DARE: a Framework for Dynamic Authentication of Remote Executions

Erdem Aktas, Kanad Ghose
Department of Computer Science
State University of New York, Binghamton, NY 13902
{eaktas,ghose}@cs.binghamton.edu

Abstract

With the widespread use of the distributed systems comes the need to secure such systems against a wide variety of threats. Recent security mechanisms are grossly inadequate in authenticating the program executions at the clients or servers, as the clients, servers and the executing programs themselves can be compromised after the clients and servers pass the authentication phase. This paper presents a generic framework for authenticating remote executions on a potentially untrusted remote server – essentially validating that what is executed at the server on behalf of the client is actually the intended program. Details of a prototype Linux implementation are also described, along with some optimization techniques for reducing the run-time overhead of the proposed scheme. The performance overhead of our technique varies generally from 7% to 24% for most benchmarks, as seen from the actual remote execution of SPEC benchmarks.

Keywords: Computer Security, Signature-based Authentication, Trusted Computing.

1. Introduction

As distributed systems become pervasive, security mechanisms have to be geared up to meet the increasing threats against such systems. Existing security mechanisms for distributed systems rely on the authentication of the clients and servers and on the protection of the data being processed or communicated using known encryption mechanisms on secure channels. These security mechanisms are grossly inadequate in authenticating the program executions at the clients or servers, since the clients, servers and the executing programs themselves can be compromised after the clients and servers pass the authentication phase. Existing techniques that implement a dynamic root of trust based on support for Trusted Platform Modules (TPMs) [Int 04, TCG 07, MPP+ 07] represent a good first step in validating remote execution on a potentially untrusted host. A dynamic root of trust is implemented by essentially validating the signature of an executable just prior to its execution. As such, these mechanisms do not address the vulnerabilities possible at run time in large software systems that often call dynamically linked modules either locally or remotely, nor do they address the dynamic substitution of regions of the executables by malicious components in the host kernel. Our research specifically addresses these current limitations of mechanisms that implement just a dynamic root of trust. We do this by continuously validating the control flow signature of remotely executed code as they execute. We believe that our approach will be extremely relevant towards implementing secure distributed systems where some parts of the system run on potentially untrusted hosts.

Existing mechanisms for validating remote execution are directed at detecting security breaches at a remote server that executes programs on behalf of a client. Some examples of such approaches are:

- The binaries of the program executing remotely can be altered or corrupted in the remote host.
- The process running on behalf of the server or libraries used by it can be altered or corrupted at run time, or their binaries can be corrupted before execution.
- The authentication process at the server, such as the ones used in some current solutions [Ka 07, UCT 04, HCF 04] may itself be compromised.

Unfortunately, all of the existing techniques for authenticating remote execution are impractical, or limited in the protection they offer and may have adverse performance implications, as we describe later in Section 2. The technique described in this paper for authenticating remote execution addresses these limitations of existing techniques. Solutions based on the validation of the code just before the execution (static validation) are not sufficient as a compromised memory management or file system module can supply the original code to the verification function while executing the compromised code. The static validation of executions is also limited in its inability...
to address compromises made by code injection at run-time, typical of some viruses. We thus have an emerging need for a mechanism that validates a process at run-time. The specific problem that we address in this paper is the dynamic (that is, run-time) authentication of code running on a potentially untrusted remote server at a client’s request. Such a scenario is typical in the world of today’s Internet, where applications are executed on servers to serve a request from a client. Existing solutions addressing this or similar problems assume the availability of some trustworthy components at the potentially untrusted server [Ka 07, AMD 05, SPD 04, Int 04, Int 07] and the most practical and acceptable solutions available make use of the Trusted Platform Module (TPM) [Int 04, TCG 07].

We present a general framework for authenticating the executions of programs on a remote server in a distributed environment – essentially validating that what is executed at the server on behalf of a client is actually the intended program. The framework for building such systems essentially provides answer to the question "is the program running on the remote host really the program I wanted to run?" Our approach relies on the continuous validation of the control flow signatures of the program executing at the server. A verification node, which could be the client itself, continuously validates the control flow signatures for the execution at the server through a challenge-response sequence. The verifier specifies randomly-chosen points (random "checkpoints") within the control flow from a set of checkpoints identified from an a-priori analysis of the executable. The verifier challenges the server to verify a control flow signature at each such checkpoint. A random "session" key is also specified as part of the challenge and the server responds with a signature for the control flow at the specified checkpoint, combining the session key with the signature. The use of the verifier-specified random session key for generating the signature at the server also ensures that the signatures are immune to replay attacks. As a further safeguard, the generation of a signature can randomly overwrite the contents of specific hardware-maintained instrumentation registers (that are commonly found in desktop and server CPUs). By using signatures that incorporate the contents of such instrumentation registers, we alter the generated signature for the next checkpoint within the control flow, again thwarting replay attacks. Our approach is thus quite different from prior approaches to control flow validation that require source-code level modifications and use signature validation at predictable points within the control flow path.

Our proposed framework called DARE (Dynamic Authentication of Remote Executions), has infrastructures to support a variety of signature generation schemes as well as the associated functions that are required as part of the signature validation process. The current Linux prototype of DARE on X86 and X86/64 based hosts incorporate a variety of optimizations for reducing the run time overhead of DARE authentication mechanism. The performance overhead of DARE varies generally from 7% to 24% for most benchmarks, as seen from the actual remote execution of representative SPEC benchmarks. The prototype implementation uses the addresses and outcomes of the last four branch instructions that are executed (as held in some specific instrumentation registers called Model Specific Registers, MSRs), cache miss statistics (again, held in MSRs) and user-level data constraints for the signatures.

2. Related Work

Implementing a software solution for validating remote execution is an open area of research and only a few practical solutions have been proposed to date. Techniques for monitoring system call sequences for intrusion detection, such as [HFS 98], can be extended for authenticating remote execution. However, these techniques will have a system-wide performance overhead. Also, an attacker who simulates the correct system call sequence using a maliciously injected code can easily mislead the system call sequence monitoring mechanism.

Baldi et al introduced the TrustedFlow protocol as one of the first approaches that authenticate the execution of code on a remote host by using idiosyncratic signatures (“tags”) at specific points in the control flow path [BOY 03a, BOY 03b]. The first problem of the trusted flow protocol is in its assumption that current software technology is enough to obfuscate the functions for generating signatures. This assumption is questionable and in any case, as has been said earlier, obfuscation is never a solution for security. The second problem with the TrustedFlow approach is that an attacker can run a malicious program and the original program simultaneously and still redirect the correct signature sequence from the original code, while the malicious code does something else. A third limitation has to do with the tampering of the code of the target program, without impacting any of the code for the obfuscated function generator. Detecting such a violation also requires additional trusted support on the remote host.

Kennell and Jamieson [KJ 03] has used the side-effects of a running process, such as the number of misses and hits on the instruction and data TLBs,
performance counter values (executed instruction count, executed branch count, etc.), and a special random tag generator to generate unique signatures for a remotely executed program. It is unclear how the code for generating the signature is integrated with the executing program on the remote system. In a subsequent paper [UCT 04], Shankar et al mention the weakness of the Kennell and Jamieson’s technique. Kennell and Jamieson’s approach relied on the use of simulators or emulators for gathering the correct checksum values that serve as the signature. Furthermore, the checkpoint locations are possibly defined statically, so an attacker can determine (by using similar simulators/emulators) the correct signatures, thereby compromising the mechanism.

Xu et al propose a technique for detecting anomalies in the execution of a program by monitoring control flow into statically identified basic blocks of the binary [XDC 04]. A fundamental limitation of this technique has to do with the static, a priori marking of basic blocks, making the technique prone to compromise. The technique of Xu et al monitors the system call sequence by altering the kernel system call trapping mechanism. This implies that any system call will trigger the anomaly detection mechanism first, resulting in serious performance degradation for other programs that are not being validated, including the OS itself.

The Trusted Computing Group [TCG07] has standardized the concept of a Trusted Platform Module (TPM), a hardware device for generating and storing a secure hash value. The TPM can be incorporated into a computing platform and can serve as the basis for a root of trust [Ka 07]. Many modern processors or chipsets incorporate such a TPM [AMD 05, Int 04, MPPRS 07]. Seshadri et al [SLS+ 05] has developed a remote execution authentication mechanism, called Pioneer, based on the use of a software implementation of a root of trust. Pioneer is designed for legacy systems that lack hardware support for attestation, such as a TPM. Pioneer relies on the knowledge of the exact machine details at the executing end and relies on timing bounds to avoid any compromise on the hash used to authenticate the boot code, which is the root of trust.

Monrose et al [MWR 99] rely on the execution of traces at a verifier for participants in a SPMD style distributed computation to verify if the participants performed their computations correctly. The traces are derived from the call stack states of the participants and the technique essentially compares the call stack information of remote participants with that at the verifier. The applicability of this scheme to a general purpose distributed system is thus limited.

3. The DARE Framework

DARE is a generic framework for validating the execution of binaries, supplied by a client, on a remote server (hereafter called the Compute Server, CS). The CS is a potentially untrusted host. We now describe the features and facilities within DARE.

3.1. Assumptions

DARE’s goal is to validate the execution of code on a potentially untrusted server even though part or all of these servers, including the operating system on the servers, can be compromised. Short of re-executing the binaries on a trusted host and verifying the results (or signatures) against those from the untrusted host, we cannot do anything to validate the execution on the untrusted host. Validating executions on a host where all components are suspect is thus not practically viable. Instead, we rely on the following trusted components within the CS:

1. There is a secure storage on the server for a key supplied by the client. DARE needs to keep all sensitive data (i.e. checkpoint locations, keys for secure channels, and information related to checkpoints) encrypted on the CS. The key (called the master key) for encrypting and decrypting such data and the related encryption and decryption functions, in turn, have to be themselves kept secure. We assume that a secure storage mechanism to store and secure the master key is available. Such storage can be implemented using a dynamic root of trust mechanism built on TPM support, as described in [Int 07].

2. At least two new system calls (described later in Section 3.3) are trusted. This can again be implemented using a dynamic root of trust mechanism, as in [Int 07, SLQP 07] because the two new system calls in question are relatively short and authenticating them prior to their execution may still be practically viable. This assumption does not translate to a strict requirement, but is more of a matter of convenience and performance, as the two system calls as well as the interrupt handler can be validated using the proposed validation mechanism or using a dynamic root of trust mechanism.

3. Library functions on the server that are called by the program being validated are secure. In Section 5 we show how the proposed solution can be extended to validate these library functions.

4. The communications between the server and the client takes place on secure channels.

The two trusted system calls mentioned above are described later in Sections 3.3.2 and 3.3.3.
3.2. The Major Components of DARE

DARE consists of the following components:

- A static analyzer that runs at the client that requests the execution of a program on a remote server. The remote server (called the computation server, CS) is potentially untrusted. This analyzer identifies the locations of the control flow checkpoints and determines their associated signatures.

- A challenge generator that runs on a trusted host that performs the authentication checks. This host (called the authentication server, AS) could be the client machine itself. The challenge generator sends a list of randomly chosen checkpoints that need to be enabled for signature generation and other information to prevent replay attacks.

- A checkpoint enabling module that runs on the remote and potentially untrusted host, CS. This module enables the specified checkpoints by dynamically inserting calls to functions for generating signatures and stores the original information at the location of these calls for later restoration during the actual execution of the program being validated. This module is invoked through one of the trusted system calls (Sec. 3.1).

- A signature generator that runs on the CS and generates control flow signatures at the enabled checkpoints. The signature generator is invoked through the second of the trusted system call mentioned in Sec. 3.1. The generated signature is sent to the AS on secure channels as the response to a challenge. Some additional processing is required in the signature generation step as described later.

- A signature verifier that runs on the AS, whose role is to verify if the response to a challenge matches the expected signatures and takes appropriate actions on potential validation failures.

Figure 1 depicts the various hosts and the messages they exchange as part of the authentication process in the DARE framework.

3.3. Functional Components of DARE

We now present DARE’s functional components.

3.3.1 The Static Analyzer

DARE’s static analyzer is used on the client side to perform a basic block level data flow analysis of the binaries of the program whose remote execution has to be authenticated. This analysis derives and uses control flow probabilities into basic blocks as is routinely done in many modern optimizing compilers. (A basic block is a set of consecutive instructions that does not have any branch.) We do this by first identifying the frequently called functions (“call blocks”) and then analyzing the basic blocks within each such call block. A call block is a series of basic block starting from the call instruction and ending with a ret instruction.

Detecting the dynamically linked functions is easy but deciding the critical local functions that are called frequently is not as easy. To identify the most critical basic blocks — that is blocks with the higher execution frequencies, we need to generate a jump tree that holds calls and indirect jump sequences for each call block. We developed a utility to automate this step. This utility program generates the basic blocks, their call frequencies and constructs the jump table. For added flexibility, DARE permits the programmer to modify the critical basic block list produced by the analyzer.

This static analysis identifies a full set of checkpoints within the most-likely control paths, located at the entry point to the most frequently executed basic blocks and ensures an acceptable level of coverage for verifying the control flow within the program as it executes. This analysis also generates information that can be incorporated into verifiable signatures at each of these checkpoints.

3.3.2 The checkpoint enabling module

This module is implemented as a system call, “sys_startcheck”, which selects a random subset of the checkpoints from the full set of such checkpoints, with the random selection specified by the verifier. This random selection also ensures that a good coverage of the control flow path. sys_startcheck() is invoked once before application is run to request a challenge from the verification server. The verification server will respond with challenge that includes a randomly generated tag value (hereafter called the session tag) and a list of checkpoint locations. After receiving the checkpoint list, the original codes at checkpoints will be substituted with system call for signature generation and the original binary contents at each such location will be saved. These original contents are restored after successful signature generations and verifications at the checkpoints to enable the original
executions to continue.

3.3.3 The Signature Generator

The signature generator is implemented as another system call, “sys_checkpoint”, which is inserted by the verifier component when needed. When the verification commences, the CS will wait on a challenge from the AS. As this initial challenge, the AS will send a list of randomly selected checkpoint locations, or modify the existing list of checkpoint locations by adding or deleting checkpoint locations.

- Collects the signature-specific information and generate the signature for the current checkpoint location. The nature of the signature will be described in the Section 4.
- Encrypts and sends the signature combined with the session tag to the authentication server.
- Receives the response from the authentication server and take appropriate action depending on the response. The authentication server can send a new list of the checkpoint locations, or modify the existing list of checkpoint locations by adding or deleting checkpoint locations.
- Patches the original code in the previously passed checkpoint with a call to “sys_checkpoint” to permit signatures to be generated correctly if a signature generation is required at a future time at this previous checkpoint. The original bit sequence at the previous checkpoint’s location is saved before this patchup.
- Copies back the original binary contents to the current checkpoint location, change the X86 eip register’s value appropriately and let the original execution continue until the next checkpoint.

3.3.4 The Challenge Generator and Verifier

The challenge generator is called by the signature verifier component when needed. When the verification commences, the CS will wait on a challenge from the AS. As this initial challenge, the AS will send a list of randomly selected checkpoint locations, as described earlier. This random selection and enabling of checkpoints makes it very difficult to predict the sequence of signatures expected as responses to subsequent challenges by the AS. In effect, we dramatically reduce the possibility of pre-executing the program on a remote host and generating the signatures in advance for later replay by the untrusted CS in an effort to deceive the AS. To see this, suppose we have N call blocks and each call block has m basic blocks that are critical – that is, have high execution frequencies. The challenge generator will randomly select k basic blocks from these m basic blocks at the beginning of the challenge. Therefore, the total number of different checkpoint locations in the checkpoint list will be equal to C(m,k)^N where C(x,y) stands for the combination function. As an example if N=200, m=10, and k=3 (which are fairly representative of a small to medium-sized application) then, the total number of different outcomes is (10! /3! x [10-3]!)^200 ≈ 1.2 x 10^400. We believe that this low probability of predicting a specific random sequence of signatures make it possible for DARE to certify typical remote executions as genuine. If this is not enough, the DARE framework permits the AS to alter the list of enabled checkpoints dynamically. In fact, as described in Section 8, we use a similar mechanism to handle checkpoint within loops and for reducing the associated performance penalty.

4. The Nature of the Signature

The crux of our authentication mechanism lies in the ability to characterize the unique, expected control flow signatures at dynamically and randomly chosen checkpoints from a set of pre-generated checkpoints. The signature at a checkpoint is multi-dimensional in nature and has various components that identify the execution path taken to the checkpoint as well as components that are indicative of the state of execution at the checkpoint. To derive a signature, we can use a variety of information maintained in MSR registers, such as number of cache hits, stall cycle count, sequence of past few system calls and other such information. Contents of general purpose registers can also be used as signature components. In general, a signature has some exact components (specific register values, last few branch instruction addresses, last system call id), as well as components (“inexact values”) that can be matched to expected values within a given range of deviations (such as number of cache hit, committed instruction counts, stall cycle counts etc.). (Committed instruction counts are not exact; program branches can cause variations.) Because of these two diverse class of components in the signature, signature verification is not just a comparison for equality but rather a mix of comparison for equality and comparison for matched within a pre-specified range.

The actual signature generated can also use constraints based on specific register values (regular registers, not MSRs) and verify, for example, if the value of register X is strictly greater than the value of register Y. In general, the more the number of components we use in a control flow signature, the better are the chances of producing a unique signature for the control flow path to the point of verification.

Given the features of contemporary hardware, what constitutes a practically viable, robust and unique signature for a checkpoint is a legitimate area of
research by itself. We are investigating a number of approaches for identifying and using exact and "inexact" components for a control flow signatures that allow a unique control flow signature to be associated with a control flow path.

We have developed DARE as a generic framework. Users are free to choose signature components and verification functions depending on the target application’s need. In the current implementation signature components chosen are unaffected by a context switch or process migration to another core. If components affected by a context switch are present, one solution will be to reinitialize them on context switches.

5. Run-time Validation of CS Libraries

One naïve way to implement assumption (3) of Sec. 3.1 is to statically link the library modules with the binaries of the program being validated and validate the resulting binary using the proposed framework. Although, this approach works, it is not desirable, as all the advantages of dynamically linked libraries are lost. An alternative approach will be to set up an interrupt handler to generate a signature on every m committed instructions where m is chosen at random and small enough for a good coverage and big enough for an acceptable performance penalty, as in [CM 05]. This, of course, requires the timer interrupt mechanism to be validated using a dynamic root of trust or similar mechanisms. Yet another approach for validating library functions will be to use the per-branch tracing mechanisms supported in some modern hardware to track each and every branch executed in the library code (Sec. 7). The current version of DARE does not validate the library functions. Note that our approach to validate the execution of library functions does rely on the use of some smaller (and fixed) components that are certified using a dynamic root of trust mechanism. This is still significantly better than validating these libraries before their execution using a dynamic root of trust mechanism that leaves out any run-time checks. Finally, note that we prefer not to verify the actual program using the approach just described for library functions, as our signature validation approach for the program is at a more finer-grained level and thus more robust. Where a similar level of validation is necessary for the library functions, the practical choice will be to link them in statically with the program binaries and validate them like the rest of the program itself.

6. Run-time Validation of Signatures

Run-time verification of control flow signatures in DARE proceeds as follows:

1. (CS side) Prior to commencing the actual execution, the CS connects to the verifier (AS, authentication server) over a secure channel and sends a challenge request message to the AS.
2. (AS side) As the initial challenge, the AS will send a randomly generated tag and a list of the checkpoint locations which are selected from a full-set of pre-generated checkpoint locations (Sec. 3.3.1).
3. (CS side) After receiving the checkpoint list, the CS will enable the specified checkpoints and patch the original program code with a call to "sys_startcheck", as described in Sec. 3.3.2 and starts the program execution.
4. (CS side) When a checkpoint location has been reached, the CS will generate a signature, combine the signature with the randomly generated session tag received in Step 2, encrypt this signature and send it to the AS for verification. The encryption key is stored securely in the CS (Sec. 3.1).
5. (AS side) If the signature verification component in the AS validates the signature sent by the CS, it sends a go-ahead message to the CS. This go-ahead message can optionally change the set of currently enabled checkpoints and re-initialize variables/system-level activity counters when necessary. If the validation fails, the AS can enable additional checkpoint locations to scrutinize any suspicious behavior (or rule out false positives) through a more frequent challenge-response exchanges or abandon server-side execution and mark the server execution as compromised (or untrusted). The client decides on actions that have to be taken when validation fails.
6. (CS side) After receiving a go-ahead message, the server side system call restores the original binary contents for the current checkpoint and returns. Before returning to the original execution, the original code in the previously validated checkpoint location will be re-substituted with a call “sys_checkpoint” (Sec. 3.3.3) to permit a validation of the signature at this previous checkpoint, should it be required subsequently.

7. Resiliency of DARE

In this section, we provide non-formal arguments to establish that the proposed authentication scheme is secure, as a formal proof of security is well-beyond the scope of this paper.

First, we examine the impact of compromising the two functions calls installed at the CS. If a different executable is run at the CS and these two system calls are compromised - either by substitution or through trapping and emulation - the signature generated will
not match the expected signature, as the contents of the MSR registers cannot be reproduced. The signature produced at the checkpoint is very likely to have both exact and inexact components (Sec. 4) that will fail to match the expected signature. There is, of course, a pathological case, where the signature of a bogus code will match the expected control flow signature at a checkpoint. Note, however, that even with such a highly improbable "accidental" signature match at a specific checkpoint, it is practically impossible to have such matches consistently for a series of randomly chosen checkpoints. Put in other words, such accidental matches are not likely to continue across a contiguous series of checkpoints, making it possible for the proposed scheme to easily identify bogus, unauthenticated execution at the CS on behalf of a client. What if the signature generation functions are compromised? Here again, following arguments similar to the one we just made, we can easily see that even if a single signature can be made to match the expected signature, producing a series of such matches is practically impossible.

It is also not possible to trap to an emulator from the system call for signature generation and generate the signatures correctly. To correctly reproduce some of the inexact signature components such as number of cache hits and number of stall cycles, the entire execution has to be emulated. This is because the emulation on traps cannot reproduce the states of internal hardware components such as caches, branch predictors and load miss predictors.

DARE is highly resistant to buffer overflow attacks, direct code injection or library substitution attacks since:

- DARE works on basic blocks, and these attacks limit the malicious user to compromise only one basic block without any branch, seriously limiting the extent to which the attacker can compromise the application. If the malicious code injects a branch out of the compromised block, the execution of the branch is recorded in the MSRs that keep track of branches.

- If one basic block produces unexpected results (as values stored in general registers), this basic block will be assumed as suspicious and the AS can dynamically generate additional challenges, with an increased frequency to get a higher coverage for the control flow and thus detect if the unexpected results are a consequence of compromise or a false positive.

- These attacks, in general, will affect components of the global program state, as reflected in the contents of MSRs that store cache access statistics, TLB statistics, system call statistics and other similar information. DARE can thus detect these attacks, as it includes these MSRs in the “variable” part of the signature (Sec. 4).

Provided that the assumptions of Section 3.1 are implemented as described in that section, replay attacks are not possible, as unique session tags are used for each challenge-response pair on secure channels. Also, as explained in Section 3.3.2, DARE’s use of randomly and dynamically chosen checkpoints makes it practically impossible to generate a sequence of fake signatures that match the expected signatures.

The accuracy with control flow is validated by DARE is a function of the number of checkpoints that are dynamically enabled as the code executes. However, one has to note that the ratio of the average number of checkpoints at which signatures are verified to the number of basic blocks in the program being verified is not an indicator of the control flow coverage achieved by DARE. This is because at each checkpoint, we have the record of the last four braches executed (in the Pentium® 4 and the Intel® Xeon® Processor Family CPUs, this number is 16), so that we actually have a record of control flow through the 4 preceding basic blocks (recall that a basic block is a sequence of code with one entry point and one exit point, with no branches in-between the entry and exit points). Many new Intel processors also permit the logging of all executed branches in a branch trace stack in memory [Int 08]. If finer-grained control flow checking is necessary, this branch tracing mechanism can be optionally used at the cost of performance. In addition, global state information such as cache miss statistics, instruction commit rates, TLB miss rates, user-level variables provide an indirect coverage of control flow across several basic blocks and potentially for the entire control flow path up to that checkpoint.

8. Performance Evaluation

From a performance perspective, authenticating an execution as described can have a large overhead if signatures are generated at closely spaced checkpoints. This is so as the generation of a signature requires a system call to read and update MSR registers, signature encryption and possible network communication with the AS. The return from this system call requires memory-to-memory copying to restore the 7 bytes of the original binary before execution continues. To reduce the communication and verification overhead, DARE buffers the generated signatures and verifies a set of signature at each challenge but that reduction is not enough by itself for each application so DARE supplies optional performance enhancement method.
The optional approach of reducing the signature generation and verification overhead is to reduce the number of checkpoints at which signatures are generated. There is a price to be paid for this – we now execute longer sections of code between verifications. Our approach here is to disable signature checking at some specific checkpoints temporarily: if a specific checkpoint location is trapped and verified N times, this checkpoint location will be disabled for the execution of the next M instructions. Here, the values of N and M are highly dependent on the applications and may be chosen empirically for each checkpoint or a common set of values for M and N can be used globally. The key idea is to choose M and N in a manner that keeps the verification overhead to an acceptable level.

Table 1. Execution statistics with the proposed authentication scheme and using temporary disabling of checkpoints for heavily encountered checkpoints

<table>
<thead>
<tr>
<th>Number</th>
<th>Soplex</th>
<th>Trap Count</th>
<th>%</th>
<th>Runtime(s)</th>
<th>Trap Count</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>125.20</td>
<td>17092049</td>
<td>36.8</td>
<td>242.68</td>
<td>34108</td>
<td>7.9</td>
<td></td>
</tr>
<tr>
<td>116.29</td>
<td>1645177</td>
<td>35.8</td>
<td>243.62</td>
<td>34006</td>
<td>7.9</td>
<td></td>
</tr>
<tr>
<td>112.14</td>
<td>1623332</td>
<td>34.4</td>
<td>243.68</td>
<td>34014</td>
<td>7.9</td>
<td></td>
</tr>
<tr>
<td>108.28</td>
<td>1565952</td>
<td>32.9</td>
<td>242.12</td>
<td>34019</td>
<td>7.9</td>
<td></td>
</tr>
<tr>
<td>109.73</td>
<td>1458943</td>
<td>33.3</td>
<td>243.02</td>
<td>33552</td>
<td>7.9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>Mcc</th>
<th>Trap Count</th>
<th>%</th>
<th>Runtime(s)</th>
<th>Trap Count</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>71.63</td>
<td>863557</td>
<td>33.4</td>
<td>14943</td>
<td>8.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78.10</td>
<td>78512</td>
<td>33.4</td>
<td>14137</td>
<td>8.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45.31</td>
<td>655389</td>
<td>31.5</td>
<td>13274</td>
<td>8.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>79.12</td>
<td>62079</td>
<td>7.6</td>
<td>13807</td>
<td>8.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>Libquantum</th>
<th>Trap Count</th>
<th>%</th>
<th>Runtime(s)</th>
<th>Trap Count</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>76.72</td>
<td>1303671</td>
<td>36.6</td>
<td>2844.59</td>
<td>3827346</td>
<td>63.5</td>
<td></td>
</tr>
<tr>
<td>63.12</td>
<td>805393</td>
<td>22.4</td>
<td>2175782</td>
<td>300.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>88.45</td>
<td>84092</td>
<td>30.9</td>
<td>282942</td>
<td>4.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76.09</td>
<td>6689</td>
<td>6.5</td>
<td>96004</td>
<td>0.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>71.18</td>
<td>7000</td>
<td>5.6</td>
<td>12240</td>
<td>0.6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>Bzip2</th>
<th>Trap Count</th>
<th>%</th>
<th>Runtime(s)</th>
<th>Trap Count</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>76.96</td>
<td>82097</td>
<td>37.2</td>
<td>814.18</td>
<td>60147</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>65.82</td>
<td>139265</td>
<td>35.76</td>
<td>847.74</td>
<td>35984</td>
<td>0.7</td>
<td></td>
</tr>
<tr>
<td>66.13</td>
<td>38839</td>
<td>24.14</td>
<td>390.03</td>
<td>39708</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>66.98</td>
<td>69669</td>
<td>20.0</td>
<td>377.83</td>
<td>39317</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>66.98</td>
<td>69693</td>
<td>20.0</td>
<td>377.83</td>
<td>39317</td>
<td>0.2</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Execution statistics with the proposed authentication scheme and using temporary disabling of checkpoints for heavily encountered checkpoints

To control the number of instructions executed after disabling a checkpoint location, we use the elapsed instruction counters in the current Intel processors [Int07]; similar counters can be used on other platforms. We can thus let the application run for a while without the signature generation/validation overhead associated with a checkpoint location. This is very important for long running loops in the code. A malicious user cannot exploit this fact since it is very unlikely for such a user to guess when a checkpoint location will be disabled and for how long it remains disabled. In the current prototype, specific checkpoints are temporarily disabled by the sys_checkpoint call in response to a request from the AS that is piggybacked to the validation message from the AS to the CS. A disabled checkpoint is re-enabled by the running process itself, based on the value of the elapsed instruction counter.

Figure 2. Percentage increase in the execution times of the selected benchmarks with the use of the proposed execution authentication mechanism for different values of M

Figure 3. Run times (in seconds) of benchmarks with and without execution authentication. Optimized authentication configurations from Table 1 are used.

We applied our approach to authenticate the remote execution of the well-known SPEC CPU2006 benchmarks [SPEC 06]. The results presented here do not show the complete set of benchmarks for the sake of brevity; instead we have selected a few benchmarks whose behavior is representative, vis-à-vis the performance overhead. The signature components used in the prototype implementation described here are addresses of the four most recently-executed branches, selected general register values, stall cycle count and the ID of the most recent system call.
We evaluated our scheme with the option of temporarily disabling specific, heavily-encountered checkpoints, as discussed above. We ran each benchmark 5 times and used the harmonic mean of the five runs to get the execution times reported here. Our compute server and authentication server are identical and have a 2.4 GHz Intel Core Duo processor with 2 Gbytes of RAM, connected via a 100 Gbits/sec. switched Ethernet.

Figure 2 depicts the percentage increase in the execution time with the use of our remote authentication mechanism against the base case – normal execution without any execution authentication. For the results shown in Figure 2, the value of N, the threshold count for disabling signature checking at a specific checkpoint was set at 1000 – that is, if a checkpoint was encountered 1000 times, signature checking was temporarily disabled at this checkpoint for the following M executed instructions. The values of M used, as shown in Figure 2, ranged from 100,000 (1E5) to 10⁹ (1E9). For some benchmarks, increasing M has the expected result – the overhead for execution authentication decreases. For benchmarks like gobmk and soplex, increasing M has little effect on the overhead. The soplex and gobmk benchmarks both have fewer instructions executed between checkpoints as the average basic block sizes are very small for these programs, resulting in a high authentication overhead. The only way to reduce the execution overhead in this case will be to overlap signature transmission and verification with execution and to use a different set of values for M and N. Another general approach for reducing the authentication overhead will be to use the notion of superblocks [Hwu+ 93] or dominator blocks to reduce the number of checkpoints and still retain a high degree of coverage of the expected control flow paths. In general, M and N need to be tuned independently for each benchmark and possibly for each checkpoint. Investigations along these lines are in progress.

Table 1 essentially presents a superset of the data depicted in Figure 2 and shows the absolute run times, the number of traps (calls) made to sys_checkpoint and the percentage increase in execution time with the proposed execution authentication mechanism enabled. Data for N=1000 and different values of M is presented in Table I. Figure 3 depicts the actual runtimes for the selected SPEC benchmarks – one for the benchmark as such without any execution authentication and the best execution time obtained from the data of Table I with the use of the proposed authentication mechanism.

As mentioned above, overlapping execution and the control flow signature verification can reduce the overhead of authenticating execution – we are currently implementing this and some other optimizations to further the overhead of authenticating execution.

9. Advantages of the DARE Framework

The DARE framework for validating remote execution directly addresses the limitations of existing techniques, as described in Section 2. The advantages of the DARE framework are as follows.

• DARE is a generic framework that can be customized to use a variety of signatures, permit users to take optional actions to change global information that form part of the signature after each validation, change the frequency of challenges and responses dynamically to tailor the tradeoff between performance and coverage.

• Automated, high-coverage checkpoint generation: Unlike other approaches, we can generate a set of checkpoints into a pool of checkpoint addresses automatically.

• Dynamic checkpoint selection and trapping: Dynamic checkpoint selection and dynamic breakpoint insertion allows us to overcome a fundamental security hole in existing solutions such as [BOY 03a, BOY 03b, HFS 98, KJ 03, XDC 04].

• Use of secure control flow signatures: To prevent the attacks like guessing the signature by using simulator like programs and/or replaying previous communication messages, we use the control flow specific properties to generate the signature of execution at each checkpoint.

• Localized performance impact: From the view of performance, while we can check the function call sequence including system call sequence for the monitored code, DARE affects only the target process/monitored code, unlike the techniques of globally intrusive mechanisms such as those of [XDC 04] and [HFS 98].

Transparency: A good security mechanism should be fully transparent to the user and also to the programmer. DARE is fully transparent to both in this respect. The programmer does not need to decide on the checkpoint list, function calls for monitoring; nor does the programmer need to worry about the configuration of the authentication entity.

10. Conclusions

This paper proposed a generic framework with a new approach for validating the remote execution of a program. DARE differs from existing approaches in its use of dynamic checkpoint selection and dynamic generation of signatures based on the control flow path and the side effects of execution. These signatures
are generated using the model specific registers used for CPU and system monitoring as found in most contemporary microprocessors. We described the implementation of a prototype system in Linux. We also proposed a technique for reducing the overhead of execution authentication on a program's run time by temporarily disabling signature generation at heavily encountered checkpoints.

An important consideration in the proposed execution mechanism has to do with the exact nature of the signature for control flow. In most modern desktop/laptop/server processors, the MSRs hold statistics of cache misses, accuracy of branch prediction, committed instruction count, TLB miss rates etc. We are investigating the use of these measures, along with explicit control flow information held in the MSRs (such as last few branches and system calls executed) to get generalized signatures of control flow that can be matched stochastically against a set of reference signatures. Such generalized signatures can possibly reduce the number of checkpoints needed and may well hold the promise of reducing the runtime overhead of the proposed scheme. Last but not the least; we have started investigations to formally prove that the proposed execution authentication mechanism is robust and correct.

Acknowledgements

This work is supported in part by the NSF (Award Nos. CNS 0454298) and by the Center of Computing Technologies at SUNY-Binghamton.

References

[CM 05] Simon P. Chung, Aloysius K. Mok, "On Random-Inspection-Based Intrusion Detection", in Proc. of the 8-th Int'l Symp. on Recent Advances in Intrusion Detection (RAID 2005).
[SLQP 07] Arvind Seshadri, Mark Luk, Ning Qu, Adrian Perrig, SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes, Symposium on Operating Systems Principles, 2007