Adaptive Defenses for Commodity Software through Virtual Application Partitioning

Dimitris Geneiatakis Georgios Portokalidis Vasileios P. Kemerlis Angelos D. Keromytis

Columbia University
Department of Computer Science
New York, NY, USA
{dgen, porto, vpk, angelos}@cs.columbia.edu

ABSTRACT

Applications can be logically separated to parts that face different types of threats, or suffer dissimilar exposure to a particular threat because of external events or innate properties of the software. Based on this observation, we propose the *virtual partitioning* of applications that will allow the selective and targeted application of those protection mechanisms that are most needed on each partition, or manage an application's attack surface by protecting the most exposed partition. We demonstrate the value of our scheme by introducing a methodology to automatically partition software, based on the intrinsic property of user authentication. Our approach is able to automatically determine the point where users authenticate, without access to source code. At runtime, we employ a monitor that utilizes the identified authentication points, as well as events like accessing specific files, to partition execution and adapt defenses by switching between protection mechanisms of varied intensity, such as dynamic taint analysis and instruction-set randomization. We evaluate our approach using seven well-known network applications, including the MySQL database server. Our results indicate that our methodology can accurately discover authentication points. Furthermore, we show that using virtual partitioning to apply costly protection mechanisms can reduce performance overhead by up to 5x, depending on the nature of the application.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Monitors; D.4.6 [**Security and Protection**]: Information flow controls

General Terms

Performance, Reliability, Security

Keywords

Application partitioning, adaptive defenses, risk management, authentication, dynamic taint analysis, instruction-set randomization, information flow tracking

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

1. INTRODUCTION

Software faces a multitude of threats that enable attackers to execute arbitrary code through code-injection and code-reuse attacks [15, 35, 42, 49], bypass security mechanisms like user authentication [7, 9], and exfiltrate sensitive data [12, 19]. Oftentimes, attacks are enabled by bad system design and configuration errors [8, 45, 48], or even originate from otherwise "trusted" users [18, 47]. A plethora of defensive mechanisms that mitigate such threats have been proposed in the past [1, 2, 6, 28, 32, 54, 55], but very few of them have seen broad adoption [34].

Two of the largest obstacles in the adoption of these security defenses are the requirement for source code by approaches that are applied at compile time [2, 32] and the significant performance overhead imposed by solutions operating solely on binaries [28, 55]. Techniques that are both lightweight and require no software recompilation have been also proposed [1, 31], but they often address a narrower set of threats. Furthermore, many of these techniques are not orthogonal to each other and cannot be easily integrated. That is, even when they can be combined, their cumulative overhead becomes prohibitive.

This paper builds on the observation that *software does not have uniform security requirements throughout its execution.* In particular, different parts or components of an application are usually targeted by different types of attacks, or suffer dissimilar exposure to a specific threat, because of external events or innate properties of the software. For instance, protecting a service against sensitive data leaks is only relevant after first reading the data. Similarly, consider an FTP server that serves both authorized and public users. In the latter case, the server is intrinsically more exposed to exploits because more of its functionality is accessible by everyone. In both examples, execution is partitioned to segments with different security requirements. If we can identify these partitions, we can apply diverse protection mechanisms as and when needed to control their exposure to different types of threats, while avoiding cumulative overheads.

We propose *virtually partitioning* the execution of applications, and adapting the defenses being deployed based on the executing partition. The benefits of using adaptive defenses are twofold. First, it enables us to apply multiple protection mechanisms selectively, disabling mechanisms that are not relevant, or of a high priority, in favor of deploying more appropriate ones. Second, it provides a risk management mechanism that is orthogonal to existing protection schemes. In essence, virtual partitioning provides a tunable knob that controls the intensity of the defenses being applied in exchange for more resources (*e.g.,* CPU cycles or memory).

Our work focuses on automatically identifying how intrinsic properties of the software partition it to segments with disparate security requirements. We postulate that using events, such as reading sensitive data from a database (DB), for separating an application into different partitions is straightforward and can in fact be implemented by intercepting process-operating system (OS) interactions. We support our claim by implementing a runtime environment that can partition applications based on such events, and apply information flow tracking to monitor sensitive information.

However, our main goal is to describe a more generic and flexible framework for virtually partitioning applications. We present a methodology for automatically determining how and where user authentication splits application execution to partitions with asymmetric exposure to attacks, by dynamically profiling binaries at runtime and without the need for source code, debugging symbols, or any other information about the application at hand. We have developed a runtime environment that uses the virtual partitions to dynamically adapt the defensive mechanisms applied on binaries at runtime. We reuse existing schemes, such as dynamic taint analysis (DTA) [20] and instruction-set randomization (ISR) [36], and apply them selectively on the pre- and post-authentication parts of the application.

Others works [4, 37, 40, 51] have also focused on shrinking the attack surface of applications by reducing the parts that are exposed to attack, and isolating the most vulnerable parts, using techniques like sandboxing and privilege separation. However, most of these schemes are bound to specific applications that need to be *a priori* designed to work that way, or require access to source code.

To the best of our knowledge, this is the first work on virtual partitioning. We applied our approach on well-known server applications utilizing their built-in authentication mechanisms, as well as commonly used frameworks such as Pluggable Authentication Modules (PAM) [44]. Our findings demonstrate that we can automatically identify their authentication points with accuracy. We evaluate our runtime to verify its ability to dynamically switch between different protection mechanisms, and determine the performance benefits that can be gained by reducing the intensity of defenses after user authentication. For this purpose, we utilized DTA and ISR on the pre- and post-authentication partitions respectively.

Results show that we can greatly reduce the user-observable overhead of DTA by up to *5x*, by replacing DTA with ISR after the user successfully authenticates. Although other configurations (*i.e.,* combinations of mechanisms) may not enjoy the same improvements in performance, virtual partitioning can still be used to adapt the applied defenses. The main contributions of this paper can be summarized in the following:

- We present a methodology for automatically detecting the authentication point of any given program. It operates on binary-only software and requires little, or no, supervision.
- We enable adaptive software defenses, using multiple protection mechanisms applicable at commodity software, offering a diverse and affordable "risk management" scheme, in contrast to always-on and attack-specific protection approaches that are frequently not adopted due to their overhead.
- We implemented a virtual partitioning runtime, which can be applied on existing well-known applications (*e.g.,* MySQL and OpenSSH).
- Our software is freely available. We believe this can facilitate experimentation with virtual partitioning using a variety of software properties and external events.

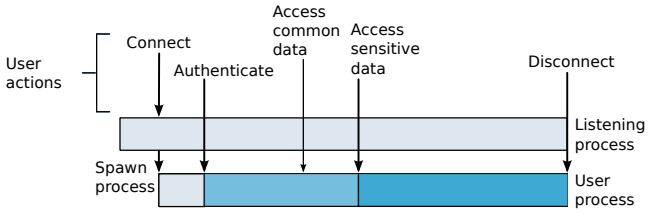


Figure 1: Virtual partitioning example: A user connects to an FTP server and performs various actions. The execution of the servicing process can be separated into different segments, based on the intrinsic properties of the server, like user authentication and accessing sensitive data.

The rest of the paper is structured as follows. In Section 2, we describe the main concept behind virtual partitioning. Section 3 elaborates on how user authentication can be used to automatically partition applications, and presents our methodology for determining the authentication points using binaries alone. In Section 4, we introduce our runtime environment that dynamically applies the partitioning and the various security mechanisms. We evaluate and discuss our approach in Sections 5 and 6 respectively. Section 7 surveys related work. Finally, we conclude this paper in Section 8.

2. VIRTUAL PARTITIONING

Virtual partitioning is based on the observation that applications, throughout their execution, face different types of threats or suffer dissimilar exposure to a particular threat, because of external events or innate properties of the software. This fact implicitly partitions their execution to segments with divergent security needs, which we can identify for diversifying the applied protection mechanisms, deploying what is needed, when it is needed, or use a multitude of security techniques without inflicting cumulative overheads.

Figure 1 depicts an FTP server, like Pure-FTPd, as it executes to handle a new user connection. When a user connects, the listening process spawns a new process to service the connection, and the user is called to authenticate. Authentication is an intrinsic characteristic of the FTP server as it partitions the execution of the user process to pre- and post-authentication parts. The pre-authentication part can be exercised by anyone who connects to the server. Exploiting a vulnerability in this part can enable a remote user to access the files of all FTP users and possibly gain administrator privileges [10, 14]. Even if the server drops administrator privileges and runs as the authenticated user immediately after authentication (*i.e.,* it is designed and implemented following the least privilege principle and privilege separation), protecting the pre-authentication part is crucial because it runs with elevated privileges. In fact, even if anyone could obtain a valid account on the FTP server (*e.g.,* as in the case of an FTP-based Dropbox [16] service), protecting the pre-authentication part protects the privileged part of the server that can access the data of all users.

Additionally, other external events can be also used to further partition the execution of an application to segments with varying security needs. As an example consider a scenario of accessing data of some importance on the server, which is also shown in Figure 1. In certain cases, it may be desirable to employ information flow tracking as soon as "sensitive" data has been accessed, to prevent their transmission over the network or monitor their use for future auditing [12, 19].

2.1 Benefits of Virtual Partitioning

The previous example aimed at demonstrating that even applications that have carefully implemented privilege separation can be partitioned to smaller segments with different security requirements. By identifying these partitions and *virtually* segmenting software execution, we can improve its security and provide a way to manage risks by applying diverse protection mechanisms on its partitions. We discuss the core benefits of virtual application partitioning below:

(a) **Manage the Attack Surface of Applications to Balance Risk and Performance.** We already discussed that services often suffer from vulnerabilities [9, 35, 49] that allow attackers to take control, by executing arbitrary code and/or bypassing authentication. At the same time, powerful protection mechanisms [28, 36, 55] that can be applied on any program, including commodity software, incur prohibitively high overheads, oftentimes exceeding 100%. For example, we can regard the unauthenticated partition of Pure-FTPd in Figure 1 as being more exposed to such attacks, since it usually runs with elevated privileges, and because users with valid credentials are considered "trusted". Note that this stands for services that authentication carries such a significance. Nevertheless, open systems where anybody can easily create an account (*e.g.,* Web services like Facebook and GMail), also suffer from similar threats.

In such systems, bypassing authentication could allow malicious users to subsume the identity of other users. Thus, in cases where software partitions face *asymmetric exposure to an attack*, partitioning enables us to apply a heavyweight protection mechanism on the most exposed part of a program, while we can use a more lightweight mechanism, or none at all, for the authenticated partition. In this fashion, we control how much to harden the security of different parts of a program, potentially incurring significantly less overhead when compared to securing it uniformly (see Section 5).

(b) **Apply Multiple Protection Mechanisms without Incurring Cumulative Overheads.** The two partitions of a service can have different primary security concerns. For example, after a user successfully authenticates with the FTP server, the process servicing him executes with reduced privileges. The primary concern now is no longer a remote attacker bypassing authentication or gaining administrator privileges, but threats such as data leakage and privilege escalation. In such cases, where the partitions face *exposure to different threats*, partitioning enables us to use different mechanisms to address them.

Instruction-set randomization (ISR) [36] or dynamic taint analysis (DTA) [28] could be used on the unauthenticated partition to protect against arbitrary code execution attacks, while information flow tracking (IFT) [57] could be used to protect from sensitive data leakage after a particular file is accessed by the user. This way, we can enable different security mechanisms based on what is needed the most by each partition, while not incurring cumulative overheads. Moreover, it enables us to apply mechanisms that could otherwise conflict with each other. For instance, both DTA and IFT require a data flow tracking framework. Utilizing such a framework to implement the mechanisms concurrently can be challenging and can also magnify their respective overheads.

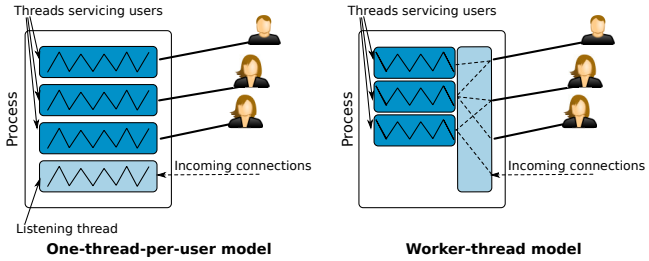


Figure 2: One-thread-per-user and worker-thread models for multi-threaded/-process servers. In the former, a different thread is assigned to service each user, while in the latter each user message can be processed by a different thread.

2.2 Applicability to Different Software Architectures

According to Welsh *et al.* [52], most Internet services are built by following the multi-threaded or multi-process server model, where multiple clients connect to a server for requesting a service. However, due to the increased hardware parallelism available today, because of the advent of multi-core CPUs, multi-threaded architectures are nowadays favored. This is predominantly done in the two ways shown in Figure 2.

One-thread-per-user model. Concurrent users are handled by spawning a new process or thread for servicing each user request. Many popular servers, such as Pure-FTPd, OpenSSH, Exim, SNMPd, MySQL, PostgreSQL, and x11vnc are build using this model. In such cases, the execution of every thread can be considered linear, and we can partition execution into an almost arbitrary number of segments based on a variety of conditions that occur over time. For example, most of these servers support user authentication to prevent or restrict unauthenticated users. If we can identify the part of the code that handles user authentication, we can use its outcome to split execution to unauthenticated and authenticated partitions. Similarly, we can monitor the files being opened and read by a thread to further partition, when a particular set of data is accessed. In this manner, we can virtually partition all software that follows the one-thread-per-user model based on events occurring at various points during execution, but not by the part of the code being executed. That is, virtual partitioning is not spatial.

Worker-thread model. Multiple threads are still allocated, but there is no strict association between an execution thread and a user. For instance, consider the Apache web server. Requests are received by a thread in the server, and distributed to the various worker threads for processing. Even though at any point in time, a user's connection is processed by a single thread, all the user's requests are not processed by the same thread. In this mode, we can still identify events like user authentication to partition execution, but as every thread continuously switches between serving different users (possibly a mix of authenticated and newly connected users), switching between partitions becomes more complex. Essentially, we need to be able to build an association between clients and worker threads to determine the partition of each thread dynamically (*i.e.,* each time work is assigned to it). Tackling such applications is beyond the scope of this paper, but it remains a problem that we plan to investigate in the future.

Alternatively, developers can also adopt a single-process event-driven model, where execution is completely driven by I/O events (*e.g.,* think of the lighttpd web server). Since reasoning about control flow in this type of systems is extremely difficult [50], we consider virtual partitioning to be a bad fit for such applications.

3. PARTITIONING BASED ON USER AUTHENTICATION

We propose a methodology for automatically partitioning an application based on user authentication. Our approach aims at automatically determining the exact point where users are authenticated, or alternatively identifying a small set of potential authentication points from which the most appropriate can be manually selected, without access to application source code. The very nature of the authentication process assists us in this case because its outcome is binary. It either succeeds or fails, causing the execution flow of the application to follow a different path in each case. This means that it exists at least one point (*i.e.,* a function or branch) in the program, where execution flow changes based on whether authentication was successful or not. We call such locations *authentication points*.

We determine an application's authentication points by monitoring its execution flow when a user successfully authenticates, and comparing it with another flow produced from a failed authentication attempt. This process is based on the following observations. First, the application contains a function that performs the user authentication. This is also the case for functions that only perform requests to remote authentication services, such as Kerberos [27]. This function needs to signal the application of its outcome, frequently by returning a value. In other cases, it may only update a global variable or an argument, but the control flow will still deviate either within this function, or in one of its calling functions. Relying on these observations, we record three types of execution flow information in order to identify a branch or/and a function, where execution flow changes as a result of the authentication process:

1. Function return value, expressed as an unsigned integer.
2. The outcome of conditional branch instructions, which can be either taken or fall-through. For instance, in the x86 architectures, instructions like `JL`, `JNL`, and so on, may jump to the address hardcoded in the instruction itself, depending on the outcome of a previous comparison (indicated by the respective bit on the `EFLAGS` register).
3. The runtime function call graph (FCG) of the application. This is a directed graph that represents the functions being called, as well as the calling relationships between them (*e.g., f1() → f2() → f3()*).

Figure 3 illustrates the methodology used to automatically determine authentication points. First, we profile the application by performing a series of successful and failed authentication attempts, while we record the information listed above using a control flow monitor (CFM). The collected information is stored in a DB and processed by our flow trace analyzer (FTA) to produce a list of possible authentication points. In the remainder of this section, we elaborate on the operation of these two components.

3.1 Control Flow Monitor

We built the CFM using Intel's Pin [25] dynamic binary instrumentation framework (DBI). Pin enables developers to instrument any binary application, and create tools that can monitor, or augment, various aspects of its execution. In our case, we created a Pintool that injects small pieces of monitoring code before every branch instruction, as well as at the entry and exit points of functions. For branches, our code records the *relative address* of each branch, and whether it was taken or not. The relative address of a branch is expressed using the name of the executable image containing it (*e.g.,* the name of the binary or a dynamic shared library) and its offset from the beginning of the image (*e.g.,* `sshd+101346`). Similarly, for the called functions we record their name (if available), their relative address, and their return value. Since CFM records relative addresses (instead of absolute ones) both for branches and functions, it can properly handle stripped applications (*i.e.,* applications that do not include any symbol information), and operates in systems where address space layout randomization (ASLR) [34] is in effect. This is because ASLR randomizes only the base address of certain system components (*e.g.,* stack, heap, and text image) when the process/program is created, without affecting the relative distance between intra-module objects.

The output of CFM is tightly bound to the specific binary being profiled. That is, compiling and running the same application on different systems can produce different traces. However, we can "move" the authentication point between different systems, if the binary was build using the same parameters. The recorded information is stored into a MySQL DB as name-value pairs, along with the information of whether the trace belongs to a successful or failed authentication run (this is supplied to the CFM by the user driving the profiling). Particularly for branches, we store their relative address and a boolean value indicating whether it was taken, while for functions we store their relative address and their return value as an unsigned integer. Note that the CFM extends the capabilities of existing tracing tools, like *ltrace*, as it records all function calls made (instead of only shared library functions), as well as fine-grained control flow information, such as the outcomes of branches.

3.2 Flow Trace Analyzer

The FTA analyzes the information stored in the DB to identify potential authentication points. An initial classification is made by looking for differences in the branches taken, and the values returned between successful and failed authentication attempts. In every case, there should be at least one branch different, and possibly functions that return different values, representing the deviation in execution flow due to the distinct authentication outcomes.

We examine whether a particular branch or function name (*i.e.,* a relative address) has a distinct value (*i.e.,* whether it was taken for branches, and the return value for functions) depending on the trace it belongs into. For instance, in the flow traces shown in Figure 3, the function `check_scramble` always returns `0` when authentication is successful and `1` when it fails. This is implemented by a MySQL stored procedure that exports the unique *function-return value* and *branch-boolean* pairs, effectively producing a list of distinct branch-outcome and function-return value pairs. For brevity, we will refer to the branches and functions of such pairs as *dbranch* and *dfunc* respectively. We use the differing branches and functions to identify potential authentication points. In particular, we define a *dbranch* as an authentication point when one of the following rules, in order of appearance, is satisfied.

Rule 1: a *dbranch* is located within a *dfunc*. For example, function *f2()* in Figure 4 returns different values depending on the success of the authentication, and contains a branch with a different outcome. This corresponds to the optimal case, where the respective *dbranch* is identified as being an authentication point.

Rule 2: a *dbranch* is located in one of the parent functions of a *dfunc*. This condition attempts to capture functions that do not actually perform the authentication themselves, but simply convey its outcome through their return value. For instance, consider a function querying a remote authentication service. If function *f2()*, shown in Figure 4, did not return any value, but still contained a *dbranch*, it would be identified as an authentication point by this rule.

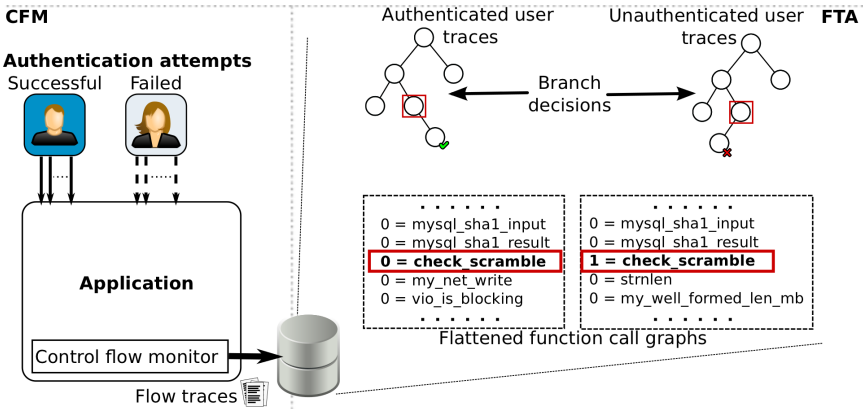
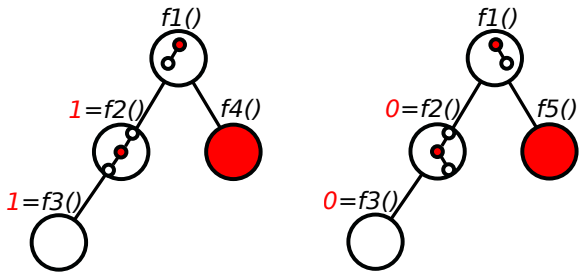


Figure 3: We use application profiling to automatically determine the authentication point of an application. The *control flow monitor* (CFM) logs the function calls and branch decisions performed by the application, while performing a series of successful and failed authentication attempts. We later analyze the collected traces with the *flow trace analyzer* (FTA) to determine the locations where execution flow changes. These locations constitute potential authentication points.



(a) Successful authentication.

(b) Failed authentication.

Figure 4: Example with the type of information collected by the profiler. The larger circles represent functions, and the numbers their return values. The smaller inner circles denote branches within the functions. This example contains all the indicators used by the FTA to determine possible authentication points: different return values, branch outcomes, and functions. The highlighted circles indicate differences between the flows of successful and failed authentication runs.

Rule 3: the FCG differs after a *dbranch*. This rule is useful for applications where the result of the authentication process is returned through a function argument or global variable, instead of the return value. For instance, consider that both functions *f2()* and *f3()* in Figure 4 do not return a value. However, *f2()* and its parent function *f1()* contain a *dbranch*, and the FCG deviates based on the outcome of the authentication (function *f4()* in Figure 4(a) is replaced by *f5()* in Figure 4(b)). In this case, *f1()* is identified as a possible authentication point.

If multiple authentication points are identified, we select the one that has been picked out by the most rules. We prioritize the rules, so for example, between authentication points picked by *Rule 1* and *Rule 2*, we prefer the first. However, if two authentication points are found equally fit, we rely on the FCG and pick the *dbranch* or *dfunc* evaluated first. Finally, it is possible that no *dfunc* is found and the FCG does not deviate, but at least one *dbranch* exists. This consists a worst case scenario that we believe is only viable in artificial or very small programs (*e.g.,* a program contained in a single, or only a couple, of functions). In this case, we consider the last evaluated *dbranch* as a possible authentication point.

The above rules formulate the conditions that occur when the execution path of an application changes depending on the result of the authentication process. Using them, we can identify any application's authentication point, as we experimentally demonstrate in Section 5. Note that in our approach we monitor the execution path followed by an application taking into account only the user's authentication input.

4. VIRTUAL PARTITIONING RUNTIME

4.1 The Pin DBI Framework

To partition binaries at runtime, we developed an environment based on the Pin [25] DBI framework. It enables developers to augment, modify, or simply monitor the execution of a binary at the instruction level. This is achieved through an extensive API that allows Pintools to *instrument* an application by installing callbacks to inspect instructions, routines, and so forth, or to modify the application by removing or adding instructions. The added code, referred to as *analysis* code, is combined with the original code using just-in-time (JIT) compilation and stored in a code cache where it is executed from. Hence, each code block is translated only once.

We chose Pin because it can run unmodified binaries and it allows us to intercept all the events that we could use for virtually partitioning an application. Furthermore, in the past it has been used to implement various security mechanisms [20,29,36,39,57]. Note that although Pin can run on multiple architectures (*e.g.,* x86/x86-64/IA64/ARM Linux, x86/x86-64 Windows, x86 MacOS X), we developed and tested our prototype on x86 Linux.

4.2 Supporting Multiple Partitions

To accommodate multiple partitions, we utilize Pin's versioning capabilities. Pin (v2.9 and later) has added inherent support for multiple code caches, or multiple instrumentation *versions*, which allow Pintools to instrument the same piece of code in different ways. Initially, an application runs in version zero that corresponds to the default code cache. Pintools can alter the code cache version that a certain thread is executing from, either statically when specific code blocks are encountered, or dynamically from the analysis code (*e.g.,* based on some condition). When a thread switches to a new version, execution continues from a new code cache allocated for that version.

In case a block of code has not been instrumented for a certain version, the instrumentation routines defined by the Pintool will be invoked again. Note that newly spawned threads and forked processes execute in the same version as their parent (*i.e.,* the thread that called `clone`). We created a Pintool for running an application in three different modes, or *versions*, using the versioning capabilities presented above (more versions can be accommodated to host more protection mechanisms). *Version 0* runs the application over Pin, as is, and without instrumenting it with additional code, while *Versions 1 and 2* can include different types of instrumentation for transparently applying various security techniques on the binary.

4.3 Switching Between Partitions

Based on authentication. Switching between different partitions based on an authentication point is straightforward. Our tool receives the authentication point identified in the profiling phase, and uses it to install analysis code that moves execution among partitions. For instance, if we determined that taking a particular branch indicates successful authentication, we use Pin's API to set up an analysis routine to execute if the branch is taken, so as to switch the version of the executing thread.

Based on data access. Accessing data of a particular interest can be also used as an event to partition application execution. For example, reading data or even opening a file can be used as a point where we can dynamically switch between partitions. We monitor file accessing system calls to switch to a different partition when user-configured files are opened or when data are read from them. We accomplish this by intercepting system calls like `open`, `dup`, and `close` to keep track of the file descriptors that correspond to the user-configured files. Furthermore, we monitor the `read` system call and switch partitions, when it is called with a file descriptor in the tracked set.

4.4 Protection Mechanisms

We incorporated two freely available Pintools, namely *libdft* [20] and *ISR using Pin* [36] in our tool. The first enables us to apply dynamic taint analysis (DTA) on applications. DTA can be used to protect from control-flow diversion attacks, like buffer overflows [28], by tracking network data and enforcing how they are used (*e.g.,* disallowing their use as control data). DTA can also prevent information leaks [57], by tracking selected data and disallowing certain operations, like transmitting tagged data over the network. ISR provides protection against code-injection through instruction-set randomization. DTA is a powerful protection technique, but incurs high overheads, while on the other hand, ISR protects against a smaller group of attacks, but it is faster.

These mechanisms can be applied in different ways based on the desired use of partitioning, as discussed in Section 2.1. If we desire to harden the pre-authentication partition of an application against memory corruption attacks and use a more lightweight mechanism after that, we can employ DTA before authentication and ISR after. To address different types of threats, like memory corruptions attacks before authentication and sensitive information leaks, we can use ISR before authentication and DTA after. We could even utilize DTA on both partitions, but to different ends (*i.e.,* memory exploits prevention vs. information leaks detection).

We can also disable all protection mechanisms, and run a partition simply over Pin with no additional instrumentation. Currently, we are not able to detach and reattach Pin to a single thread of an application, so threads need to keep running over Pin. However, if an application utilizes one process per user and if henceforth we do not desire to switch to another partition, we could detach Pin entirely, running the process natively with no additional overhead.

5. EVALUATION

In this section, we evaluate the performance implications of using virtual partitioning to apply different security techniques on the identified partitions, as well as the accuracy of our methodology in automatically determining authentication points. We employ seven Linux server applications, namely MySQL, Samba, x11vnc, SVNserve, PostgreSQL, OpenSSH, and Pure-FTPd. Our evaluation testbed consisted of a single host featuring two 2.66GHz quad-core Intel Xeon X5500 CPUs and 24GB of RAM, running Debian Linux v6 ("squeeze" with kernel v2.6.32).

5.1 Identifying Authentication Points

We profiled and analyzed the applications using the methodology described in Section 3. For those applications supporting more than one authentication mechanism, like OpenSSH and Pure-FTPd, we tested each of them individually. Note that in order to discover the authentication points, we did not use any kind of input fuzzing [43]. The only input modified was the credentials. Table 1 summarizes the results of our analysis. The FTA successfully discovered the correct authentication point in all cases. For the applications in Table 1, we only had to perform a single successful and a single failed authentication attempt, while collecting traces with the CFM. Other applications may require more traces to be collected before the FTA can identify an authentication point. However, these results indicate that our methodology works well with commodity software.

The authentication points for MySQL, Samba, and OpenSSH (configured with public key authentication), were determined by a single rule (*Rule 2*). In contrast, multiple rules (*Rule 1 and 2*) selected an authentication point in the case of PostgreSQL, OpenSSH, and Pure-FTPd, when authenticating through the PAM [44] mechanism. This is because the result of the authentication process is used in more than one locations in the application. We followed the strongest indicator (*Rule 1*), as described in Section 3. Last, *Rule 3* determined the authentication points for OpenSSH, Pure-FTPd, SVNserve, and x11vnc, when password authentication was employed, since a function-return value pair (*i.e.,* a *dfunc*) was not found.

Note that when OpenSSH and Pure-FTPd authenticate through PAM, we identified the same authentication point. In particular, the FTA initially detected two functions that both matched *Rule 1 and 2*, namely `verify_pwd_hash` and `_unix_verify_pwd`. We can infer, by simply examining the function names, that both are part of the authentication process. The FTA utilized the FCG to select the first authentication point, even though selecting either one would still be correct. Our FCG defined the relationship between these functions as follows: `pam_sm_authenticate` → `_unix_verify_pwd` → `verify_pwd_hash`, which indicates that the caller function only uses the value returned by the callee, and hence, results in the selection of `verify_pwd_hash`.

One might argue that the authentication points, and particularly the functions, can be identified by manually inspecting the source code. However, this is far from being practical, as source code or debugging symbols are not always available (*i.e.,* not applicable to commodity software), and even when they are, most applications consist of hundreds of thousands lines of code (LOC) and numerous symbol names. For example, MySQL v5.0.67 has more than a million LOC and ten thousand symbols, as reported by the `cloc` [30] and `nm(1)` utilities respectively. Additionally, function names are not always indicative (*e.g.,* consider `check_scramble` in MySQL), and as many applications implement their own custom authentication mechanisms (*i.e.,* the top 5 applications in Table 1), documentation (when available) does not necessarily provide any insight on the matter.

Service	Authentication scheme	Function/return value pair	Branch address	Matched rules
MySQL	Custom password verification	`check_scramble`/0	`mysqld+2616425`	2
Samba	Custom password verification	`hash_password_check`/1	`smbd+1598125`	2
SVNserve	Custom password verification	*not applicable*	`svnserve+28684`	3
PostgreSQL	Custom password verification	`md5_crypt_verify`/0	`postgres+1656160`	2
x11vnc	Custom password verification	*not applicable*	`vncserver+66913`	3
OpenSSH	OpenSSL public key signature verification	`RSA_public_decrypt`/35	`libcrypto+564191`	2
	PAM framework	`verify_pwd_hash`/0	`pam_unix+25607`	1 & 2
	Unix password verification	*not applicable*	`sshd+101346`	3
Pure-FTPd	PAM framework	`verify_pwd_hash`/0	`pam_unix+25607`	1 & 2
	Unix password verification	*not applicable*	`pure-ftpd+28296`	3

Table 1: Authentication points identified by the FTA. The table lists the function-return value pairs and branches that determine successful authentication, as they were selected by FTA. Note that some applications support multiple authentication schemes. In all cases, we manually verified that that selected authentication point is correct by inspecting each application's source code.

For the applications implementing their own custom authentication mechanisms, we validated the identified authentication points by manually inspecting the source code for the functions that were selected by the FTA. In the case of MySQL, we located the function `check_scramble`, which according to the developers' comments "*returns zero in the case of correct password otherwise a non-zero value*". In Samba, the function `hash_password_check` was annotated by developers with "*compare password hashes against those from the SAM*", while the `md5_crypt_verify` function in PostgresSQL checks user encrypted or plain-text passwords depending on the configured method used for authentication.

When using PAM, we located the `verify_pwd_hash` function in the source code, however we did not find any comment regarding its functionality. On the other hand, the documentation pointed us to `pam_authenticate`, which performs authentication by calling `pam_sm_authenticate`. After further examining the source code, we discovered the following relationship: `pam_sm_authenticate` → `_unix_verify_pwd` → `verify_pwd_hash`, which confirms the correctness of our result. When using Unix password verification with OpenSSH and Pure-FTPd, we identified only one particular branch, and not a function, as an authentication point, because the result is returned via an argument (see *Rule 3* in Section 3.2). Specifically, Pure-FTPd uses the `pw_unix_check` function to check a password's correctness, and updates its first argument (`AuthResult * const result`) with its outcome. The same is true for x11vnc that uses its own password verification mechanism.

Finally, when OpenSSH was configured to use public key-based authentication, we identified `RSA_public_decrypt`, which is called by `openssh_RSA_verify` in OpenSSL's source code. According to the documentation, the function returns the size of the recovered message digest on success and -1 otherwise.

5.2 Partitioned Execution

One of the applications of virtual partitioning is to manage the attack surface of software that inherently consists of segments with asymmetric exposure to attacks, as we described in Section 2.1. The first part of our evaluation showed that we can automatically detect such partitions based on user authentication. In this section, our goal is to evaluate the performance benefits that can be reaped, by utilizing virtual partitioning to apply otherwise expensive protection mechanisms on the most exposed part of applications. This allows us to strike a balance between the overhead imposed on the application and its exposure to attacks.

In particular, we evaluate the configurations described in Section 4.2. We employ DTA in the pre-authentication partition of the application and switch to either ISR, or no protection, in the post-authentication partition. We compare these two configurations with running the application natively, under Pin with no instrumentation (null Pintool), and entirely under ISR or DTA. We should note that when employing DTA, ISR is also active, since utilizing ISR for the entire execution was more efficient than activating it after authenticating.

Among the applications listed in Table 1, we used OpenSSH, Pure-FTPd, Samba, and MySQL for this part of the evaluation. For the first three, we transferred 1GB of randomly generated data over a 1Gbps connection, and measured the average *throughput* and *total execution time* needed to complete the transfer. For convenience, we used public key and PAM authentication for OpenSSH and Pure-FTPd respectively. For the MySQL server, we used its own *test-insert* benchmark suite, which reports the time needed to complete a set of actions, like table creation, data insertion and selection, and so on. All tests were iterated 20 times, and figures draw the average and standard deviation (std.dev.).

Figure 5 illustrates the time required to complete MySQL's test-insert benchmark. Applying DTA and ISR on the server for the entire duration of the test increases execution time by 4.8x and 2.6x respectively, when compared to native execution. In contrast, partitioning slows down execution by 1.8x and 2.6x, when using DTA only for the non-authenticated part of the execution, and then switching to *no instrumentation* and *ISR* respectively. We observe that the overhead of applying DTA diminishes, as the unauthenticated partition runs only for a short period of time. In general, partitioned execution performs similarly to the mechanism applied on the authenticated partition.

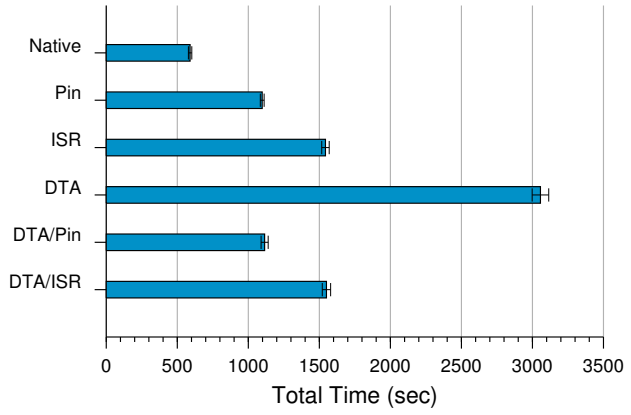


Figure 5: Time to complete MySQL's *test-insert* benchmark suite. The test performs a collection of different SQL operations (table creation, data insertion and selection) and includes user authentication. The six bars correspond to running the server natively, under a null Pintool, DTA, and ISR, and with virtual partitioning (DTA/Pin, DTA/ISR).

Figures 6 and 7 show the results of our experiments when the virtual partitioning is applied on Pure-FTPd, Samba, and OpenSSH. Particularly, Figure 6 draws the time needed to complete the transfer of 1GB file when the server runs natively, under the different protection schemes, and using our partitioning solution. Correspondingly, Figure 7 draws the achieved throughput. These experiments corroborate the results of MySQL, as we see a similar pattern in the performance of these servers. A notable difference is that as the experiments become more short-lived, the performance of configurations running under virtual partitioning falls somewhere in between the performance of the individual mechanisms, instead of being closer to the faster one. For instance, as illustrated in Figure 6, transferring the data using DTA/Pin partitioning takes 22.64s on average, which is between the ISR transfer (21.50s) and the DTA transfer (25.52s). It should be noted that when using such a configuration we can achieve significant performance improvements on CPU-bound applications, such as OpenSSH and MySQL.

In Figure 8, we employ partitioning based on data access and apply DTA to detect information leaks. We run the MySQL server natively, under Pin, using DTA to detect information leaks, and using DTA along with partitioning. We use MySQL's *test-ATIS* benchmark, which creates 28 tables, inserts data, performs a series of *select* operations on them, and then deletes them. When using DTA, we configured the table *airport* as being sensitive, so all data being read from the table were tagged and tracked. With partitioning, DTA is only activated when data are first read from this table, after the tables are created and a set of selects has already been performed. We observe that, in this particular setup, employing partitioning performs approximately $2x$ faster than having DTA enabled continuously.

Finally, we calculated how many instructions were encountered and translated by Pin when running in the different partitions (remember Pin only translates each instruction once for every version), and compared it with the actual number of instructions executed on pre- and post-authentication segments. We illustrate the results in Figure 9. As expected, we observe that all applications execute for the most part in the authenticated partition. Interestingly, we see that OpenSSH uses more code to perform user authentication than to later copy the data, even though the majority of its instructions run in the authenticated partition.

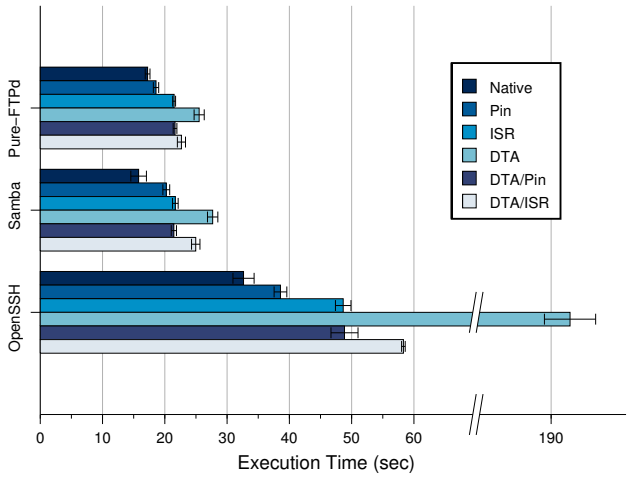


Figure 6: Total execution time for Pure-FTPd, Samba, and OpenSSH, when operating under different configurations. We transfer 1GB of data over a 1Gbps link.

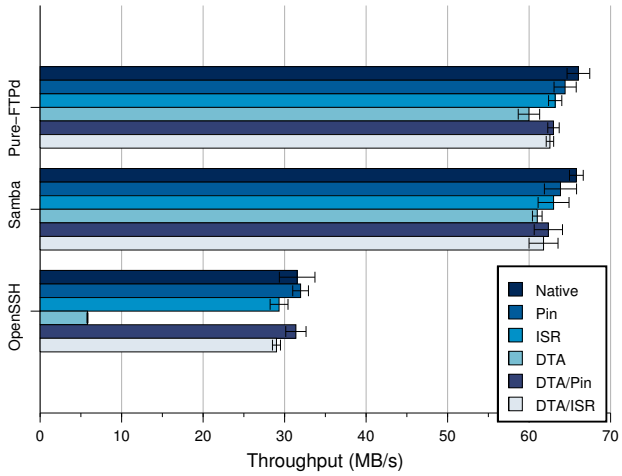


Figure 7: Throughput for Pure-FTPd, Samba, and OpenSSH, when operating under different configurations. We transfer 1GB of data over a 1Gbps link.

MySQL follows a somewhat similar pattern. Such services are great candidates for virtual partitioning, since the area of code that is exposed to all network users is relatively large, while their execution is concentrated in the post-authentication partition.

6. DISCUSSION

Virtual partitioning aims at providing adaptive defenses for software, by enabling administrators to control the attack surface of applications. In addition, it is orthogonal to existing solutions, in the sense that it does not affect their operation. For instance, if a program utilizes a tool such as Orp [33] for protection against Return-Oriented Programming (ROP), then our solution will not affect the effectiveness of that tool. The same also stands for other defenses, such as ASLR [34]. More importantly, virtual partitioning provides the means to strike a balance between the amount of resources dedicated to security and the level of protection provided, by offering a way to control the intensity of the applied defenses.

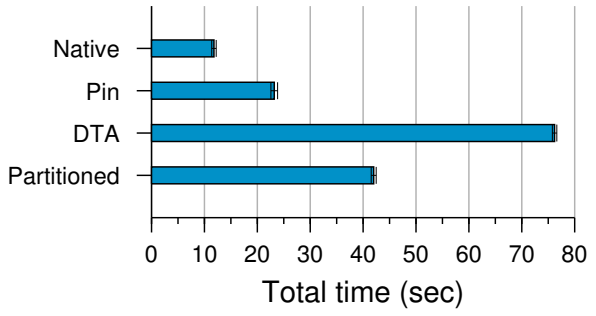


Figure 8: Time to complete MySQL's *test-ATIS* benchmark. It creates 28 tables and performs a series of `SELECT` operations on the them. We ran the MySQL server natively, under Pin, and using DTA to track the data of a particular table (*airport*). First by applying DTA throughout the benchmark, and then by partitioning execution and activating it only after data are read from the table (labeled as *Partitioned*).

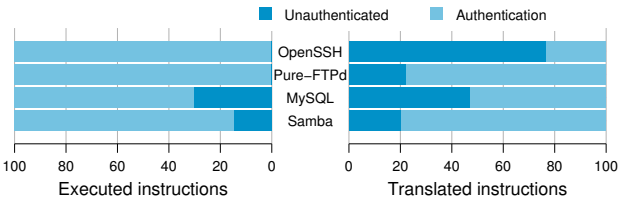


Figure 9: The percentage of instructions translated and run in the authenticated and unauthenticated partitions. The test cases we employed were: copying a 1GB file over OpenSSH, Pure-FTPd, and Samba, and the MySQL *test-insert* benchmark.

Essentially, it enables us to switch the security mechanisms applied to commodity software (*i.e.,* binary-only software), when a particular event occurs (*e.g.,* authentication or accessing sensitive data).

Virtual partitioning is not made obsolete by applications that follow the separation of privilege and least privilege principles. For instance, consider that an attacker launches a zero-day memory corruption attack against an OpenSSH server, exploiting a vulnerability in its pre-authenticated part, to gain access as a legitimate user. Privilege separation ensures that authorized users exploiting the server cannot gain elevated privileges, because the process that serves them runs with their own, typically lesser, privileges. However, the part of the server that handles authentication executes with administrative privileges until the user authenticates, so an attack targeting OpenSSH's pre-authentication code would not only grant an attacker access, but also administrative privileges.

If we augment the server with virtual partitioning, we can protect its pre-authenticated part against such attacks,[1] without incurring significant overhead to legitimate users. In fact, since OpenSSH allocates a separate process for each user, we could even configure our runtime to completely detach from the process after successful authentication. Note that in this case we cannot further partition execution by introducing other security mechanisms after certain events of interest occur.

[1]Note that we tested MySQL, running over our virtual partitioning solution, against exploits for CVE-2008-0226 [11] and CVE-2009-4484 [13] (both occurring before authentication) and DTA managed to capture the respective attacks.

Virtual partitioning can also be used to enhance the security in software being identified as vulnerable to specific threats, or when it uses weak security algorithms for backward compatibility. In that case we can identify a possible partition point only if we know the "parameters" that affect software security. For example, the names of vulnerable functions, the set of weak algorithms, or any information used by the software and weakens its security. Exploiting this information allows us to identify the point of interest for partitioning the vulnerable software, and to enable protection mechanisms that can reduce the risk of its security requirements being violated.

Note that in case we erroneously detect an authentication point, it simply means that we will not switch from one mechanism to the next, as desired. This can lead to slower performance or applying an inappropriate security mechanism, but it does neither affect application correctness, nor the possible switching mechanisms (e.g., accessing sensitive data). However, taking into account the results of our evaluation (see Section 5) we did not encounter such a case.

Limitations and Future work. Throughout this paper, our main focus has been on how to adapt software defenses and dynamically manage the attack surface of binary-only software. However, nowadays more and more network-facing software is written in interpreted languages (*e.g.,* PHP, Perl, or Ruby on Rails), making it an imperative to also provide protection services to applications written in such languages. In principle, the concept of virtual partitioning could be applied in programs written in any language, since the technique is language *agnostic*, but our current prototype (CFM, FTA, and runtime) is solely geared towards x86 binaries. To this end, we are currently in the process of investigating whether we can utilize Facebook's HipHop [17] code transformer, so as to provide selective and targeted protection to PHP applications. Our initial involvement showed promising results, since for "toy" applications we managed to identify authentication points with moderate effort. Ongoing work focuses on extending FTA so that in can handle PHP programs compiled with HipHop. Finally, in future work, we will investigate other partitioning schemes that allow for accurate, targeted application of software defenses as and where needed.

7. RELATED WORK

7.1 Privilege Separation

Software design principles like *privilege separation* and *least privilege* have been around for a long time, even though admittedly, they are frequently ignored due to performance considerations and rapid software development cycles. Saltzer *et al.* [41] were among the first to summarize many such separation principles, and explore their application in address space segmentation. The OKWS toolkit [23] also defines various security guidelines, for assisting the users of the toolkit in building secure and highly complex systems, like web servers and browsers. Similarly, Capsules [22] proposes a programming model for web applications, which enables developers to divide applications in privilege separated components, facilitating the safe integration of third-party modules. However, these approaches require that an application is designed and implemented with these considerations in mind, and cannot be applied on existing software.

Certain popular applications do incorporate the above principles to perform some type of partitioning. For instance, OpenSSH immediately drops administrator privileges after a user authenticates, while a new processes is spawned for each connected user, as proposed by Provos *et al.* [38]. The Chrome web browser also uses processes to isolate its components and protect against malicious content [40]. The browser is divided into two protection domains, the browser's kernel that interacts with the operating system, and

the rendering engine that runs with restricted privileges in a sandbox. This way, a vulnerability in one of the components does not affect the others. Our approach is orthogonal to such designs, as we have shown by applying virtual partitioning on OpenSSH.

Systrace [37] uses system call monitoring to detect and prevent intrusions, based on policies that define benign application behavior in terms of valid system call profiles. It also introduces *privilege elevation*, a technique that enables a process to temporarily acquire administrator privileges to execute certain system calls. This technique eliminates the need for `setuid` and `setgid` applications, by essentially partitioning a program to privileged and unprivileged segments, with the first containing all the system calls that need to execute with higher privileges. Unlike virtual partitioning, privilege elevation requires that the user manually specifies the privileged partitions of the application. It is also more limited, as the partitions are more rigidly defined. Bapat *et al.* [3] investigate the automatic specification of the privileged Systrace segments using black box analysis. Similarly to Systrace, the Privman library [21] provides a systematic and reusable approach for partitioning operations that require higher privileges through the provided API. However, applications need to be modified to use the library.

Murray *et al.* [26] propose performing privilege separation at the library level. In their scheme, dynamic libraries can be separated from the main application by moving them into a shadow process where they execute in isolation. Calls into the library are performed through a special memory area which passes control to the separated library running in the shadow process. Their approach requires a hypervisor like Xen, and only offers coarse-grained partitioning that does not suffice for the scenarios we describe in this paper. A semi-automatic privilege separation technique is proposed in Privtrans [5]. The developed tool can retrofit privilege separation in applications, when source code is available and provided that the programmer annotates it to highlight the privileged operations. This way, OpenSSH-like privilege separation can be integrated in other applications with moderate effort.

7.2 Data Separation

Secure program partitioning [56] introduces a compiler-based technique that partitions a program into subprograms, so that only certain subprograms can access "sensitive data". This approach requires that the programmer annotates the code to specify the sensitive data, as well as the policies that are to be applied. Wedge [4] focuses on preventing information leaks. It allows the creation of compartments that are guaranteed not to leak sensitive user information. Sensitive memory objects are identified and used to build the appropriate code segments that can access them. A similar approach is also employed by Liu *et al.* [24] to protect the Java virtual machine (JVM) against illicit memory operations that may corrupt it, causing it to crash. ABYSS [53] and AEGIS [46] both describe a secure chip architecture that enables some regions of an application to execute in a secure mode (encrypted), while the remaining ones executed in insecure mode (unencrypted), hence partitioning in this way their execution. In both architectures, the application designer defines the execution mode for each application region.

8. CONCLUSIONS

We described *virtual partitioning*, a technique for dynamically managing an application's attack surface by adapting the applied software defenses and tuning their intensity, based on the currently executing facet (partition) of the application. To demonstrate feasibility and practicality, we examined applications that employ user authentication, and used the point where authentication takes place during execution for partitioning them. We presented a methodology for automatically determining the authentication points in binary applications, with little or no user supervision, and developed a runtime environment that can execute unmodified binaries using virtual partitioning, while applying techniques such ISR and DTA on the different partitions.

Results show that our solution can automatically determine the authentication point of well-known applications, without supervision and without the need for source code or symbol information. Furthermore, our runtime environment that switches between the various protection mechanisms based on the virtual partitions, incurs little overhead. Our experiments clearly demonstrate that applying a heavyweight protection mechanism, such as DTA, only on the pre-authentication part of applications, incurs moderate overhead compared to its continuous application. This establishes that virtual partitioning, besides enabling the application of diverse defense mechanisms, provides a practical alternative to the "paranoid", always-on deployment of heavyweight techniques.

Availability

Our prototype implementation is available at: `http://code.google.com/p/virtual-partitioning/`

9. ACKNOWLEDGMENTS

This work was supported by the US Air Force under Contract AFRL-FA8650-10-C-7024 and by DARPA under Contract FA8650-11-C-7190. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, or the Air Force.

10. REFERENCES

[1] P. Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the* 19^{th} *USENIX Security Symposium (USENIX Sec)*, pages 177–192, 2010.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In *Proceedings of the* 29^{th} *IEEE Symposium on Security and Privacy (S&P)*, pages 263–277, 2008.

[3] D. Bapat, K. Butler, and P. McDaniel. Towards Automated Privilege Separation. In *Proceedings of the* 3^{rd} *International Conference on Information Systems Security (ICISS)*, pages 272–276, 2007.

[4] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the* 5^{th} *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 309–322, 2008.

[5] D. Brumley and D. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the* 13^{th} *USENIX Security Symposium (USENIX Sec)*, pages 57–72, 2004.

[6] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the* 7^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–160, 2006.

[7] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the* 14^{th} *USENIX Security Symposium (USENIX Sec)*, pages 177–192, 2005.

[8] COMPUTERWORLD. Microsoft BPOS cloud service hit with data breach. `http://www.computerworld.com/s/article/9202078/Microsoft_BPOS_cloud_service_hit_with_data_breach`, December 2010.

[9] CVE. CVE-2003-0780. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0780, September 2003.

[10] CVE. CVE-2006-6170. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6170, November 2006.

[11] CVE. CVE-2008-0226. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0226, January 2008.

[12] CVE. CVE-2009-1394. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1394, April 2009.

[13] CVE. CVE-2009-4484. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-4484, December 2009.

[14] CVE. CVE-2012-2110. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2110, April 2012.

[15] S. Designer. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63, August 1997.

[16] Dropbox. Homepage of Dropbox ^{TM} Cloud Storage. http://www.dropbox.com, June 2012.

[17] Facebook Developers. HipHop for PHP. https://developers.facebook.com/blog/post/2010/02/02/hiphop-for-php--move-fast/, June 2012.

[18] GEEKOLOGIE. Disgruntled IT Administrator Commandeers San Francisco City Network, Gets Arrested, Sticks It To The Man By Refusing To Give Up Password. http://www.geekologie.com/2008/07/disgruntled_it_administrator_c.php, July 2008.

[19] ICS-CERT. Progea Movicon Data Leakage and Denial-of-Service Vulnerability. http://www.us-cert.gov/control_systems/pdf/ICSA-11-056-01.pdf, March 2011.

[20] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the* 8^{th} *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 121–132, 2012.

[21] D. Kilpatrick. Privman: A Library for Partitioning Applications. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*, pages 273–284.

[22] A. Krishnamurthy, A. Mettler, and D. Wagner. Fine-Grained Privilege Separation for Web Applications. In *Proceedings of the* 19^{th} *International World Wide Web Conference (WWW)*, pages 551–560, 2010.

[23] M. Krohn. Building Secure High-Performance Web Services with OKWS. In *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX ATC)*, pages 185–198.

[24] T. Liu, Y. Li, A. Schofield, M. Hogstrom, K. Sun, and Y. Chen. Partition-based Heap Memory Management in an Application Server. *SIGOPS Operating Systems Review*, 42(1):98, January 2008.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the* 26^{th} *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200.

[26] D. G. Murray and S. Hand. Privilege separation made easy: trusting small libraries not big processes. In *Proceedings of the* 1^{st} *European Workshop on System Security (EuroSec)*, pages 40–46, 2008.

[27] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

[28] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* 12^{th} *Network and Distributed System Security Symposium (NDSS)*, 2005.

[29] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing Security Checks on Commodity Hardware. In *Proceedings of the* 13^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 308–318, 2008.

[30] Northrop Grumman Corporation. CLOC: Count Lines of Code. http://cloc.sourceforge.net, April 2012.

[31] G. Novark and E. D. Berger. DieHarder: Securing the Heap. In *Proceedings of the* 17^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 573–584, 2010.

[32] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries. In *Proceedings of the* 26^{th} *Annual Computer Security Applications Conference (ACSAC)*, pages 49–58, 2010.

[33] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the* 33^{rd} *IEEE Symposium on Security and Privacy (S&P)*, pages 601–615, 2012.

[34] PaX. Homepage of The PaX Team. http://pax.grsecurity.net, June 2012.

[35] P. Porras, H. Saidi, and V. Yegneswaran. Conficker C Analysis. Technical report, SRI International, 2009.

[36] G. Portokalidis and A. D. Keromytis. Fast and Practical Instruction-Set Randomization for Commodity Systems. In *Proceedings of the* 26^{th} *Annual Computer Security Applications Conference (ACSAC)*, pages 41–48, 2010.

[37] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the* 12^{th} *USENIX Security Symposium (USENIX Sec)*, pages 257–272, 2003.

[38] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of the* 12^{th} *USENIX Security Symposium (USENIX Sec)*, pages 231–242, 2003.

[39] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the* 39^{th} *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 135–148, 2006.

[40] C. Reis and S. D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *Proceedings of the* 4^{th} *ACM European Conference on Computer Systems (EuroSys)*, pages 219–232, 2009.

[41] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, April 1975.

[42] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the* 14^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 552–561, 2007.

[43] S. Sidiroglou, O. Laadan, C. R. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: Automatic Software Self-healing Using REscue points. In *Proceedings of the* 14^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48, 2009.

[44] D. Smørgrav. Pluggable Authentication Modules. `http://www.freebsd.org/doc/en/articles/pam/`, May 2012.

[45] Sophos. Groupon subsidiary leaks 300K logins, fixes fail, fails again. `http://nakedsecurity.sophos.com/2011/06/30/groupon-subsidiary-leaks-300k-logins-fixes-fail-fails-again/`, June 2011.

[46] G. E. Suh. *AEGIS: A Single-Chip Secure Processor*. PhD thesis, Massachusetts Institute of Technology, 2005.

[47] The Register. Disgruntled admin gets 63 months for massive data deletion. `http://www.theregister.co.uk/2008/06/13/it_manager_rampage_sentence/`, June 2008.

[48] The Wall Street Journal. Google Discloses Privacy Glitch. `http://blogs.wsj.com/digits/2009/03/08/1214/`, March 2009.

[49] US-CERT. SSH CRC32 attack detection code contains remote integer overflow. `http://www.kb.cert.org/vuls/id/945216`, October 2003.

[50] R. von Behren, J. Condit, and E. Brewer. Why Events Are A Bad Idea (for high-concurrency servers). In *Proceedings of the* 9^{th} *Workshop on Hot Topics in Operating Systems (HotOS)*, 2003.

[51] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: practical capabilities for UNIX. In *Proceedings of the* 19^{th} *USENIX Security Symposium (USENIX Sec)*, pages 29–46, 2010.

[52] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the* 18^{th} *ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243, 2001.

[53] S. R. White and L. Comerford. ABYSS: An Architecture for Software Protection. *IEEE Transactions of Software Engineering*, 16(6):619–629, June 1990.

[54] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the* 15^{th} *USENIX Security Symposium (USENIX Sec)*, page 121–136, 2006.

[55] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the* 14^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 116–127, 2007.

[56] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure Program Partitioning. *ACM Transactions on Computer Systems (TOCS)*, 20(3):283–328, August 2002.

[57] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. *SIGOPS Operating Systems Review*, 45(1):142–154, January 2011.