Component-Oriented Monitoring of Binaries for Security

Raghavendra Rajkumar, Andrew Wang, Jason D. Hiser, Anh Nguyen-Tuong,
Jack W. Davidson, John C. Knight
151 Engineer's Way, Charlottesville VA 22904
Department of Computer Science, University of Virginia
{rr4ff|aaw6f|hiser|nguyen|davidson|knight}@virginia.edu

Abstract

Security monitoring systems operate typically at the process level. Various authors have indicated that monitoring at a finer level of granularity than the process is highly desirable. In this paper, we introduce COMB, a framework for imposing policies to confine the behavior of applications. Unlike previous approaches, our technique is applied per component (functions, libraries, and/or plugins) while requiring only the availability of the binary executable form of the program.

To demonstrate the feasibility of COMB, we report a case study on a real-world, representative program, the Firefox web browser. Two characteristics of Firefox permit possibly untrusted code to be executed. First, it provides an extensible architecture to allow third-party developers to extend its functionality, and second it makes use of more than 150 external libraries. Using a simple system-call monitoring policy applied to Firefox plugins, we show that COMB can provide protection with reasonable overhead.

1. Introduction

In this paper, we present COMB (Component-Oriented Monitoring of Binaries), a framework that enables fine-grained monitoring and security-policy enforcement of binary programs. Existing methods of confining the behavior of binary programs typically operate at the process level or system level, by monitoring and regulating the interactions between processes and the underlying operating system [2][23][27].

COMB provides the ability to divide a single process into *components*, and then monitor and enforce distinct policies on components *individually*. COMB defines a component as the code within specified address ranges. Defining components by address ranges provides the flexibility to include various notions of functional abstraction, including functionality encoded in static libraries, shared libraries and plugins. The policies that COMB enforces include those associated with sequences of actions, including sequences involving multiple components.

In typical applications, process-level monitoring forces identical monitoring policies to be applied to the entire program. We refer to such monitoring techniques as *coarse-grained*. Several authors have observed that *fine-grained* monitoring, as provided by COMB, would improve the accuracy of many security techniques [7][13][16][17][24].

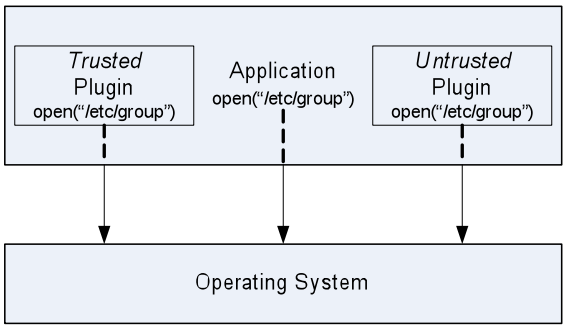

Figure 1. System calls originating from untrusted plugins are indistinguishable from calls from trusted sources

As an example of the utility of fine-grained monitoring, consider the increasing use of *extensible* applications. From the perspective of the operating system, extensions—commonly referred to as *plugins*—are indistinguishable from the hosting application unless they operate as independent processes. Several widely used applications, including Mozilla Firefox and the Apache web server, utilize plugin architectures with plugins operating in the same process as the hosting application. Such applications are extended, often by users, using plugins from different sources with different levels of trust. In many cases, an action that is normal for one plugin or the hosting application might be considered malicious for another. For example (see Figure 1), opening critical files such as "/etc/group" might be required by the hosting application and acceptable for a plugin from a

trusted source but problematic for certain files or certain plugins.

Browsers in which plugins run as individual processes provide finer granularity. Several such browsers already exist, e.g. Google Chrome [1] and Microsoft Gazelle [29], and others are under active development, e.g. Mozilla Electrolysis [14]. We conjecture that over the long-term even a process-per-plugin model would not be sufficiently fine-grained. For example, a plugin might itself be composed of external libraries and so the unit of confinement desired may need to be refined even further. In addition, the existing large installed base of extensible monolithic applications, such as Internet Explorer, Firefox, Apache and various media players, is unlikely to be retired in the near future.

To evaluate COMB, we have built a prototype implementation. Our prototype operates on IA-32, ELF program binaries and does not require access to source code or debugging information. As a case study in this paper, we describe the application of COMB to the Firefox browser using only the downloaded binary form. As an example of a simple COMB policy, we demonstrate how system-call-based policies can be applied to individual Firefox plugins. Our threat model is that of benign plugins that might contain vulnerabilities through which attackers can attempt to gain control of the system. According to Symantec, there were over 400 such vulnerabilities identified in browser plugins in 2008 [26].

To summarize, the major contributions of this paper are:

- The development of a general and flexible framework for monitoring, modifying and restricting the behavior of components such as dynamically-loaded libraries or plugins.
- The ability to do so using only the binary executables of programs.
- A working prototype to establish the feasibility of the COMB framework using system-call monitoring and prevention as a sample policy.

This paper is organized as follows. Section 2, presents the architecture of COMB, and section 3 discusses the potential approaches to exploiting fine-grained monitoring in security applications. Section 4 summarizes our prototype implementation, and section 5 describes a case study based on Firefox and presents performance results based on the SPEC2000 benchmarks. Section 6 discusses related work, and finally section 7 presents our conclusions.

2. Component-Oriented Monitoring

COMB extends traditional monitoring techniques by exposing to the system's policy enforcement mechanism additional information about an action being taken. We refer to this information as the *context* of the action. Traditional policies either allow or deny actions based upon the operation and parameters of the action. Expressed as a function, this monitoring is:

```
seq (operation × args)
            → {allow, deny}
```

COMB policies take the form:

```
seq (operation × args × context)
            → {allow, deny, response}
```

where `response` is a response function that can replace the original action with recovery code.

Although other approaches are possible, our implementation of COMB relies upon four architectural techniques: (1) applications are executed using *software dynamic translation*; (2) a *component memory map* is maintained during execution; (3) a *shadow stack* is maintained for each thread during execution; and (4) *policy recognizers* operate with data from the monitoring mechanism to detect and respond to policy violations.

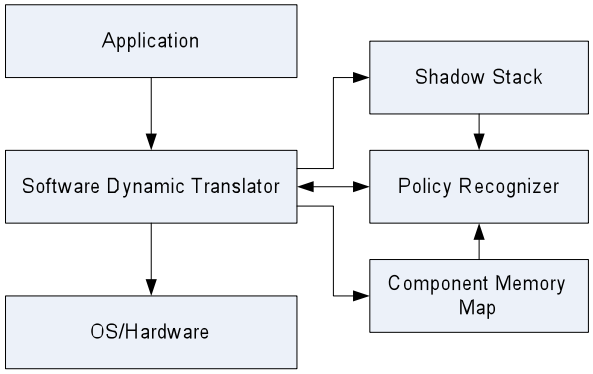

Figure 2. Overall architecture of COMB

The overall architecture of COMB is shown in Figure 2, and the basic principle of operation is as follows. Prior to (for static items) and during (for dynamic items) execution of the application, the component memory map is built and updated to reflect the locations of items loaded into memory. The entries that are included in the map are determined by the components for which fine-grain monitoring is desired.

During execution, the shadow stack is maintained to provide a complete and consistent structure of return addresses for function calls. Such information provides the necessary context to express COMB policies.

The software dynamic translator (SDT) enables software malleability and adaptivity at the instruction level by providing run-time monitoring and code modification capabilities. SDT can affect an executing program by injecting new code, modifying existing code, or controlling the execution of the program in some way. These capabilities give system designers unprecedented flexibility to control and modify a program's execution [12][14][21][22]. In Section 4, we provide an overview of Strata, the software dynamic translator used to implement COMB.

For COMB, we use the SDT to examine each instruction prior to the instruction's execution. This examination allows the translator to detect the binary structures corresponding to the actions of interest prior to the actions being taken by the program. Many actions such as system calls are unlikely to be directly located in the responsible components, because the actual action is effected by a library function. Thus, the component responsible for the action is determined by examining the shadow stack looking for a return address that is located within a component of interest. The details of the component are then obtained from the component memory map.

Once the component responsible for the action is identified, the triple—action, arguments, context—is submitted to one or more policy recognizers. If a policy violation is detected, the response is applied by the binary translator. We envision a wide variety of responses that might be appropriate depending on the knowledge that the policy author has about the component. A partial list of responses includes:

- Terminate the application. Although this approach is very "brute force," it may be the only safe option in some cases.
- Restrict the component's privileges by replacing the offending action with a virtualized error, such as a returning "File Not Found" when attempting to open a file that should not be opened.
- Log the offending policy violation for off-line auditing.
- Throttle the component's progress by inserting a pre-defined delay. This action would be useful if the malicious activity was to consume excessive memory, network, or disk bandwidth causing denial-of-service for the remainder of the machine.
- Raise an alarm to a remote machine or human indicating that a possible security breach may be underway.
- Increase the monitoring level of the application to thwart potential exploits via heavier-weight instrumentation. Such an approach would be useful for individual components with well-known APIs. If the API is being misused, a heavy-weight detector may be useful for determining if the misuse is intentional and benign due to a new version or restriction to the API, or malicious and might cause a memory overwrite or other security violation.
- Restrict execution of the component to a set of deemed safe functions (which may be no functions at all). For future requests to non-safe functions, return a virtualized error code.

As a simple example of the use of COMB, consider an attempt by an untrusted plugin to make a prohibited system call. The binary translator detects the system call in the binary program and identifies the associated parameters. Using the shadow stack and the component memory map, the component initiating the call is identified and the simple policy of denying the system call is invoked. The desired response of deleting the call and returning an error code is then effected by the binary translator.

3. Exploiting Fine Grained Monitoring

The fine-grained monitoring capabilities of the COMB framework coupled with the use of software dynamic translation opens up a rich design space for designing and enforcing security policies. The potential benefits are enhanced by two aspects of the fine-grained monitoring framework: (1) monitoring operates on binary programs with no additional information required; and (2) monitoring is effected before an action is taken by the program thereby preventing damage from the action and permitting response actions to be taken. We highlight several such policies:

- **Prevention of Excess Resource Consumption**. Previous work has demonstrated the viability of software dynamic translation for enforcing policies to limit network bandwidth at the process level using system call interposition [21]. COMB polices are more expressive in that they can enforce restrictions at the level of components.
- **API Restriction**. Application programs rely on external libraries to implement functionality. These libraries typically define an API that imposes requirements on call sequences or parameter values that programmers must conform to. Libraries often fail to verify compliance, either accidentally or intentionally for efficiency or ease-of-implementation purposes. Example requirements include failure to release allocated memory blocks, attempts to release allocated blocks more than once, and failure to close open files. In the context

of extensible applications such as Firefox, the API regulates the lifecycle of extensions with respect to the hosting application as well as providing bi-directional support services to handle URLs, streams, memory, drawing and events [18]. With the ability to monitor events at the granularity of function calls, fine-grained monitoring enables policies that check for compliance with the API.

- **Blocking System Calls with Context**. COMB policies can take into account both the arguments to a system call and the context information provided by the component details. This precision enables the expression of policies to confine the behavior of intra-process components. Furthermore, the use of software dynamic translation enables a richer response than simply shutting down a program. For example, error virtualization techniques that return an error code (as if the error originated from the operating system) [25] are easily implemented and tailored for each system call.
- **High-Level Behavioral Policies.** Though not part of the current COMB implementation, a promising venue to explore is to enable policies at higher semantic levels. An example would be the ability to write policies explicitly against the fact that "program or component X is rendering an image, playing a movie, or printing a file".
- **Component Shutdown.** In response to a failed component, a policy response might specify that the component should be shut down and how to achieve that goal. Many applications can work acceptably with one or more missing components. For example, the Firefox web browser can render most web pages without the Adobe Reader plugin. Likewise, a word processing program could function without the dictionary component needed to provide spell-checking support. If the component violates its policy (say, by trying to execute an `execve` system call), the response mechanism could be to shut down that component. The safest response allows no more instructions from the component's address range to execute. In the case that some functions are necessary for the proper shutdown procedure to complete, those functions could be individually specified in the component. For other functions, an error-virtualization technique could be used to elide execution when the functionality is requested. In both the web browser plugin case and the word processor dictionary component case, gracefully degrading the performance of the application by removing only the compromised component allows the user to continue work until their data can be written to permanent storage, and the program can be restarted in an uncompromised state.

In general, applying fine-grained policies to commonly used libraries and widely-deployed extensible platforms such as browsers and web servers would result in reusable policies that could easily be shared across multiple applications. Again, since COMB operates at the binary level, such universal policies could be put into effect even for programs and libraries for which the source code is unavailable.

4. Implementation

In this section, we summarize our prototype implementation of COMB. Other implementation approaches are possible.

The application is executed using software dynamic translation. In general, software dynamic translation can affect an executing program by injecting new code, modifying existing code, or controlling the execution of the program in prescribed ways. Examples of software dynamic translation systems include Strata [22], Pin [14], and Dynamo-RIO [12]. We chose Strata for our prototype.

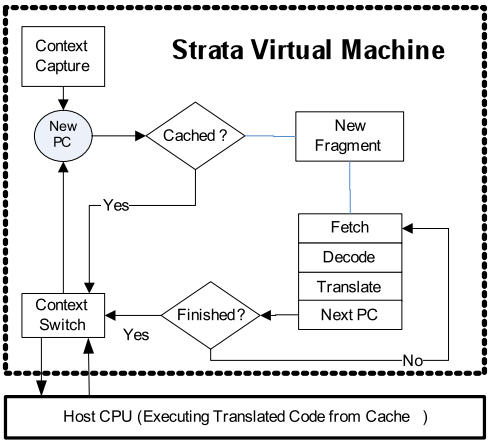

Figure 3. Strata Virtual Machine

The Strata Virtual Machine mediates application execution by examining and translating instructions before they execute on the host processor. Blocks of translated instructions called fragments are held in a Strata-managed cache, called the fragment cache. Strata is implemented as a co-routine resident with the application and is entered by capturing and saving the application context (e.g., program counter, condition codes, registers, etc.). Following context capture, Strata processes the next application instruction fragment. If a translation for this fragment has been cached, a context switch restores the application context and begins executing cached translated instructions on the host

machine. Performance assessment of optimized versions of Strata indicates that it imposes modest overhead on the application [10][22], and this is one of the reasons why we chose Strata for our prototype.

The component memory map (CMM) maps virtual address ranges to components. The CMM is implemented as a splay tree. To populate the CMM, the application binary code is instrumented using Strata's introspection capabilities to monitor `open`, `mmap` and `close` system calls, so as to detect the dynamic loading of libraries. This is illustrated in Figure 4.

A global table that maps file descriptors to filenames is maintained, and updated whenever an `open` or `close` system call is made. The `mmap` system call loads a file into memory, given its file descriptor. The mapping from an address range to a file descriptor yields a mapping from an address range to a label when composed with the file descriptor table. The resultant mapping is stored in the CMM.

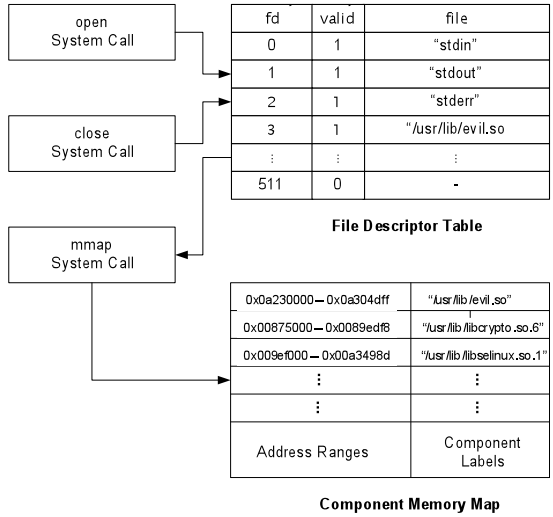

Figure 4. Creation of the Component Memory Map

The CMM can contain multiple entries for a single dynamically loaded library. This occurs because functions may be loaded lazily, where calls to multiple functions in a dynamic library correspond to multiple loads. Moreover, the two entries might not be contiguous if the two function calls are separated by other dynamic loads.

To maintain the shadow stack, our implementation instruments call/return instructions to push/pop the return address. As an optional feature, our prototype verifies that the address of a return matches the top of the shadow stack to detect various stack-corrupting attacks.

Figure 5 shows how the component memory map and the shadow stack are used for fine-grained monitoring. When a system call is made, the shadow stack is first composed with the CMM to produce a new structure, the *component stack*. The component stack is identical to the shadow stack but with component names replacing addresses. The detailed content in Figure 5 comes from the Firefox case study described in the next section.

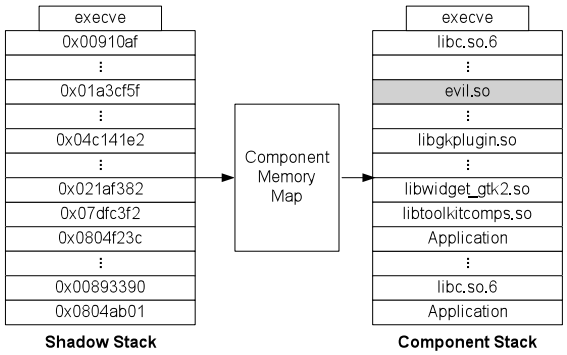

Figure 5. Creation of the Component Stack from the Shadow Stack

The component stack is examined to verify that each component in the stack is permitted to make the desired action. The entire stack is examined, because a malicious action might be initiated by a component lower in the stack that uses other benign components to achieve its goals. In general, arbitrary complex policies may be expressed but we use a relatively simple policy for illustrative purposes. In the example in Figure 5, `libc.so.6` is allowed to perform system calls, because `libc.so.6` is the standard interface for making system calls. `evil.so` (a malicious component) does not have permission to make the `execve` system call. Thus the system call is not permitted and a response action is initiated.

To build on this scenario, we provide sample pseudo-code (Listing 1) for a more sophisticated policy in which, upon detection of `execve`, we turn off all system calls for the plugin. On lines 14-16, we turn on monitoring for `execve` for `evil.so` and register a policy handler. Lines 1-13 illustrate this handler. Strata will automatically invoke the handler with the system call number and associated arguments. On line 3, we search for `evil.so` in the component stack. If it is not present, we allow the system call to proceed normally (line 11). However, if `evil.so` is on the stack, then it is performing a malicious action. We then turn off all system calls for this plugin by registering the `disallow_policy` handler (lines 5-6), and return an appropriate error code for the policy using (line 7).

```
(1)  int execve_policy(int sysNum, const char*
     filename, char *const argv[], char *const
     envp[]) {
(2)    // is the component on the stack?
(3)    if (componentStack.lookup("evil.so"))
(4)    {
(5)      unregister("evil.so", ALL_SYSCALLS);
(6)      register("evil.so", ALL_SYS_CALLS,
           disallow_policy);
(7)      return ERROR_VIRTUALIZE(sysNum);
(8)    }
(9)    else
(10)   {
(11)     return ALLOW_CALL;
(12)   }
(13) }

(14) void init_policy() {
(15)   register("evil.so", SYS_execve,
         execve_policy);
(16) }

(17) void disallow_policy(int sysNum) {
(18)   return ERROR_VIRTUALIZE(sysNum);
(19) }
```

Listing 1. Pseudo-code for implementing COMB policies

5. Evaluation

As a case study of our fine-grained monitoring framework, we applied the framework to the Mozilla Firefox browser (version 2.0). The goals of the case study were to determine the feasibility, efficacy, and overall performance of the system on a significant application for which we were not the authors.

A variety of vulnerabilities have been documented for existing Firefox plugins. For purposes of our case study, we chose not to use these plugins because the associated vulnerabilities provide neither a comprehensive nor a controllable set of samples. Instead, we developed a custom plugin (`evil.so`) that was designed to have controlled and predictable behavior. We seeded a variety of exploits into this custom plugin so that invoking the plugin had the effect of an adversary exploiting a vulnerability in a Firefox plugin.

Firefox is a large, real-world program that uses a large number of shared libraries. As part of this case study, we detected 151 shared libraries being loaded during a single execution. Approximately two-thirds of these libraries were system libraries and the rest were Firefox-specific libraries. The large number of third-party libraries poses a threat as a source of potentially exploitable vulnerabilities. Figure 5 shows a snippet of the shadow and component stacks for Firefox when an `execve` system call is made by the `evil.so` plugin.

The shadow stack actually contains 52 entries that correspond to 5 dynamically loaded libraries (not including pre-linked libraries). Note that the component shadow stack contains duplicate entries (not shown in the Figure except for `libc.so.6`) as intra-library calls are made.

Case Study Methodology

The protocol we followed with this case study was:

1) The Firefox browser binary was downloaded from the distribution site.
2) The custom plugin was developed and installed to operate with the prepared version of Firefox.
3) We developed fake malicious web pages and verified that the seeded exploits were triggered when COMB is not activated
4) The software dynamic translator was installed in the binary.
5) The browser was used to view a variety of randomly selected web sites, including sites that make heave use of JavaScript such as Google Local.
6) The browser was used to view a set of custom-developed web sites that were designed to invoke the custom plugin and cause the associated undesirable behavior.

Case Study Results

Firefox is a large and complex program. It is also representative of the type of application for which fine-grained monitoring can provide significant value. We did not test Firefox (in the sense of comprehensive software testing) operating with COMB in a systematic manner, but our use of the browser to view a variety of web pages illustrates the feasibility of applying fine-grained monitoring to a program of this complexity.

Malicious actions that could be triggered using our custom plugin include the following seeded system calls: `execve`, `mprotect`, `chmod`, `creat`, `mkdir`, `setuid` and `unlink`. These were chosen as a sample subset of system calls deemed security-critical by other researchers [17]. However, we note that COMB can easily be configured to monitor any system call. The custom plugin does not need to make any of these calls for its normal operation. Using our prepared web pages that invoked the plugin, each of these calls was made to occur in various ways with various parameters.

The results obtained from these exploits demonstrated the efficacy of the approach. All of the synthetic exploits were detected prior to the associated

system call being made. There were no false positives when viewing other web pages that did not result in the plugin being invoked. Although we included only default recovery actions, i.e. returning an error code to indicate the system call had failed, a wide variety of responses to attack are possible.

Measurement of the time and space performance of a program like Firefox is problematic since the metric of interest is not well defined. The issue with a major application is utility because there are many factors (network load, server loads, client machine capabilities, etc.) that impact the performance as perceived by the user. We assessed the overall performance of the monitored version of Firefox merely by asking users whether the interactive performance was deemed adequate. In all cases it was. Concrete measurement of such overheads is presented in the next section.

5.1. Performance

To supplement the subjective assessment discussed in the previous section, we conducted a benchmark assessment of the overhead imposed by the COMB monitoring framework on the C/C++ subset of the SPEC2000 benchmarks.

Performance on these benchmarks is shown in Figure 6. All experimental data were measured on a 2.66GHz quad-core Mac Pro with 8GB of main memory. The benchmarks were run on top of Fedora Core 8 running inside of VMWare Fusion.

The first bar shows the performance of the Strata SDT system alone, normalized to native execution (i.e., no monitoring and no SDT). The second bar shows performance with COMB monitoring (using a simple policy to disallow `mprotect` calls from the standard C library.) As the figure shows, average overhead is 31% and 44% for the two configurations. The difference of 13% between the two configurations measures the cost of implementing the component memory map and monitoring policies above and beyond a standard configuration of the Strata virtual machine.

Many benchmarks (`164.gzip`, `175.vpr`, `181.mcf`, `256.bzip2`, `300.twolf`, `177.mesa`, `179.art`, `178.equake`, and `188.ammp`) show little or no slowdown. These are the benchmarks with relatively fewer indirect branches and call/return pairs. We see the benchmarks (`252.eon`, `254.gap`, and `255.vortex`) with the most call/returns have the most additional overhead with COMB monitoring.

Although non-trivial, our unoptimized prototype numbers indicate that such a system could be used for a wide variety of applications, including interactive programs (such as Firefox), and high-security or low-bandwidth servers. Further, many optimizations are possible. For instance, SDT overhead can be reduced [8][9][10]. Another example would be to optimize the shadow stack to only track components, i.e. a component shadow stack, so that COMB could more cheaply determine which component is responsible for the behavior.

5.2. Security of COMB

For COMB to protect an application from attacks, it must protect itself. If it fails to do so, attacks will instead use a vulnerability in the application to attack the monitor program and subvert its protections.

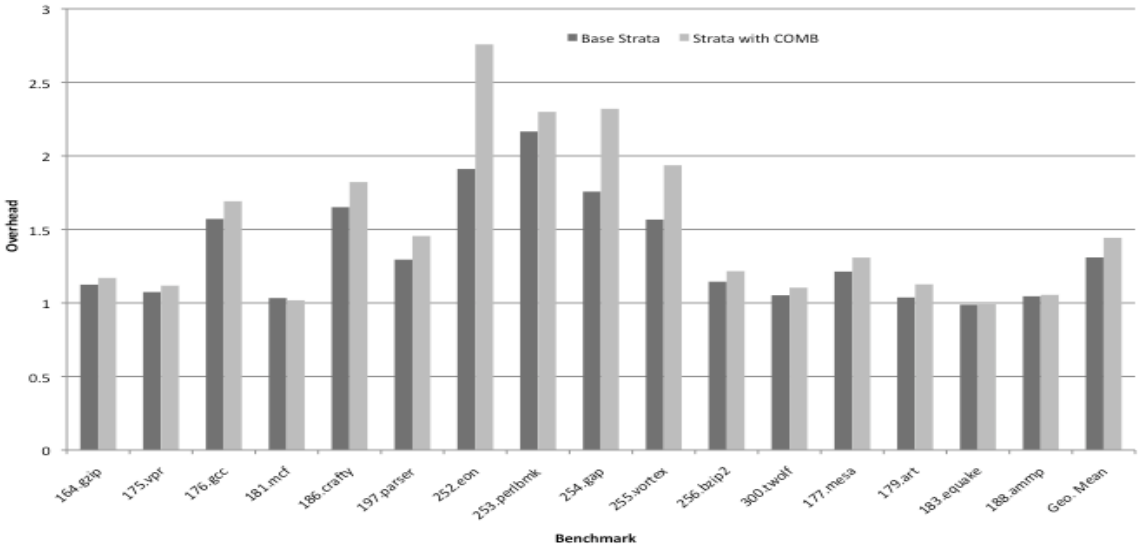

Figure 6. SPEC Benchmark. Performance is normalized to native execution.

In our experimental prototype, we have not implemented any protections. We do feel however that thorough protection would be easy to implement. To protect the SDT's code and data structures, the SDT can use hardware page protection mechanism to prevent any application instructions from reading or writing the SDT's data structures. The SDT has to carefully manage transitions between SDT code and non-SDT code to turn on and off page permissions during the switch. This mechanism has been shown to be inexpensive in previous work [10].

Per-page protections can be used for infrequent operations, such as adding to the component memory map. However, frequent operations like pushing or popping from a shadow stack require a much lighter-weight protection mechanism for efficiency. To protect the shadow stack, the best protection mechanism would be to use an i386 segment register to access memory that would normally be unavailable to the application. Although such a mechanism might cause issues for applications which use the non-standard segment register, such applications are very rare, and it would be easy to detect and provide comprehensive (and consequently more expensive) monitoring for these applications.

6. Related Work

Anomaly Detection. Early pioneering work on system call monitoring was reported by Forrest et al. in 1996 [5]. That work established system call sequences as an accurate depiction of program behavior [6]. When the sequence-based approaches began to falter against mimicry and other attacks [28][3][19], the model was extended to further monitor data flow of the program by examining system call arguments [13][16]. Additional sources of information such as execution context, return addresses and library calls have been useful in increasing detection rates while decreasing false positive rates [4][7][11][17][24]. Execution context information has long been used to model a program's control flow, but we add a further level of semantic detail through mapping addresses to logical program components with our virtual memory map. This additional level of information enables us to write precise, fine-grained policies that further compartmentalize functionality while also taking advantage of qualitative judgments of quality and trust.

Mandatory access control. SELinux [23], AppArmor [2], and Tomoyo [27] are examples of mandatory access control frameworks, whose objective is to enforce restrictions on access to system objects. As these systems reside at the OS/Process boundary, they do not make use of fine-grained context information.

Modular browser architectures. Browsers are moving towards more modular architectures with the goal of isolating high-risk components from the browser kernel. The best examples of this are Google Chrome [1][20] and Microsoft's experimental Gazelle web browser [29], which sandbox tab rendering into separate processes. Recent versions of Internet Explorer implement a similar type of privilege separation [30], while Mozilla has been experimenting with a Chrome-like architecture with their Electrolysis project [15].

However, even after logical division of the application into separate processes, there are limits to the granularity of process-level monitoring. Our approach complements these new browser architectures, and through our fine-grained monitoring methodology can subdivide these sandboxed components yet further. The increased precision allows for more expressive policies and better enforcement.

7. Conclusions

We have presented COMB, a system for fine-grained monitoring of actions taken by applications to permit security policies to be applied at an intra-process, component level. COMB's implementation is based upon software dynamic translation, a shadow stack mechanism, a map of memory addresses to components, and a set of security policy recognizers.

The user experience of our initial prototype on the Firefox browser, a complex real-world application, is encouraging. On a selected subset of the SPEC benchmark suite, we observed overheads of 44% in absolute terms or 13% overhead above and beyond the overhead incurred by running within a software dynamic translation (SDT) environment. We expect to be able to reduce this overhead considerably based on our previous experience in optimizing SDT systems.

COMB provides a framework within which the problem of securing component-based systems can be addressed. Where core applications and add-on components are acquired from sources with different levels of trust, COMB allows the monitoring of the core application and the individual components to differ and also to be changed over time should trust in a component change over time.

COMB provides a basic capability that can be used in a variety of contexts to improve the accuracy and resolution of techniques developed by others including intrusion detection, application sandboxing, etc. In future work, we will leverage the COMB framework to investigate the feasibility and efficacy of such techniques.

8. Acknowledgements

This research was supported in part by the National Science Foundation under grant CNS-0716478, the Army's Research Office under grant W911NF-10-1-0131, and by the Department of Defense under grant FA9550-07-1-0532. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring agencies.

9. References

[1] Barth, A., C. Jackson, C. Reis, and The Google Chrome Team, The Security Architecture of the Chromium Browser, Stanford Security Laboratory, 2008.

[2] Bauer, M., Paranoid Penguin – An Introduction to Novell AppArmor, Linux Journal, Jul 1, 2006. http://www.linuxjournal.com/article/9036

[3] Chen,, S. J. Xu, and X. C. Sezer, "Non-control-data attacks are realistic threats", In 14th Annual Usenix Security Symposium, Aug 2005.

[4] Feng, H. H., O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly Detection Using Call Stack Information", IEEE Symposium on Security and Privacy, page 62, 2003.

[5] Forrest, S., S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for Unix Processes", IEEE Symposium on Security and Privacy, pp. 120-128, Oakland, CA, May 1996.

[6] Forrest, S., S. A. Hofmeyr, A. Somayaji, "The Evolution of System-Call Monitoring", 2008 Annual Computer Security Applications Conference (ACSAC '08), 2008.

[7] Gao, D., M. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection", ACM Conference on Computer and Communications Security, pp. 318-329, October 2004.

[8] Hiser, J. D., D. Williams, A. Filipi, J. W. Davidson, B. R. Childers, "Evaluating fragment construction policies for SDT systems", Intl. Conf. on Virtual Execution Environments, 2006.

[9] Hiser, J. D., D. Williams, W. Hu, J. W. Davidson, J. Mars, B. Childer, "Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems", Intl. Symposium on Code Generation and Optimization, 2007.

[10] Hu, W., J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill, "Secure and Practical Defense Against Code-injection Attacks", Virtual Execution Environments, 2006.

[11] Jones, A., Y. Lin, "Application Intrusion Detection using Language Library Calls", *Annual Computer Security Applications Conference*, 2001.

[12] Kiriansky, V., D. Bruening, and S. Amarasinghe, "Secure Execution Via Program Shepherding", 11th USENIX Security Symposium, August 2002.

[13] Kruegel, C., D. Mutz, W. Robertson, and F. Valeur, "Bayesian Event Classification for Intrusion Detection", Proceedings of the 19th Annual Computer Security Applications Conference, 2003.

[14] Luk, C., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation", ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, Illinois, USA., pp. 191-200, June 2005

[15] Mozilla, "Content Processes," 2009. https://wiki.mozilla.org/Content_Processes

[16] Mutz, D., F. Valeur, G. Vigna, C. Kruegel, "Anomalous System Call Detection", ACM Transactions on Information and System Security (TISSEC), pp. 61-93, Feb 2006..

[17] Mutz, D., W. Robertson, G. Vigna, and R. Kemmerer, "Exploiting Execution Context for the Detection of Anomalous System Calls", Recent Advances in Intrusion Detection, 2007.

[18] Mozilla, "Gecko Plugin API Reference", https://developer.mozilla.org/en/Gecko_Plugin_API_Reference

[19] Parampalli, C., R. Sekar, and R. Johnson., "A Practical Mimicry Attack Against Powerful System-call Monitors", 2008 ACM Symposium on Information, Computer and Communications Security, pp. 156-167, New York, NY, USA, 2008. ACM.

[20] Reis, C, A. Barth, and C. Pizano, "Browser Security: Lessons from Google Chrome", Communications of the ACM, Aug 2009.

[21] Scott, K., and J. W. Davidson, "Safe Virtual Execution using Software Dynamic Translation", 18th Annual Computer Security Applications Conference, Washington DC, 2002.

[22] Scott, K., N. Kumar, S. Velusamy, B. Childers, J. Davidson, M. L. Soffa, "Retargetable and Reconfigurable Software Dynamic Translation", Intl. Symposium on Code Generation and Optimization, March 2003.

[23] Security-Enhanced Linux. Jan 15, 2009. http://www.nsa.gov/research/selinux/

[24] Sekar, R., M. Bendre, P. Bollineni, and D. Dhurjati, "A Fast Automaton-based Method for Detecting Anomalous Program Behaviors", 2001 IEEE Symposium on Security and Privacy, 2001.

[25] Sidiroglou, S., Locasto, M. E., Boyd, S. W., and Keromytis, A. D., “Building a Reactive Immune System for Software Services”, USENIX Annual Technical Conference, pp. 149-161, April 2005.

[26] Symantec Corporation, “Trends for 2008”, Symantec Global Internet Security Threat Report, Volume XiV, A pril 2009. http://www.symantec.com/connect/sites/default/files/b-whitepaper_internet_security_threat_report_xiv_04-2009.en-us.pdf

[27] TOMOYO Linux: Behavior oriented system analyzer and protector. Dec 2009. http://tomoyo.sourceforge.jp/

[28] Wagner, D. and D. Dean, “Intrusion Detection via Static Analysis”, Proceedings of the IEEE Symposium on Security and Privacy, Oackland, CA, May 2001.

[29] Wang, H. J., C. Grier, A. Moshchuk, S. King, P. Choudhury, and H. Venter, “The Multi-Principal OS Construction of the Gazelle Web Browser”, Microsoft, Feb 2009.

[30] Zielgler. A., “IE8 and Loosely-Coupled IE (LCIE)”, 2008. http://blogs.msdn.com/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx