Privilege separation made easy

Trusting small libraries not big processes

Derek G. Murray
University of Cambridge Computer Laboratory
Cambridge, United Kingdom
Derek.Murray@cl.cam.ac.uk

Steven Hand
University of Cambridge Computer Laboratory
Cambridge, United Kingdom
Steven.Hand@cl.cam.ac.uk

ABSTRACT

At the heart of a secure software system is a small, trustworthy component, called the Trusted Computing Base (TCB). However, developers persist in building monolithic systems that force their users to trust the entire system. We posit that this is due to the lack of a straightforward mechanism of partitioning – or *disaggregating* – systems into trusted and untrusted components. We propose to use *dynamic libraries* as the unit of disaggregation, because these are a familiar abstraction, which is commonly used in mainstream software development.

In this paper, we present our early ideas on the *disaggregated library* approach, which can be applied to existing applications that run on commodity operating systems. We first make the case for a new approach to disaggregation, and then describe how we are implementing it. We also draw comparisons with the wide range of related work in this area.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*

General Terms

Design, Security

Keywords

Disaggregation, Libraries, Virtualisation

1. INTRODUCTION

Many current software systems are implemented as monolithic processes, which place a large burden of trust on their clients and users. We believe that the lack of a straightforward mechanism for subdividing software into trusted and untrusted components is holding back the adoption of "best practices" for security. Therefore, we present a new approach, based on the well-understood and commonly-used concept of *dynamic libraries*.

Several researchers have discussed the problem of dividing a monolithic piece of software into several smaller pieces, each of which runs with the least necessary privilege. Disaggregation [16], partitioning [8], privilege separation [17] and TCB-reduction [13, 21] are simply different names for the process of dividing software into a small trusted computing base (TCB), and a larger untrusted part. However, existing solutions to this problem either use ad hoc techniques or source code annotation to split the code.

Dynamic libraries – i.e. collections of executable code and data which are linked into a process when it is loaded – are a natural unit of modularity in most common programming languages. Clients may invoke functions in dynamic libraries using the same calling convention as any other function, which is primarily due to the fact that a dynamic library exists in the same address space as its host process[1]. The dynamic library may also be used as a unit of abstraction, by hiding the implementation and merely presenting the client with the library interface. However, dynamic libraries are not secure against malicious clients.

We therefore propose a new approach which allows dynamic libraries to be used as the unit of disaggregation. We recognise that an address space is not necessarily the same as a protection domain [6], and thereby run the host process and disaggregated library in different protection domains, but the same logical address space. One may then disaggregate an application or service by moving sensitive or privileged operations into a dynamic library, and disaggregating that library using our proposed mechanism. Unlike most previous work, we aim to apply this technique directly to existing commodity operating systems and applications. We also aim to maintain binary compatibility with existing executables, so that recompilation is not necessary.

We intend for this approach to be useful in a wide variety of research into disaggregation and system security. For example, we intend to use it to answer the following questions:

- How should we choose an appropriate TCB? Are there any metrics that determine the "optimal" TCB? Can we use this approach to perform experiments on a variety of different TCBs, without manual intervention?
- Can this approach be used to "automatically" disag-

[1]Thanks to position-independent code, the same dynamic library may in fact exist in many different address spaces, each belonging to a different host process.

EuroSec '08 Glasgow, UK

gregate existing applications or services? If not, what magnitude of change is required to yield a secure solution?

- How easy is it for "regular developers" to develop a disaggregated system using this approach? What additional primitives – e.g. library support – are required to aid the programmer when developing for the disaggregated environment?
- What are the performance costs of disaggregation using this approach, compared to other techniques, such as RPC and IPC? How can this approach be optimised?

We begin by introducing the background to disaggregation, and making the case for a new disaggregation technique (Section 2). We then describe our design for how dynamic libraries can be used as the basis for disaggregation (Section 3). We also consider the wide range of related work (Section 4).

2. MOTIVATION

Why do we need a new approach to disaggregation? Several techniques are established in the research literature: microkernels [13, 21] and source code annotation [5, 14] are two of the best known. However, these are not widely used in real-world development, which continues to produce monolithic application written on top of monolithic operating system kernels. This suggests that the cost of adopting these techniques may outweigh the benefits (at least to developers) of disaggregation. We therefore need a new approach, which is compatible with existing software development practice.

In this section, we answer three pertinent questions:

What do we want to do? We want to improve the trustworthiness of the TCB. We define the term and discuss why it is an important concept in systems security. (Subsection 2.1.)

Why do we want to do it? We consider two of the most important use cases for disaggregation – privilege separation and data protection – and explain how disaggregation can be used to achieve these. (Subsection 2.2.)

How are we going to do it? We explain how dynamic libraries provide a useful abstraction for disaggregation, and how we can provide effective protection for their code and data. (Subsection 2.3.)

2.1 The trusted computing base

The motivation for most disaggregation work is either to reduce the size or improve the trustworthiness of the *trusted computing base* (TCB) [5, 13, 16, 21]. The TCB may be defined in various ways: at the high level, Hohmuth *et al* refer to the "set of components on which a subsystem S depends as the *TCB of S*." [13] In previous work, we made the low-level definition of the TCB as the set of code positions from which a privileged operation (i.e. one that can undermine the security of the system) may be invoked with arbitrary inputs [16]. Both definitions are equivalent, and imply that all code that may undermine the security of the system must be trusted.

In order to illustrate how the TCB is calculated, consider a concrete example: the management software for the Xen virtual machine monitor (VMM) [2]. The VMM is managed by a privileged virtual machine (VM), known as *Dom0*. The Dom0 kernel (which is a modified version of a standard Linux or Solaris kernel) has the special privilege that it can ask the VMM to (i) map any physical memory on the machine, and (ii) set the virtual CPU context for all other virtual machines. This privilege is necessary for Dom0 to be able to build new VMs. However, it could clearly be used to undermine the security of any VM on the same physical host: the Dom0 kernel must be trusted not to abuse the privilege. The Dom0 kernel makes this functionality directly available to user-space processes, using a device driver. Therefore any process which can open the "privileged command" driver can similarly undermine the security of any VM, and must be trusted not to do so. In effect, this places the VMM, the Dom0 kernel and every user-space process running as the root user into the TCB of every VM.

If we remove these privileges from the Dom0 kernel, and instead give them to a small component that has the sole function of building new VMs, we can reduce the TCB of every VM to include only this component and the VMM[2] [16]. We still provide the necessary functionality (i.e. VM building) using the same privilege, but we use an appropriate interface at a level of abstraction that makes it impossible for an attacker to make arbitrary use of that privilege. Since the amount of software in the TCB is now much smaller, and unchanging, we can be satisfied that the system is more trustworthy.

The main difficulty in TCB analysis is that there is no metric to quantify a "good" TCB, and therefore it is hard to argue that a particular TCB is optimal. The most often-used argument for trustworthiness is the number of lines of code that comprise the TCB [13, 17, 21]. However, from the above example, it is clear that the choice of an appropriate interface also has an important role in defining the TCB. We hope to use our new disaggregation approach to evaluate a wide variety of different TCB configurations, and hence develop better quantitative metrics that can be used to evaluate trustworthiness.

2.2 Disaggregation use cases

In this work, we will consider two key use cases for disaggregation: *privilege separation* and *data protection*.

Most existing disaggregation efforts target the privilege separation case, which splits a system into a small kernel that requires elevated privileges and a larger, untrusted process that does not [16, 17, 21]. Indeed, the division of computer systems into user and supervisor modes can be considered the earliest example of disaggregation [9]. Our work in this paper is motivated by an earlier effort to disaggregate the management software used with Xen virtual machine monitor, as described in the previous subsection. The converse of privilege separation is untrusted extension, whereby the majority of an application or service is trusted, but it is linked with an untrusted module (e.g. a plug-in or kernel module). Our approach will generalise to cover this case.

Data protection usually seeks to protect the confidentiality and integrity of passwords, private keys and other sensitive information. For example, Xen VMs can be given a virtual trusted platform module (vTPM), which provides

[2]For simplicity, we ignore the fact that Dom0 can perform arbitrary DMA, which would be solved by the use of an I/O memory management unit [3].

trusted computing support within a VM [4]. However, the vTPM implementation includes an emulator running in the management VM, which stores sensitive keys and register contents in unprotected data structures[3]: again, this places the entire management VM (which could map, and read or write these values) in the TCB of all virtual TPMs. By disaggregating the portion of this code which manages these data structures, behind an interface similar to that which a hardware TPM provides, it would be possible to remove the management VM from the TCB.

2.3 The solution: dynamic libraries

Dynamic libraries provide a useful abstraction to application programmers: a call to a dynamic library is implemented in the same manner as any other function call, and no limits are placed on the parameters, which may be scalar values, composite structures, pointers or even functions. This richness is possible because the caller and callee exist in the same address space. However, this also implies that the dynamic library is unprotected from the calling process. If a dynamic library carries out a sensitive operation, protected only by sanity checks in the library functions, a malicious caller can circumvent these checks by reverse-engineering the library and jumping directly to the sensitive operation in the library code.

We want to implement the TCB in one or more dynamic libraries, whilst excluding the majority of the calling process. An important requirement is that the calling convention for these libraries is not changed. In particular, it must be possible to pass a pointer to the library, as this is a common idiom in many programming languages for passing large arguments, or for returning data to the caller.

Chase *et al* distinguish between the concepts of *protection domain* and *address space*, which are not necessarily equivalent. They present an operating system architecture that uses a single address space, in which multiple possibly-overlapping protection domains reside [6]. One advantage of this approach is that pointer-rich data can trivially be transferred between protection domains without the complicated (and inefficient) serialisation and marshalling of arguments. Banerji *et al* describe how a library can be loaded in a different protection domain from its client using a modified operating system [1]: we propose to do the same using a commodity operating system, such as Linux or Microsoft Windows.

In order to load a dynamic library in a different protection domain from its host process, we require a privileged supervisor which can enforce the necessary isolation between the two protection domains, and which is necessarily part of the TCB. For example, we could use an operating system kernel, microkernel or virtual machine monitor to achieve this isolation, to different degrees of trustworthiness. We would then intercept library calls and perform the appropriate protection domain switch.

As we stated earlier, the key goal of this work is to make it easier for developers to create disaggregated applications and services by using existing development techniques. It is also important that the resulting software can be used in a familiar execution environment – i.e. on top of a commodity operating system – although the user may not want to have to *trust* the operating system kernel. In the following section, we describe how to implement our disaggregation approach using a commodity operating system on top of a VMM.

[3]An alternative suggested by the authors is to use a cryptographic coprocessor to host the vTPM software, but this is, as yet, not a common feature of commodity systems.

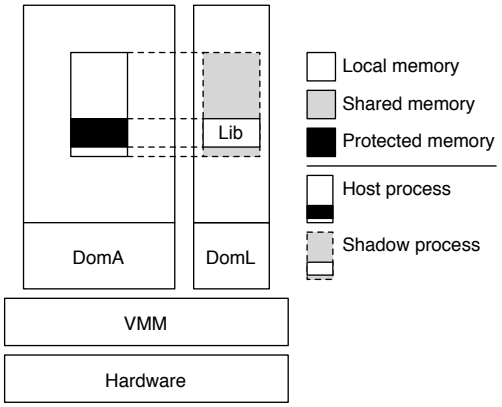

Figure 1: The host process and disaggregated library run in separate Xen domains, respectively DomA and DomL. In DomL, a shadow process loads the library at the appropriate address.

3. DISAGGREGATED LIBRARIES

We are currently implementing our dynamic library disaggregation approach. We implement our mechanism on the Linux operating system, running on top of the Xen virtual machine monitor, due to the availability of and our familiarity with the source code for these products. However, we expect that our approach is sufficiently general that it could be applied to other common operating systems or virtualisation technologies.

In the following discussion, we use the term *host process* to refer to the process that calls the library (excluding the library itself), and *disaggregated library* to mean the dynamic library which is protected from the host process.

To provide protection, we run the host process and disaggregated library in separate Xen domains (virtual machines). This ensures that the VMM mediates any sharing or communication between the two protection domains. Figure 1 shows a host process running in DomA, while the disaggregated library runs in DomL. In DomL, we create a *shadow process*, which contains the library loaded at the same address as it would have in the host process. However, the host process actually loads a special memory area, which is configured to pass control to the disaggregated library when an attempt is made to execute it (see Subsection 3.1 for more details). The shadow process has access to the memory of the host process (see Subsection 3.2).

We also illustrate how our approach can be applied to a real-world example (Subsection 3.3), and briefly discuss other implementation issues (Subsection 3.4).

3.1 Calling mechanism

In order to explain the calling mechanism, we first describe how the dynamic loader instantiates the disaggregated library in the host process, then describe the sequence of events when a call occurs.

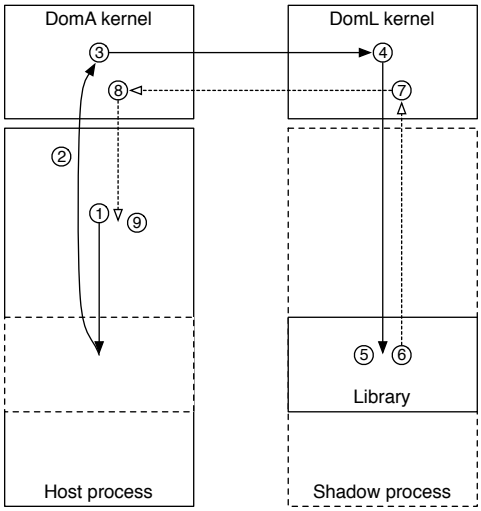

Figure 2: The sequence of events that occur in the execution of a call to a disaggregated library. The solid lines correspond to the initial call (steps 1–5). The dashed lines correspond to the return (steps 6–9).

When the host process attempts to load the dynamic library, it finds a structure that resembles a standard ELF shared object file, and can be parsed by the dynamic loader. For example, the host process must parse the symbol table in order to resolve dependencies on the library in other parts of the executable[4]. The loader will then attempt to map the text (executable code) and data segments of the library. However, the library file – implemented as a custom device inode – will instead create page table entries that mark these pages as "not present", and associate a special-purpose fault handler with these. At the same time, a shadow process is created in the library domain, and the library is loaded at the appropriate location.

Figure 2 illustrates the following sequence of events that occur when handling a call to the library. We give a concrete example of these steps in Subsection 3.3.

When the host process attempts to call any of the library code (1), this will cause a page fault, which transfers control to the DomA kernel (2). The handler is responsible for marshalling the relevant CPU state to the library domain (DomL), and notifying it that an invocation is pending (3). At this point, execution switches to the DomL kernel, which checks the incoming instruction pointer to ensure that it matches a valid function entry point (4). If it does not, DomL sends an error response to the DomA kernel, which then signals that a protection fault has occurred in the host process.

Assuming that the instruction pointer is valid, the shadow process can continue to execute the desired function (5). If it requires data from the host process, it can access it directly (assuming it has permission from the host process), using the memory sharing technique that we describe in the following subsection. When the library function returns (6), the DomL kernel marshals the relevant CPU state back to the host process, and notifies it of completion (7). The CPU state for the host process is restored (8) and control returns to the instruction following the original call (9).

This calling mechanism enables the disaggregated library to fulfill the privilege separation use case that we described in Subsection 2.2. For example, in the case of two domains running on Xen (as in Figure 2), we can give additional capabilities to DomL so that it can request privileged operations from the VMM. If we instead implemented this approach on a regular operating system kernel, we could set the user ID of the shadow process so that it may access protected resources: this would effectively give `setuid` functionality to libraries.

3.2 Memory sharing

We avoid the need for argument marshalling by sharing the address space between the host process and shadow process. We implement our data protection use case (see Subsection 2.2) by changing the protection bits on the relevant logical pages when we switch between the two domains. However, this leaves the question of how the address space is shared. We take a simple approach, based on the grant table mechanism for explicit page sharing that is provided by Xen [11].

In this approach, the address space of the shadow process contains only the library itself, mapped at the location where it was loaded into the host process. All necessary runtime structures – such as the stack, a private heap and a private memory-mapping area – must be contained within the library, by allocating uninitialised storage for them. By including them within the footprint of the library, we ensure that the host process will not use these areas of memory, and thereby avoid address space clashes. The remainder of the shadow process address space comprises one or more specially-crafted virtual memory areas. When an attempt is made to read from or write to one of these areas, the page fault handler requests that the host process grants the shadow process access to the relevant page. The shadow process can then map that page, and satisfy the page fault.

The advantage of this approach is that it can be implemented using existing mechanisms, without modifying the VMM or core parts of the kernel[5]. A straightforward implementation of this approach is unlikely to perform well, due to the overhead of handling each page fault. However, by using techniques which determine in advance the set of pages likely to be required, it would be possible to make this work quite well. For example, out-parameters are often pointers to local variables on the caller's stack, so it may be a worthwhile optimisation to set up a permanent mapping from the shadow process to the stack in the host process.

3.3 Worked example

In order to demonstrate our technique, let us consider the example of domain building in the Xen VMM, which was the subject of our earlier work [16]. In the existing Xen tool stack, domain building (for a paravirtualised Linux domain) is implemented using the following function (shown here with simplified parameters), which is implemented in `libxenguest.so`:

[4]The library itself may contain dependencies, and we discuss this case briefly in Subsection 3.4.

[5]We do, however, need to implement an additional kernel module, but adding code using an established extension point is far less invasive than modifying the kernel itself.

```
int xc_linux_build_mem(unsigned int domid,
                       unsigned int mem_mb,
                       const char *image_buffer,
                       unsigned long image_size,
                       unsigned long *console_mfn);
```

Consider the following code which uses this function (where all variables are appropriately defined locally):

```
rc = xc_linux_build_mem(domid, mem_mb, image_buffer,
                        image_size, &console_mfn);
```

This function call will be compiled into the following assembly code:

```
lea    -0x28(%rbp),%rax    // &console_mfn
mov    -0x18(%rbp),%rcx    // image_size
mov    -0x20(%rbp),%rdx    // image_buffer
mov    -0x4(%rbp),%esi     // mem_mb
mov    -0x8(%rbp),%edi     // domid
mov    %rax,%r8
callq  4004e8 <xc_linux_build_mem@plt>
mov    %eax,-0xc(%rbp)     // rc
```

The call is made to the entry for `xc_linux_build_mem` in the Procedure Linkage Table (PLT), which is an indirect mechanism for calling dynamically-linked functions [19]. The PLT contains an indirect jump to an offset in the Global Offset Table (GOT), which is set to the function address when the library is loaded. We maintain binary compatibility with this generated code: i.e. it is possible to replace an existing dynamic library with a disaggregated library, without recompiling or (statically) relinking the executable.

When the jump from the PLT occurs, the target address is not present, and a page fault occurs. This is handled by the `vm_area_struct` which corresponds to the mapped library. For this call, it is clear that we must marshal the contents of registers `%rax`, `%rcx`, `%rdx`, `%esi` and `%edi` (corresponding to the arguments), as well as `%rip` (corresponding to the function entry point). It then notifies the library domain that a call is pending, using an interdomain event channel, and blocks pending a response.

The implementation of `xc_linux_build_mem` is not important, but it effectively creates a new virtual machine with ID `domid` and size (in megabytes) `mem_mb`. The kernel image is taken from `image_buffer`. The function passes out a frame number for the "console frame" (which is used for communication with the virtual machine) in `console_mfn`, and also returns an error code, which is stored in `rc`.

During the execution of `xc_linux_build_mem`, it must read the contents of `image_buffer` and write to `console_mfn`, which are both defined as pointers in the host process (in the assembly-code example above, these are pointers to the stack of the host process). When these addresses are accessed, a page fault occurs in the shadow process[6], and the pages must be mapped from the host process into the shadow process.

Upon returning from `xc_linux_build_mem`, the library domain must marshal the contents of `%eax`, which contains the return value from the function. It then notifies the host domain, which unblocks the host process, restores the CPU state (including the new return value in `%eax`), and resumes execution within the calling function.

[6]This assumes that no speculative mappings have been made.

3.4 Other implementation issues

In the interests of brevity, we will not go into the details of low-level implementation issues. However, we note that it will be necessary to address the problem of calling other libraries – which may be untrusted[7] – from the host process. We will therefore need a mechanism to switch back to the host process when the disaggregated library calls an untrusted function. It will also be necessary for the dynamic loader in the host process to resolve symbol dependencies in the disaggregated library, and so the library must provide limited and validated access to its relocation data structures (such as the Global Offset Table [19]).

Another implementation issue is how to decide which kernel (the host process domain or the library domain) handles system calls made by the disaggregated library. A simple solution to this would be to use the C runtime library (`libc`) that runs in the host process to perform system calls, via the above method for calling untrusted libraries.

4. RELATED WORK

The idea of isolating untrusted (or less-trusted) code has existed since the earliest days of operating systems. Corbató *et al* observed that memory protection was necessary to prevent simultaneously executing processes from interfering with each other on a time-sharing system [9]. Graham discussed protection and security in an "information processing utility" and introduced the general model of concentric protection rings, which persists in modern hardware [12]. Our approach is orthogonal to the ring structure, which enables us to have more disaggregated libraries than there are protection rings. We can also implement dependencies that are more complicated than the strict hierarchy that is enforced by ring protection.

At the same time, capabilities were conceived as a mechanism for implementing the principle of least privilege [10], and they were implemented in systems such as the Cambridge CAP computer [22], the Hydra operating system [24] and EROS [20]. As we intend to apply our technique to existing applications, we do not take an explicitly capability-based approach, though we do make use of low-level capability-like features, such as the VMM grant table [11].

Single address space operating systems (SASOSs) have greatly influenced our ideas. We share one main goal: a pointer valid in one protection domain should be valid in another. Chase *et al* drew the initial distinction between addressing and protection in their description of the Opal SASOS [6]. However, a key difference between these architectures and our own is that we aim to apply our techniques to existing applications, with minimal modifications. Opal and other SASOSs [15] continue to use explicit IPC or RPC – albeit with pointer arguments – which requires modifications to (or the rewriting of) existing applications. In addition, the SASOS approach is an extreme example of separating addressing and protection: our approach is a hybrid, which retains the use of separate process address spaces for regular operation, and only splits address spaces between different protection domains where there is a clear use case.

Several authors have previously looked at disaggregation,

[7]N.B. This does not necessarily undermine security, as long as the disaggregated library does not leak sensitive data to the untrusted code, and performs sanity checks on all data that it obtains from the untrusted code.

under a variety of names. Singaravelu *et al* extracted the security-critical components from several legacy applications, and ran these atop the L4 microkernel [21]. Provos *et al* performed privilege separation on OpenSSH, by dividing the server daemon into two processes – a privileged master and unprivileged slave – which communicate using pipes [17]. Hohmuth *et al* implemented a reduced-TCB virtual private network gateway using L^{4}Linux, also on top of the L4 microkernel [13]. Each of these projects required modifications to the code to enable communication between disaggregated components, whereas our approach is transparent to the caller of a TCB function.

Two recent works make interesting use of memory protection in order to defend against untrusted code. Witchel *et al* implemented Mondrix, which is a version of the Linux kernel that uses special hardware support to implement isolation between different kernel modules that run in the same address space [23]. While we do not assume special hardware, it would be interesting to implement our technique on a system that provides intra-address space memory protection in hardware, as this would probably improve performance. Chen *et al* implemented Overshadow, which presents an untrusted operating system with an encrypted view of processes' memory [7]. In our work, we assume that the host process trusts its operating system, but we use virtualisation to protect the disaggregated library from the operating system[8].

Automatic partitioning has also been described in the literature. Kilpatrick's Privman library can be used to perform privilege separation on Unix applications and daemons, by performing mediation on privileged system calls: an unprivileged process performs regular execution, and a trusted monitor performs the system calls, as long as the security policy is respected [14]. More recently, Brumley and Song developed Privtrans, which uses privilege annotations on functions and variables, and data-flow analysis to place the calls to the trusted monitor [5]. Chong *et al* took the novel approach of automatically and optimally partitioning a web application between server and (untrusted) client, so that integrity and secrecy constraints are preserved [8]. The optimisation in this case involves minimising the number of network messages that are sent between the client and server. We also intend to investigate automatic partitioning using our new approach, which we hope will require only small modifications to or annotations on the existing source code. We expect that, because our mechanism for disaggregation uses concepts that are familiar to mainstream software developers, it will be easier for developers to experiment with different partitioning decisions.

Dynamic (shared) libraries have previously been used as an abstraction to provide protected execution and distributed processing. Rao and Peterson introduced the concept of a Distributed Shared Library (DSLib) [18]. A DSLib can be executed either in the local address space, or on a remote host (in a different address space). However, communication with a remote instance is by traditional RPC, and requires the marshalling of arguments. Banerji *et al* developed Protected Shared Libraries (PSLs) and Context Specific Libraries (CSLs) [1]. A PSL is similar to our disaggregated library, whereas a CSL is used to implement sharing between the host process and PSL. However, this approach required changes to the host process, core operating system, linker and loader, which may have hindered its adoption. By contrast, we use a standard operating system kernel, linker and loader, and we do not require changes to the host process. Furthermore, we achieve a much smaller TCB by using a VMM to provide isolation between the host process and disaggregated library, rather than a monolithic operating system kernel.

[8]We protect the library from operating system on which the host process runs. We do not attempt to protect it from the library domain OS, which can be much smaller and hence more trustworthy than the host OS.

5. CONCLUSIONS

We have presented a new approach for performing disaggregation based on dynamic libraries. We believe that dynamic libraries are a natural unit of disaggregation, and, because they are commonly used in software development, our approach has a lower barrier to entry than existing disaggregation techniques. If adopted, it will encourage the development of software with a smaller TCB, which will in turn bring improvements in software security.

By making disaggregation easier, we also make it easier to experiment with different "cuts" through a piece of software. A valid criticism of existing work on disaggregation is that it tends to consider only a single partitioning of the code into trusted and untrusted components. This is typically based on an *a priori* judgement of how the TCB should be defined. While this partitioning is always qualitatively better than the monolithic status quo, we cannot know that it is the best partitioning unless we consider alternatives. We want to automate this process as far as possible, and we will consider this question in future work.

6. ACKNOWLEDGMENTS

Thanks are due to our colleagues Periklis Akritidis, Michael Fetterman, Euan Harris, Theodore Hong, Eric John, Stephen Kell, Grzegorz Milos, Henry Robinson, Amitabha Roy and David Simner for their comments on earlier drafts of this paper, and for many fruitful discussions about how to implement this new approach.

7. REFERENCES

[1] A. Banerji, J. M. Tracey, and D. L. Cohn. Protected shared libraries: a new approach to modularity and sharing. In *ATEC'97: Proceedings of the Annual Technical Conference on Proceedings of the USENIX 1997 Annual Technical Conference*, pages 5–5, Berkeley, CA, USA, 1997. USENIX Association.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on operating systems principles*, pages 164–177. ACM Press New York, NY, USA, 2003.

[3] M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. V. Doorn, A. Mallick, J. Nakajima, and E. Wahlig. Utilizing IOMMUs for Virtualization in Linux and Xen. In *Proceedings of the 2006 Ottawa Linux Symposium*, 2006.

[4] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: virtualizing the trusted platform module. In *Proceedings of the 15th*

USENIX Security Symposium, pages 21–21, Berkeley, CA, USA, 2006. USENIX Association.

[5] D. Brumley and D. Song. Privtrans: automatically partitioning programs for privilege separation. In *SSYM'04: Proceedings of the 13th USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[6] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4):271–307, 1994.

[7] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2008.

[8] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 31–44, New York, NY, USA, 2007. ACM.

[9] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *Proceedings of the Spring Joint Computer Conference*, pages 225–244, 1962.

[10] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, 1966.

[11] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, 2004.

[12] R. M. Graham. Protection in an information processing utility. *Commun. ACM*, 11(5):365–369, 1968.

[13] M. Hohmuth, M. Peter, H. Härtig, and J. Shapiro. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th ACM SIGOPS European workshop: beyond the PC*. ACM Press New York, NY, USA, 2004.

[14] D. Kilpatrick. Privman: A Library for Partitioning Applications. In *Proceedings of Freenix 2003*, Berkeley, CA, USA, 2003. USENIX Association.

[15] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[16] D. G. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *Proceedings of the 2008 International conference on Virtual Execution Environments*, 2008.

[17] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[18] H. C. Rao and L. L. Peterson. Distributed shared library. In *EW 5: Proceedings of the 5th ACM SIGOPS European workshop*, pages 1–5, New York, NY, USA, 1992. ACM.

[19] SCO Group. System V Application Binary Interface – Intel386 Architecture Processor Supplement, 1996.

[20] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. *SIGOPS Oper. Syst. Rev.*, 34(2):21–22, 2000.

[21] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of EuroSys 2006*, 2006.

[22] M. V. Wilkes. *The Cambridge CAP computer and its operating system (Operating and programming systems series)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1979.

[23] E. Witchel, J. Rhee, and K. Asanović. Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection. *SIGOPS Oper. Syst. Rev.*, 39(5):31–44, 2005.

[24] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: the kernel of a multiprocessor operating system. *Commun. ACM*, 17(6):337–345, 1974.