combined with other methods to enhance security of tolerance. In particular, combining IMR/RE with checkpointing (or rollback to an earlier point upon tamper detection) can undo tampering.

In a typical implementation, various other software-protection techniques would be incorporated into a tamper-tolerance system. These include data transformations [31,2], delayed responses [29], as well as other applicable tamper-resistance and obfuscation techniques. The main goal is to strengthen the above basic schemes, as well as to satisfy "engineering assumptions" required by the security model we adapt [15], as described later.

2.3 Tamper-Tolerance Algorithms

Using the above concepts, we present a general algorithm outline to implement TTS in practice. Given a program P to be protected, along with optional user-specified security and maintenance parameters, the algorithm transforms P into a new tamper-tolerant version P'. P is typically a single application, library module or driver running in either user or kernel mode. The following are general basic steps of the transformation process to protect P:

1. Break up P into a number of blocks, which may range in granularity from individual instructions to the entire program P. In practice, subdivision

into standard basic blocks (i.e., code sections with no incoming or outgoing internal branches) or functions is likely to suffice.
2. Duplicate, individualize and rearrange the blocks within the program, relying on user-specified parameters to determine number(s) of copies and specifics of individualization (e.g., types and degrees of diversifying transformations).
3. Inject appropriate code to implement IMR/V, IMR/DC, IMR/RE, or a combination of two or three thereof, as specified by user parameters. Such code may include opaque predicates for block selection.
4. Optionally inject code to manage result correction, data transformations, delayed responses, checkpointing, or a combination of two or more thereof. Also optional is injection of any additional obfuscation measures.
5. For enhanced security, optionally iterate the above steps two or more times, so that the tamper-tolerance measures are themselves protected by one or more layers of tamper tolerance. This is important for leveraging the basic IMR approach.

3 Security Modeling

This section presents a security model to evaluate the effectiveness of TTS schemes. Our main goal is a method to estimate the complexity of breaking TTS in practice (e.g., in terms of the number of observations and modifications required by any successful attacker). Different applications require different levels of resistance; e.g., a few weeks without cracks might suffice for copy protection on some games, while even a few hours may be enough for quickly refreshed Web-based code. Although "unbreakable" software protection remains an open problem, this is not necessary in many, if not most, business contexts.

Theoretical treatment of obfuscation has yielded negative general results [7,17]. This indicates that no tool can protect all possible programs, but certain limited classes of functionality are amenable to obfuscation [22,33]. However, this is in a very strict "all or nothing" model, where any non-negligible information extracted efficiently from obfuscated code is considered a break. On the other hand, our modeling approach is geared mainly towards estimating the time and effort required to defeat protection.

To assess security of TTS, we adapt a *graph-game* tamper-resistance framework that treats a program as a graph and execution as a walk on the graph [15]. The program is protected by integrity checks contained in some nodes, and the attacker's goal is to isolate all such nodes. Although originally intended for modeling tamper-resistant software, this approach can be adapted naturally to TTS. We provide an informal but self-contained summary of this framework, along with a description of our changes to model TTS.

3.1 Graph-Game Model

The main elements of the graph-game model [15] are as follows:

- **Program**: The program P to be protected is viewed as a graph G, such as a control-flow graph (CFG). This CFG should be "semi-random," which is achieved via transformations that add nodes and edges.
- **Execution**: This is modeled as a "semi-random" walk on the graph G. Most program CFGs result in walks that are not particularly random, given typical patterns of control flow. The model relies on various transformations to randomize the runtime walk.
- **Integrity checks**: Unknown to the attacker, some nodes in the CFG execute probabilistic checks to verify correct program operation at a particular time and place. In practice, this abstract notion of an integrity check may be implemented in various ways, such as code-byte checksums [18,9], oblivious hashing [10,19], and integrity-checking expressions [20]
- **Tamper responses**: When certain sets of checks fail, a tamper response is initiated. This is typically a delayed crash, or a program failure designed to deflect attention away from its causes (i.e., tampering with nodes).

With these elements, the graph game involves an attacker able to run the protected program, tamper with nodes, and observe program behavior. The program's response to tampering (i.e., time and place of failure) reveals information about the placement of integrity checks. Once the attacker identifies a node containing such a check, he may disable it. His goal is to identify all nodes responsible for integrity verification, which he can then eliminate to unprotect the program.

The analysis in [15] essentially shows that in order to achieve success, the attacker must perform a large number of program observations and node modifications. This number may be quadratic (or higher) in terms of the workload required to protect the program. In general, this is our main goal – i.e., the attacker should expend an order of magnitude more effort than the protector.

3.2 TTS Modeling

The graph-game framework is almost directly applicable to TTS. Our only main change is to replace the notion of delayed crashes with *delayed fixes*. More specifically, instead of responding to the attacker's tampering by eventually crashing, the program will correct its operation and continue running. When the attacker tampers with graph nodes, he may observe altered program behavior for some time, after which the program resumes its untampered operation. Like a delayed crash, this length of time gives the attacker information about whether any of the tampered nodes contained integrity checks. With this modification, the original analysis of the graph-game model [15] essentially applies as is.

For concreteness, we describe a simplified variant of this modeling approach. Assume the protected program contains n integrity checks, each executed with probability p. Let d denote the average time required by the attacker to determine that a node contains an integrity check. To identify all integrity checks, the attacker must run the program an average of n/p times, each taking time d.

Thus, the time required by the attacker is dn/p. If we set p to $1/n$, the time becomes dn^2, or quadratic in the number of checks.

Like the original graph-game model, this approach makes a number of "engineering assumptions" that must be approximated in practice. For example, all integrity checks must be distinct, so that an attacker cannot easily use knowledge of one check to identify others. In addition, converting a typical CFG to a "semi-random" graph, along with approximating execution as a "semi-random" walk, may incur significant performance and code-size penalties. Nonetheless, software-protection techniques like individualization and opaque predication can help satify these requirements. In fact, the previously described elements of TTS can be viewed as a set of engineering techniques to satisfy the model's practical assumptions about tamper-tolerance.

3.3 Impossibility Results

While general-purpose obfuscation has been proved impossible in the model of Barak et al. [7], we are not aware of analogous work for tamper-resistance. However, it is straightforward to show that general-purpose tamper-resistance (or tamper-tolerance) is impossible as well. Informally, if we assume the existence of a generic tool to tamper-protect any program, we can simply feed such a tool's output to itself. Since any tamper protection involves some form of program transformations, this would essentially force the tool to break its own tamper protection in order to modify the tamper-resistant program. Thus, such a tool cannot exist.

The graph-game model [15] actually does propose an approach for general-purpose tamper-protection. However, this model implicitly assumes that the input program is not already tamper-protected, thus rendering the above impossibility argument irrelevant. In addition, the argument assumes that the transformation tool uses no secret key or other non-public information, but the scenario is somewhat different if this is allowed. A more detailed analysis of related results for tamper-resistance and -tolerance may appear elsewhere.

4 Implementation and Experiments

We have implemented an initial version of a tool that applies some of the tamper-tolerance techniques described in this paper by transforming and compiling high-level source code. The tool is built on top of a code transformation framework based on Phoenix [23]. Phoenix is a Microsoft program analysis and compiler framework based on a common intermediate representation (IR) that provides the building blocks of the program to be transformed.

This current section describes the tool in more detail, and includes some experimental results obtained by applying the tool to several SPEC CPU2006 benchmarks.

4.1 Tool Implementation

In this initial version of the tool, we implemented the IMR/RE scheme, which incorporates many of the building blocks upon which the other schemes may be built. In addition, we also implemented a stack-based checkpointing and rollback mechanism in order to illustrate the performance under simulated tampering and detection conditions. Future work will enhance the tool to implement the IMR/V and IMR/DC schemes.

The tool processes each function in an input program and creates a configurable number of copies of the basic blocks in the function. The number of copies may be specified in a configuration file. Additionally, the functions to be processed may also be specified, and a parameter may be used to control the code coverage, or percentage of blocks processed within the functions. We could have instead opted to copy entire functions rather than basic blocks, but copying at the basic-block level provides a finer level of granularity that could allow for more targeted applications of the technique.

The multiple copies are each individualized, which in the case of our tool is achieved by applying a different combination of simple chaff-code injection and code-substitution transformations to each code block in a way that does not alter the functionality of the block. In practice, any individualization scheme that produces functionally equivalent individualized code may be used. Finally, the tool inserts code to select (randomly, as is the case for the IMR/RE scheme) one of the multiple copies to execute. In our prototype, a simple pseudo-random number generator is inlined in the code to select randomly one of the code blocks to which to transfer control.

4.2 Experimental Results

This section presents experimental results obtained by running the tool on selected SPEC CPU2006 benchmarks (data compression, transportation scheduling, database search, chess and video compression). The benchmarks were run and timed on a Pentium 3.0 GHz workstation with 3 GB of RAM. In the set of tables that follow, we measured for each benchmark the binary code size and runtime performance before and after applying the tamper-tolerance transformations. We express these measures in the tables as *ratios* relative to the baseline benchmark with no tamper-tolerance applied. The results show the impact of the addition of tamper-tolerance on the code size and runtime performance of the program.

Tables 1 and 2 show how applying IMR/RE by duplicating a random sampling of 25% of the code blocks affects the code-size and performance respectively relative to the baseline. Because not all blocks are the same size, and because we perform the duplication of blocks in high-level source code, it does not necessarily follow that the size of the resulting executable file will increase by exactly the same amount as statically-linked library code is not seen at compile-time and will not be subjected to the transformation. In addition, the random-block-selection code will add to both the code size and performance overhead. We repeated the

Table 1. Code-size impact of block redundancy, 25% code coverage

Benchmark	Number of redundant copies (n)			
	$n = 2$	$n = 4$	$n = 8$	$n = 16$
401.bzip2	1.650	2.319	3.525	5.875
429.mcf	1.189	1.400	1.611	2.256
456.hmmer	1.879	2.640	4.109	7.298
458.sjeng	1.776	2.475	3.929	6.859
464.h264ref	1.907	2.770	4.414	8.054

Table 2. Performance impact of block redundancy, 25% code coverage

Benchmark	Number of redundant copies (n)			
	$n = 2$	$n = 4$	$n = 8$	$n = 16$
401.bzip2	2.267	2.674	3.287	4.903
429.mcf	2.010	2.352	2.762	2.238
456.hmmer	2.275	2.389	2.310	4.629
458.sjeng	2.840	3.797	4.858	6.274
464.h264ref	2.222	3.655	3.909	5.274

Table 3. Code-size impact of block redundancy, 50% code coverage

Benchmark	Number of redundant copies (n)			
	$n = 2$	$n = 4$	$n = 8$	$n = 16$
401.bzip2	2.325	3.394	5.375	10.65
429.mcf	1.333	1.667	2.222	3.644
456.hmmer	2.599	4.119	7.023	13.07
458.sjeng	2.475	3.886	6.529	11.79
464.h264ref	2.701	4.394	7.699	14.32

measurements for number of copies $n = 4$, 8 and 16. This clearly shows the increase in code size due to the additional redundant code as more copies are introduced. The performance overhead may be attributed to the block-selection code as well as changes to the spatial and temporal ordering of code blocks which can affect the efficacy of CPU-based optimizations such as caching and branch prediction. These performance impacts can also vary depending on the profile of the code running the benchmark and the actual blocks selected for redundancy. For instance, duplicating a block within a performance-critical, tight inner loop would cause the block-selection code to be run on every iteration of the loop, thereby amplifying the performance overhead.

Tables 3 through 6 repeat the measurements for expanded code-coverage values of 50% and 100%. As the code coverage is increased, both code size and performance are impacted, the latter due to the increased frequency of block-selection code execution and the reduced efficacy of CPU-based optimizations. This can be seen most clearly at the 100% code-coverage level, where the

Table 4. Performance impact of block redundancy, 50% code coverage

Benchmark	Number of redundant copies (n)			
	$n = 2$	$n = 4$	$n = 8$	$n = 16$
401.bzip2	3.840	5.580	7.099	9.530
429.mcf	2.388	2.447	3.340	4.301
456.hmmer	2.834	3.738	5.808	4.803
458.sjeng	4.311	6.509	7.547	11.56
464.h264ref	4.364	5.395	6.854	10.04

Table 5. Code-size impact of block redundancy, 100% code coverage

Benchmark	Number of redundant copies (n)			
	$n = 2$	$n = 4$	$n = 8$	$n = 16$
401.bzip2	3.394	5.725	10.33	19.42
429.mcf	1.722	2.356	3.700	6.533
456.hmmer	4.149	7.235	13.26	25.50
458.sjeng	3.890	6.765	12.00	22.53
464.h264ref	4.307	7.802	14.31	27.45

Table 6. Performance impact of block redundancy, 100% code coverage

Benchmark	Number of redundant copies (n)			
	$n = 2$	$n = 4$	$n = 8$	$n = 16$
401.bzip2	6.205	9.307	12.13	16.74
429.mcf	3.990	5.324	6.618	10.10
456.hmmer	4.934	7.293	9.651	13.49
458.sjeng	8.255	12.31	16.70	25.71
464.h264ref	6.801	9.880	12.76	18.07

redundancy is applied to every block in the source code. We should also point out that the block-selection code was not optimized by the compiler, which may account in part for the slower execution times as the degree of redundancy is increased. This could be addressed by encoding the block-selection code using low-level jump tables.

In order to illustrate in general how checkpointing and rollback could be used in conjunction with IMR/RE as an effective means of result correction in the presence of tampering, we implemented a simple stack-based scheme to checkpoint the execution environment of the program before a function call, and rollbacks to restore the execution to the most recent checkpoint. This scheme does not currently support saving and restoring global state, so we are limited to applying this only in the absence of global side-effects.

We assume different probabilities of tampering within the program, and simulate detection and correction by injecting code in each function to perform a rollback to the previous checkpoint with probability p. Upon a successful rollback,

Table 7. Performance impact of rollback

Benchmark	Probability of tampering (p)				
	$p = 0.9$	$p = 0.8$	$p = 0.7$	$p = 0.6$	$p = 0.5$
401.bzip2	2.720	1.875	1.571	1.432	1.324
429.mcf	5.735	3.373	2.578	2.137	1.863
456.hmmer	2.175	1.611	1.415	1.332	1.262
458.sjeng	6.132	3.755	2.943	2.542	2.222
464.h264ref	9.679	5.395	3.936	3.159	2.651

execution will resume from the previous checkpoint, and will have a probability of $(1-p)$ of successfully proceeding beyond the tampered block. In the IMR/RE scheme, the probability that the same tampered block will be executed is $1/n$, where n corresponds to the number of copies made of the code block.

The performance impact of checkpointing and rollback under different tampering probabilities is presented in Table 7. The values in the table are again expressed as a ratio relative to the baseline execution with no tampering or rollback. As the probability p of encountering tampering decreases, the performance improves due to a greater likelihood that the tampered code is avoided upon restoration of execution following a rollback.

In all of the experiments, we applied the transformation over a set percentage of all code blocks in order to be able to perform relative measurements across the selected benchmarks. More realistic usage may involve targeted selection of sensitive code sections on which to apply tamper-tolerance, as well as protecting less critical parts of the program to avoid drawing attention to the former. As with other software-protection schemes, these need to be balanced with the code-size and performance impacts. While the current version of the tool applies the transformations to high-level source code for cross-platform compatibility, the methods described in this paper may equally be applied to low-level machine code. Finally, practical application of this and other tamper-tolerance schemes should always be done in conjunction with other protection methods as part of an overall software-protection solution.

5 Conclusion

This paper proposed the general concept of *tamper-tolerant software*, or the notion of an attacked program correcting its own operation upon tamper detection, as opposed to traditional responses that involve crashes or graceful degradation. Tamper-tolerance enables a program to continue executing rather than to fail upon attack detection. TTS is based on *individualized modular redundancy*, namely a combination of software protection and fault-tolerance techniques adapted to the malicious-attacker scenario. The approach detects tampering and fixes malicious changes via voting, result correction, randomized re-execution, or rollback. We model TTS by adapting a graph-based

tamper-resistance framework [15], enabling security analysis and estimation of attack resistance in practice.

Future work will analyze possibility and impossibility results for TRS and TTS, extending the informal discussion from Section 3. In particular, we are investigating classes of programs in terms of how well TRS and TTS can protect them, both with and without secret keys and other auxiliary information. A main goal is to put TRS and TTS on a sound yet practically useful formal foundation, eliminating the need for "ad hoc" heuristics and unpredictable security in real-world contexts.

References

1. Anckaert, B., Jakubowski, M.H., Venkatesan, R.: Proteus: Virtualization for diversified tamper-resistance. In: DRM 2006: Proceedings of the ACM Workshop on Digital Rights Management, pp. 47–58. ACM Press, New York (2006)
2. Anckaert, B., Jakubowski, M.H., Venkatesan, R.: Runtime protection via dataflow flattening. In: IARIA SECURWARE 2009 (to appear, 2009)
3. Anckaert, B., Jakubowski, M.H., Venkatesan, R., De Bosschere, K.: Run-time randomization to mitigate tampering. In: Miyaji, A., Kikuchi, H., Rannenberg, K. (eds.) IWSEC 2007. LNCS, vol. 4752, pp. 153–168. Springer, Heidelberg (2007)
4. Anckaert, B., De Sutter, B., De Bosschere, K.: Software piracy prevention through diversity. In: DRM 2004: Proceedings of the 4th ACM Workshop on Digital Rights Management, pp. 63–71. ACM Press, New York (2004)
5. Aucsmith, D.: Tamper resistant software: An implementation. In: Anderson, R. (ed.) IH 1996. LNCS, vol. 1174, pp. 317–333. Springer, Heidelberg (1996)
6. Avizienis, A.: The methodology of N-version programming. In: Lyu, M.R. (ed.) Software Fault Tolerance,ch. 2, pp. 23–46. Wiley, Chichester (1995)
7. 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)
8. Bhansali, S., Chen, W.-K., de Jong, S., Edwards, A., Murray, R., Drinić, M., Mihočka, D., Chau, J.: Framework for instruction-level tracing and analysis of program executions. In: VEE 2006: Proceedings of the 2nd international conference on Virtual execution environments, pp. 154–163. ACM, New York (2006)
9. Chang, H., Atallah, M.J.: Protecting software code by guards. In: Digital Rights Management Workshop, pp. 160–175 (2001)
10. Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., Jakubowski, M.H.: Oblivious hashing: A stealthy software integrity verification primitive. In: Information Hiding 2002, Noordwijkerhout, The Netherlands (October 2002)
11. Cloakware Corporation. Software Security Suite (2009)
12. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, New Zealand (July 1997)
13. Collberg, C., Thomborson, C., Low, D.: Breaking abstractions and unstructuring data structures. In: International Conference on Computer Languages, pp. 28–38 (1998)
14. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Principles of Programming Languages, POPL 1998, pp. 184–196 (1998)

15. Dedić, N., Jakubowski, M.H., Venkatesan, R.: A graph game model for software tamper protection. In: Proceedings of the 2007 Information Hiding Workshop (June 2007)
16. Denning, P.J.: Fault tolerant operating systems. ACM Comput. Surv. 8(4), 359–389 (1976)
17. Goldwasser, S., Kalai, Y.T.: On the impossibility of obfuscation with auxiliary input. In: FOCS 2005: Proceedings of the 46th IEEE Symposium on Foundations of Computer Science (2005)
18. Horne, B., Matheson, L.R., Sheehan, C., Tarjan, R.E.: Dynamic self-checking techniques for improved tamper resistance. In: Digital Rights Management Workshop, pp. 141–159 (2001)
19. Jacob, M., Jakubowski, M.H., Venkatesan, R.: Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In: 2007 ACM Multimedia and Security Workshop, Dallas, TX (september 2007)
20. Jakubowski, M.H., Naldurg, P., Patankar, V., Venkatesan, R.: Software integrity checking expressions (ICEs) for robust tamper detection. In: Furon, T., Cayre, F., Doërr, G., Bas, P. (eds.) IH 2007. LNCS, vol. 4567, pp. 96–111. Springer, Heidelberg (2008)
21. Linden, T.A.: Operating system structures to support security and reliable software. ACM Comput. Surv. 8(4), 409–445 (1976)
22. Lynn, B., Prabhakaran, M., Sahai, A.: Positive results and techniques for obfuscation. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 20–39. Springer, Heidelberg (2004)
23. Microsoft Corporation. Phoenix compiler framework (2008)
24. Moon, T.K.: Error Correction Coding: Mathematical Methods and Algorithms. Wiley-Interscience, Hoboken (2005)
25. Feng, Q., Joseph, T., Yuanyuan, Z., Jagadeesan, S.: Rx: Treating bugs as allergies—a safe method to survive software failures. ACM Trans. Comput. Syst. 25(3), 7 (2007)
26. Randell, B.: System structure for software fault tolerance. In: Proceedings of the International Conference on Reliable Software, Los Angeles, California, pp. 437–449. ACM, New York (1975)
27. Randell, B., Lee, P., Treleaven, P.C.: Reliability issues in computing system design. ACM Comput. Surv. 10(2), 123–165 (1978)
28. Siewiorek, D.P., Swarz, R.S.: Theory and Practice of Reliable System Design. Digital Press, Bedford (1982)
29. Tan, G., Chen, Y., Jakubowski, M.H.: Delayed and controlled failures in tamper-resistant software. In: Proceedings of the 2006 Information Hiding Workshop (july 2006)
30. Tyrrell, A.M.: Recovery blocks and algorithm-based fault tolerance. In: EUROMICRO Conference, vol. 0, p. 292 (1996)
31. Varadarajan, A.V., Venkatesan, R., Rangan, C.P.: Data structures for limited oblivious execution of programs while preserving locality of reference. In: DRM 2007: Proceedings of the 2007 ACM workshop on Digital Rights Management, pp. 63–69. ACM, New York (2007)
32. Wang, C., Hill, J., Knight, J., Davidson, J.: Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia (December 2000)
33. Wee, H.: On obfuscating point functions. In: STOC 2005: Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing, pp. 523–532. ACM Press, New York (2005)