Abstract: Digital Rights Management (DRM) deals with access to information, and copying of information. Restrictions on these operations have some fundamental practical difficulties. One of these is the so-called analog hole. This is an inevitable deficiency in any DRM scheme. Once media, such as a film or a piece of music, is perceivable by a person, it can also be recorded using video equipment and a microphone. However, these forms are often of degraded quality. Furthermore, this vulnerability does not apply to software or other forms of Intellectual Property (IP). Within a computer platform, DRM requirements can be met when a small amount of software (a nanokernel) and the integrity of some of the hardware components can be trusted upon. The kernel offers trusted storage for negotiation algorithms and access management. Access is mediated by capabilities, by means of which intrinsically secure systems can be built. The kernel code, along with hardware support, defines the Trusted Computing Base (TCB). We can limit the size and price of this TCB to such extent that it can be embedded on the processor die. This high level of integration defies even highly advanced hackers. Furthermore, we can rely on a variety of other techniques to obtain platform-wide security. Long-term storage may be secured, as well as the bus connecting the peripherals in the system. Network connections with other devices can be established in a secure fashion, aided by public key cryptography and fast, hardware-accelerated encryption. The result is a versatile yet very robust system, withstanding costly and arduous attacks, while maintaining user ownership of purchased data, satisfying “fair use” policies.
1. Goals of Access Control

Digital Rights Management attempts to provide guarantees to the creators of Intellectual Property (IP). IP in this case includes computer programs, media such as music and films, documents, personal property such as credit card information, and possibly even IP (Internet Protocol) addresses [Hansell 08]. These items are often stored and exchanged in digital form.

Data access is perpetrated by agents. An agent is most often a computer process, but may also be a hardware device such as a Direct Memory Access (DMA) controller. Employing DRM serves to ensure that agents have restricted access to certain data, or conversely, that data can only be accessed by certain agents. Rights management may be imposed by the owner of a document, or anyone with sufficient privileges. These rights may include read, write or execution access to data, copying of data, or the right to communicate data over software or hardware channels.

There is some overlap in these access management requirements. Clearly, if we are to copy data, we need to read it into some buffer first, and reading and copying necessarily occur over hardware channels. However, from a use-case point of view, the requirements are disjoint. Copyright holders may allow read access, but disallow copying (for instance, in the case of media). Alternatively, a computer program might disallow read access to protect reverse-engineering, while allowing further distribution through copying. For a normal user, the instruction and resource listing of a computer program presents little value, so no usability is lost by hiding the code. In addition, thanks to the advantages offered by the capability architecture, the user can entrust his sensitive data to untrusted software.

Some important requirements for the system are listed below.

- The potential for manufacturer lock-in is minimized. Media files are stored in an open format; a complete description of their structure and possibly decoding methods is available. This does not apply to software, because it can be reverse-engineered by inspection.
- Users are in control of the data in their system, so that they may adapt media to their specific system or choose to store it on other devices they own, such as portable media players. Users may also decide to give away software or media that they own (reselling). None of these actions enable them or the recipient to infringe on IP.
- Encryption may be used liberally in the system. To minimize the impact on performance, we can use dedicated hardware to speed up the process.
2. Overview of DRM Techniques

2.1 Overview

The common denominator between various types of digital information is the way in which they are stored. Be it *Half Life 2* or an electronic copy of Stephen King’s *It*, both are encoded in the same digital primitives: bits. A sequence of bits makes up a document or game. There is no fundamental distinction between these various types of data. Few meta-data is stored with these sequences of bits. For example, on Microsoft Windows systems, file types are commonly deduced from the file name extension (set of characters after the last dot, e.g. *exe* indicating type *executable* in *solitaire.exe*). We can modify not only the file itself, but also its type and, if we are sufficiently privileged, its access rights (usually on a per-user or per-group basis). As soon as the data is loaded into our computer memory, we can exert complete control over it.

In the past few years, we have seen various takes on DRM. Major companies such as Microsoft, Apple, EMI, Sony, Warner Brothers, and many more, have implemented DRM technology with some of their products, mostly through cooperation in panels such as the DVD Forum, the Advanced Access Content System Licensing Administrator, and the Digital Video Broadcasting Project. Most of these technologies have not stood against attacks very long, for a simple reason: it is virtually impossible to distinguish between legitimate use of a data object, such as playing a movie, and copying or distributing this object – something that we are trying to prevent. With the rise of broadband internet, a new approach has been used to some success by companies such as Valve and Blizzard (both computer games manufacturers). Their software “phones home” to the respective companies, allowing them to remotely control the installed software to some degree (this is known as remote attestation). A copy of the software will be of no use, because it will not operate before having been validated over the internet.

Having the ability to contact a trusted computer system each time a program is used can provide a large degree of security. However, this method suffers from drawbacks. Primarily, a fast connection to a trusted network is needed at all times. When there is no network connectivity, the user cannot use his software. Furthermore, in addition to raising privacy concerns, the concept is difficult to extend to media, because once the media is decoded, it exists in an unencrypted form in the user system, making it vulnerable to copying. Because of these drawbacks, we shall only consider methodologies that are designed to work in isolation.

2.2 Architectural Extensions

The Protected Media Path, introduced by Microsoft in the Windows Vista operating system, places the decoding process in an isolated space. The decoded media resides only in this protected environment, and cannot be accessed from the outside. Because the protection is done entirely in (kernel) software, the kernel needs to continuously monitor itself for unverified components. A malicious kernel component could simply circumvent the PMP, reading the contents directly from memory. The PMP method is vulnerable to kernel patching, imposes severe restrictions on the user’s ability to use his media legitimately, and involves a substantial degree of memory and CPU time overhead. In essence, the PMP method relies on kernel address space protection: the fact that userland applications cannot access kernel memory.

The idea of protected address spaces is extended by Intel’s Trusted Execution Technology (TXT) technology. TXT supports Protected Execution. This allows applications to define their own isolated environments, such that no other unauthorized software on the platform can observe or compromise the data within the environment [TXT 03]. Software that is running in such a protected environment cannot even have their address space inspected or modified by the kernel. Additionally, a secure pathway can be created from the application to the framebuffer. To enable these features, some additional hardware is required.

The same concept has been proposed by [Witchel 02]. Dubbed Mondrian Memory Protection (MMP), the basic idea is to allow software to create and manage their own protection spaces. Programs are allowed to set arbitrary access rights for memory that they have allocated. MMP enables simple sharing of data and inter-process communication. Like TXT, it calls for hardware enhancements in the CPU core and Memory Management Unit (MMU). As we shall see, MMP is similar to capability-based systems. However, capability systems are much more versatile, and support
data isolation and sharing from the ground up, rather than being added to an existing system. We will learn more about capability systems in chapter 3.

A major step forward in the area of trusted computing was made by the introduction of a **Trusted Platform Module (TPM)**. This is a standardized device that can be added to an existing system. It provides trusted storage and cryptographic controls. Components of the TPM include non-volatile storage, random number generation, fast hash calculation (SHA-1), key generation, and a microprocessor to coordinate these efforts. All of these components are embedded on a single silicon die. The TPM was introduced by the Trusted Computing Group, a collaborative initiative involving approximately 140 companies, including major players such as Compaq, Hewlett-Packard, IBM, Intel, and Microsoft. It is an ongoing effort to provide drop-in security to existing platforms (the current version, 1.2 rev. 103, was introduced in 2007). By using a trusted environment to manage things like encryption keys and digital certificates, the platform cannot be compromised unless the TPM chip is compromised. The TPM has an open architecture, and there are a number of manufacturers of integrated circuits that conform to the standard, such as Atmel, Broadcom, Infineon, Sinosun, ST Microelectronics (see figure 2.1), and Winbond.

![Figure 2.1: The ST Microelectronics ST19NP18 Trusted Platform Module v1.2 chip, released in 2006.](image_url)

Any software may make use of the TPM if it is present in a system. Microsoft Windows Vista uses the TPM to facilitate its “Bitlocker” data encryption. A TPM chip is present in most new personal computers (including Macintosh [Singer 05]), but often requires the user to “opt-in” before any of the features are enabled. A TPM, be it one adhering to the current standard or one that has the same general features, is an essential ingredient in creating a secure platform. It helps to secure network communications, and is very useful for secure storage and securing other peripherals in the system. Because each TPM chip contains a unique ID, embedded during manufacturing, it may be used to uniquely identify the system, and bind data to the system (so that it cannot be decrypted on other systems).

### 2.3 Hardware Intrusion

#### 2.3.1 Attack Levels

Despite our best software efforts, we still have to consider the fact that the user has hardware access to the system. This allows for a wide variety of attacks, which we shall discuss below. First, it makes sense to establish a taxonomy, dictating the level of resistance to attacks. Such a taxonomy was proposed by [Abraham 91], illustrated in Table 2.1.
<table>
<thead>
<tr>
<th>Security Level</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>No special security features added to the system.</td>
</tr>
<tr>
<td>II</td>
<td>Some security features in place. They are relatively easily defeated with common laboratory or shop tools such as pliers, soldering iron or small microscope.</td>
</tr>
<tr>
<td>III</td>
<td>More expensive tools are required, as well as some specialized knowledge. Tool cost may range from $500 to $5000.</td>
</tr>
<tr>
<td>IV</td>
<td>Special tools and equipment are required, as well as some special skills and knowledge. The tools and equipment may cost from $5000 to $50 000. The attack may become time-consuming but will eventually be successful.</td>
</tr>
<tr>
<td>V</td>
<td>Equipment is available but is expensive to buy and operate. Cost may range from $50 000 to $200 000 or more. Special skills and knowledge are required to utilize the equipment for an attack. More than one operation may be required so that several adversaries with complementary skills would have to work on the attack sequence. The attack could be unsuccessful.</td>
</tr>
<tr>
<td>VI</td>
<td>All known attacks have been unsuccessful. Some research by a team of specialists is necessary. Highly specialized equipment is necessary, some of which might have to be designed and built. Total cost of the attack could be one million dollars or more. The success of the attack is uncertain.</td>
</tr>
</tbody>
</table>

*Table 2.1: Embedded System Security Levels.*

The type of adversary should be taken into account when designing system protection. For competing companies, the point of defeating protection measures is to save money. Our measures should thus make it more costly to reverse engineer the product, than to design it from scratch. In other cases, the attacker might be more concerned with extracting information stored on the system than reproducing it.

### 2.3.2 Bus Eavesdropping

In the simplest case, the attacker could simply eavesdrop on bus activity while the system is accessing the relevant data (e.g. playing a film). This could be accomplished relatively easily by connecting a logic analyzer to the address, data or other buses between the CPU and peripherals such as RAM or a DAC. This case is illustrated in figure 2.2. In the photograph, we see a digital video receiver. Centered are three 74244 digital buffer ICs. Buffers are easy targets, because they “conveniently” line up the system buses and signals. The pictured taps were used to snoop MPEG data and store it on a PC.

These attacks can be defeated by encrypting all bus communications. Full bus encryption requires a significant amount of overhead. In the case of RAM, no decryption hardware is needed within the RAM ICs. If the receiving device needs to decode the received data for processing, it will need a hardware extension to store keys and/or support the decryption process; key management should be carefully planned. Furthermore, not only does bus encryption cost a lot in terms of design time and silicon area, it also hinders performance.

It should be noted that tapping into modern buses is getting increasingly difficult, thanks to the continuous increase in transfer speed. This puts slower buses at a greater risk than their high-speed counterparts, simply because of negligible transmission line effects at lower speed. Unfortunately, this does not offer serious protection.
2.3.3 Physical Keep-outs

As electrical components become smaller, it becomes easier to enhance the physical barrier of the system against intruders. This is often done rather crudely, by casting the circuit in a very hard substance, such as epoxy. This approach has been used from the early time of integrated circuits, and continues to this day. Figure 2.3 shows a partial unpotting of a speech synthesizer. The attacker was able to recover EPROM data and duplicate the circuit. Current materials provide more strength than epoxy. Potting may be enhanced by embedding tamper detection devices in the potting compound. An example is the IBM 4758 secure cryptoprocessor, which, in later generations, had several layers of small nichrome wire woven around it. Tampering with the system meant breaking the wire, which caused the device to “forget” its sensitive information. This approach however relies on having battery power available at all times (losing battery power also makes the system forget its sensitive information).

Reverse-engineering a circuit becomes more difficult as individual components become smaller. Not only does the attacker require more advanced equipment to learn about or interface any part of the system, it also allows us to embed critical parts (such as key storage) inside layers of highly integrated circuitry. This is an advanced form of physical keep-outs. Such layering could for example be provided by other layers on a silicon chip, or other chips in
the case of chip stacking. Figure 2.4 demonstrates how the silicon of a smartcard was exposed, using tools that were purchased in a pharmacy for under 30 USD [Anderson 96]. The chip is fully functional, and ready for micro-probing of internal signals. Also, once the silicon is exposed, the chip can be reverse-engineered by etching away one layer at a time, followed by microscope photography, and further technologies for determining the location of N and P areas in the silicon. The produced output can be processed into mask diagrams or a circuit schematic, from which the chip can be duplicated any number of times. Of course, extracting information from the chip in this state requires detailed knowledge of the layout. Furthermore, digitizing the chip layer-by-layer requires advanced equipment, not available to level I attackers. However, the flash-based read-protect “fuse” in a PIC 18F1320 microcontroller can be reset using ultra-violet light without clearing the program storage [Bunnie 05]. This hack can be made more difficult by careful design of (redundant) fuse locations on the die.

Fig. 2.4: A smart-card IC prepared for micro-probing of the silicon.

Another example of a physical lock is a tamper switch. The product is made so that access to the critical parts, for instance through a panel that needs to be removed, immediately renders them non-functional. This can be either permanent (the components are destroyed), or resettable, so that the device can be made operational again by an authorized person. The latter might include “forgetting” encryption keys. Many PCs support this type of switch, although it is seldom implemented. Elaborate designs can be made, for example using pressured gas to detect intrusions by pressure drops. The complexity of these schemes might not always be warranted in mass-produced, mobile devices.

2.3.4 Electronic Tricks

It has been shown that unusual voltages can make processing equipment work in unpredictable ways. For example, increasing the supply voltage of a PIC16C84 microcontroller during repeated write access to the security bit, would often clear it without erasing the memory [Garbe 07]. Changing the supply voltage can also interfere with cryptographic functions, such as random number generation, by making them output all ones or all zeros.

Often, microprocessors and microcontrollers are equipped with sensors to detect unusual voltage conditions. This type of sensing is known as “brownout detection”. However, unusual conditions may exist under normal operation, for example, when inserting a smart card into a reader. For robustness reasons, the brownout detector may be configured in software, based on the duration and severity of a voltage peak. This might allow transients to pass through without raising the alarm, which could reset the protection without destroying the protected information. Brownout detection times may be as slow as 1 millisecond [Holberg].

A similar attack involves observing power usage by the device as it operates. This might enable us to infer information about its internal state. It may be used in conjunction with lowering the operating frequency of the device, or single-stepping it (controlling the processor clock manually). Solving this problem is made difficult by stringent requirements on the power consumption of mobile devices. A common solution is to add a source of random power consumption, generating signals that are indistinguishable from the real circuit.
Generally, there can be many diverse tricks, including subjecting the device or system to electromagnetic fields, radiation, temperature extremes, or other stressors. It is very time-consuming to evaluate and defend against all of these approaches.

2.3.5 Conclusion

Although “security through obscurity” is not a viable protection approach, it may be employed in combination with other techniques to increase the difficulty for low-level attackers or low production quantities. Erasing IC markings and other labels are the simplest examples. This provides very limited protection, due to information leakage and the fact that integrated circuits often have identifiers printed on their silicon die, which can be read using a microscope. By learning from previous security systems and hacking attempts against them, we can make a limited system exceedingly difficult to reverse-engineer or extract information out of. However, there exists no system that is completely immune to the attempts of a persistent high-level attacker. We are therefore left with choosing a balance between implementation cost of a security mechanism, and its merits.
3. Capability Architectures

3.1 Introduction

In common load/store processor systems, data is located by use of an address, represented by a natural number. One can think of this method as having a large number of drawers, each labeled with a distinct number. To access the contents of a drawer, you reference it by means of its number, after which you can either read or write to it. Usually, some protection exists, in that a certain agent can access only a limited number of drawers. This scheme, referred to simply as memory protection, establishes boundaries between agents. In practice, these agents are usually tasks; each task has its own segment of memory that it and it alone can access. The Memory Management Unit (MMU) enforces these boundaries. When a task attempts to access memory out of its range, the operation is blocked, and execution is interrupted (this is usually a sign of a programming error).

Capability-based systems use a different form of addressing. Instead of viewing memory as a large stack of drawers, these systems take an object-oriented approach. A capability is a ticket or token that gives the holder permission to access an object. Our only way to access any object is to reference it using a capability we possess. Although we may know that objects exist outside the ones in our list, we have no way of accessing them, because we cannot address them.

A minimal capability contains an object identifier and access rights. The identifier uniquely names a single object in the system. The access rights indicate which operations the holder of the capability is allowed to perform on the referenced object. Operations can be performed on an object by passing the relevant capability as a parameter to a function routine. When a program attempts to perform an operation, the operating system or hardware verifies that this operation is permitted, based on the access rights in the capability. Clearly, programs should be prohibited from modifying any of the parameters of the capability. Otherwise, programs could obtain access to any object by forging the ID and access bits.

In order to obtain new capabilities, programs have to rely on operating system primitives. Some other operations that can be carried out on capabilities may include deletion, restriction of access rights (producing a less privileged capability), or copying, so that it may be used by other agents as well. Allowing a program a limited amount of direct control over the capabilities it owns, such communicating them via software or hardware channels, results in an open architecture, where information can be shared in a simple way.

Each user, program, procedure, or other object, has a list of capabilities. These capabilities identify all of the resources which that object is permitted to access. The execution environment of a procedure is made up of all the resources available at that time in its capability list.

When an agent wishes to perform an operation on a resource, it extracts the relevant capability from its list, and uses it as a parameter for a procedure or (kernel-) primitive operation. If the caller’s rights are adequate, the capabilities passed as parameters by the caller are merged with the capabilities local to the procedure. This implies that a callee may have greater freedom than the caller. The potential expansion of rights across environment domains is a key factor in achieving the flexibility perpetrated by the capability architecture. If rights were not adequate, the operation is prevented by the kernel. In case of a successful call, the kernel updates the environment. Control is then transferred to the referred body of code. After the procedure ends, the kernel restores the old environment, with the exception of any capabilities that were altered by the callee.

An object is the abstracted notion of a resource. Objects can be as small as numbers or text, but can also take the form of computer programs, files or hardware devices. Each object has (at least) a unique name, and a representation, consisting of a data part and a capability list part (abbreviated as c-list; this structure was proposed by [Wulf 74]). The c-list stores references to other objects. Neither part of an object can be accessed if a suitably privileged capability is not possessed. The capability mechanism provides protection for the carrying out of operations on resources. All of the familiar security mechanisms, such as protection for files and memory, can be modeled this way. Knowledge of the unique ID of an object does not grant any access to it, because capabilities are the only method to access resources, and may only be manipulated by kernel functions. Furthermore, at no point is there any need to make this ID known to the user.
Typical rights provided for a capability may include:
- Read or write access for the representation part of an object;
- A deletion right for the referenced object;
- The walk right, which allows a caller to extract a capability from the c-list part of an object;
- The derive right, allowing objects derived from this capability to be stored independently (more about this in section 5.2);
- The copy right, allowing the capability to be copied.

Because of the walk operation, the objects available to an agent at any point in time are defined not only by the environment, but include those that can be reached by traversing a path, beginning at the capabilities in the environment, and continuing while the walk right is available.

3.2 Dealing with Capabilities

3.2.1 Physical Addresses

An advantage of capabilities is the abstraction level that they provide. However, this results in some additional issues that we need to deal with. One of these is the fact that objects are now referenced by their unique ID, instead of a physical memory location. Because the data will have to be stored in a location-addressed system (e.g. RAM or hard disk storage) at some point, we need to introduce a mapping from object IDs to physical addresses. If we were to add a field to each capability, indicating the physical address at which the object was stored, relocation would be difficult and computationally expensive. A better solution was introduced in the CAL/TSS system, in the form of a Master Object Table (MOT). This global system table contains entries for every object in the system. Each MOT entry maps an object ID to a physical address. When an object is addressed by a capability (using the unique ID of the object), the request is directed to the MOT, which provides the location where the object can be found.

A further advantage of this approach is the level of integration we may achieve with memory management. When the amount of scratch memory (RAM) is limited, secondary storage (such as a hard drive) is often used to store data that is not required to be in RAM at that time, and is not expected to be accessed any time soon. This is known as paging. When an object is accessed that is not in RAM, a page fault occurs, and the necessary data is loaded into RAM by the operating system. The Master Object Table is a convenient location to store information about the paging status of memory segments. It should be emphasized that only the kernel is aware of the distinction between main memory and secondary storage.

3.2.2 Garbage Collection

When capabilities to an object are deleted, it may happen that the object is no longer reachable by any capability in the system. These objects are known as garbage objects and may be deleted from the system. The problem of finding these objects is known as garbage collection. Unfortunately, there are as of yet no computationally efficient methods for garbage collection in capability systems. Standard approach must be used, which might include reference counting (which might be incomplete due to circular references), copying garbage collection, or optimized methods such as weighted, indirect, or distributed reference counting [Brown 99].
4. Tainting of Data

4.1 Introduction

The tainting of data is an essential ingredient to a system that will meet our requirements. The tainting concept is threefold. First, a flag is added to each data object. In its most simple form, it is a boolean value, but it can also be a complex structure, including pointers to other data. Second, propagation rules are defined for flags. When operations are carried using data that is flagged, the outcome of the operation may be flagged as well. Finally, some operations may be denied if the operands carry a taint flag. For example, we may define our rules so that data that entered the system through the network may not executed or evaluated.

Depending on the type of flag and its semantics, we can use tainting to defy control-flow hijacking\cite{Xu06}, SQL injection\cite{Xu06}, buffer overflows\cite{Newsome05}, Format String Attacks\cite{Sub04} and many other attacks. We shall look at an example taint method, proposed by \cite{Xu06}, to illustrate how tainting works. This method is targeted towards existing load/store hardware, and specifically, the C programming language. Each byte in memory carries a boolean taint flag $\tau$, where $\tau = 1$ marks tainted data, and $\tau = 0$ indicates an untainted byte. These flags are stored in an array known as a tag map. We can extract the taint status of a byte at address $a$ by dereferencing the tag map at this index: $\text{tagmap}[a]$.

Table 4.1 lists the rules for determining the taint value of an expression $E$ (the operator semantics are ANSI C: ‘&’ yields the address of the expression following it; ‘*’ dereferences the address following it). Note that $\text{tag}(a, n)$ refers to $n$ bits starting at $\text{tagmap}[a]$.

<table>
<thead>
<tr>
<th>$E$</th>
<th>$\tau(E)$</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>0</td>
<td>Constants are untainted</td>
</tr>
<tr>
<td>$v$</td>
<td>$\text{tag}(&amp;v, \text{sizeof}(v))$</td>
<td>The taint of a variable $v$ can be looked up in the tag map</td>
</tr>
<tr>
<td>$&amp;E$</td>
<td>0</td>
<td>Addresses are always untainted</td>
</tr>
<tr>
<td>$^*E$</td>
<td>$\text{tag}(E, \text{sizeof}(^*E))$</td>
<td>Typecasts do not change taint</td>
</tr>
<tr>
<td>$(\text{cast})E$</td>
<td>$\tau(E)$</td>
<td></td>
</tr>
<tr>
<td>$\text{OP} E$</td>
<td>$\tau(E)$</td>
<td>For arithmetic/bit OP, e.g. $\text{OP} = {\text{inc, dec, ¬, etc.}}$, otherwise 0</td>
</tr>
<tr>
<td>$E_1 \text{ OP } E_2$</td>
<td>$\tau(E_1) \lor \tau(E_2)$</td>
<td>For arithmetic/bit OP, e.g. $\text{OP} = {\land, \lor, +, \div, \ast, \text{etc.}}$, otherwise 0</td>
</tr>
</tbody>
</table>

Table 4.1: Determining taint for expressions.

Next, rules are defined for transforming C code. This transformation needs to take place because no supporting hardware exists on the target platforms. If this hardware was included, taint propagation and checking would be done transparently. The transformation of a statement $S$ is defined by $\text{Trans}(S)$.

Finally, policies are introduced, which allow the system to deny requests based on the taint values of the operands. For example, to prevent control-flow hijack, we introduce the policy: Tainted values cannot be used as a target of control-flow transfer. This precludes the processor from jumping to the address indicated by a pointer $p$, if $p$ is tainted. Similarly, to prevent SQL injection attacks, we specify the policy: SQL query strings should not contain tainted SQL meta-chars (e.g. ‘‘, ‘’’, ‘;’, ‘/*’). Program code for implementing these policies is added where it is needed. In the previous example, the procedure for evaluating queries has to be augmented with policy verification code.
The previous example illustrated how high-level (C) code can be transformed to enable taint methods on standard architectures. We shall now look at an architecture that is optimized for taint tracking. Having the luxury of hardware support reduces performance overhead significantly. No longer do we need code transformation for taint tracking. Propagation rules are defined for processor instructions and registers, instead of higher-level statements and variables. The operating system (OS) still needs to explicitly flag untrusted data. Furthermore, some policies still need to be explicitly implemented for each different type of attack. [Suh 04] describes what hardware support for taint tracking might look like. Like before, one-bit taint flags are used. In principle, each byte or word in memory is tainted (some optimizations are suggested to reduce memory overhead). Taint flags are also added to each CPU register. Most of the propagation rules have the same philosophy as before: the result of a calculation is tainted if any of the operands is tainted, et cetera. They are formalized in table 4.3.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Meaning</th>
<th>Tag Propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and logic</td>
<td>ADD R1, R2, R3</td>
<td>R1 ← R2 + R3</td>
<td>(\tau(R1) \leftarrow \tau(R2) \lor \tau(R3))</td>
</tr>
<tr>
<td>Load</td>
<td>LW R1, Imm(R2)</td>
<td>R1 ← Mem[R2+Imm]</td>
<td>(\tau(R1) \leftarrow \tau(Mem[R2+Imm]) \lor \tau(R2))</td>
</tr>
<tr>
<td>Store</td>
<td>SW Imm(R1), R2</td>
<td>Mem[R1+Imm] ← R2</td>
<td>(\tau(Mem[R1+Imm]) \leftarrow \tau(R1) \lor \tau(R2))</td>
</tr>
<tr>
<td>Register clear</td>
<td>XOR R1, R1</td>
<td>R1 ← 0</td>
<td>(\tau(R1) \leftarrow 0)</td>
</tr>
</tbody>
</table>

Table 4.3: Taint propagation rules. Simplified after [Suh 04] (ignores Propagation Control Register, which allows disabling of select propagation rules).

The Xu and Suh systems both incur a large memory overhead due to the fine granularity at which the taint flags are stored. In normal processors, the overhead can be reduced depending on the addressing capabilities. For example, if we wish to mark each byte with a single taint bit, the overhead is 12.5%. If a 64-bit CPU can only address memory that is aligned at 64 bits, the overhead per word is 1.6%. It can be further reduced not storing absence-of-taint flags, delaying storage until objects are allocated (dynamic taint table), or increasing the granularity, by e.g. marking entire memory pages with a single bit. These memory optimizations spoil the uniformity of the system, reducing the
potential for hardware optimizations and increasing the potential for new security holes. Fortunately, the capability architecture provides us with a convenient “unit of data”, namely, the object. As we have learned in the previous chapter, all data in a capability system is contained within objects. The capabilities, which are used to address these objects, already carry a number of other parameters. We can simply add the taint flag to these existing parameters.

4.2 Control Dependency Data Leakage

We have now seen the basic properties and capabilities of data tainting. There is still one major security hole in these schemes. Consider listing 4.4. No explicit assignment is made from x to y. However, after the code completes, y has taken on the value of x. This illustrates data leakage through control dependencies. The previous examples showed systems where the focus was on preventing software exploitation. The aim of our own system is to prevent data leakage. Clearly, it is critical to find a solution.

```cpp
bool x, y;
...
y = false;
if (x) then y = true;
```

Listing 4.4: Data leakage through control dependency.

At the instruction level, conditional jumps do not look like an if..then..else statement. A CPU can only do a limited amount of conditional branching. Branch conditions usually apply to a single register, and are stored as processor flags. The most common ones are negative, zero, overflow, and carry. These flags indicate the status of a single register, and are themselves stored in the Processor Status Register (PSR). Control can then be transferred based on the value of any of these flags. When the register that the flags apply to was tainted, the taint value should be inherited by the PSR. Then, whenever an assignment or store operation is carried out, the taint value of the result includes that of the PSR. We thus arrive at a taint propagation ruleset that is suited for our capability system (table 4.5). The policies associated with taint flags are introduced in chapter 5.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Tag Propagation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and logic*</td>
<td>R1 ← R2 + R3</td>
<td>τ(R1) ← τ(PSR) ∨ τ(R2) ∨ τ(R3)</td>
</tr>
<tr>
<td>Load</td>
<td>R1 ← Mem[R2+Imm]</td>
<td>τ(R1) ← τ(PSR) ∨ τ(Mem[R2+Imm]) ∨ τ(R2)</td>
</tr>
<tr>
<td>Store</td>
<td>Mem[R1+Imm] ← R2</td>
<td>τ(Mem[R1+Imm]) ← τ(PSR) ∨ τ(R1) ∨ τ(R2)</td>
</tr>
<tr>
<td>Register clear</td>
<td>R1 ← 0</td>
<td>τ(R1) ← 0</td>
</tr>
<tr>
<td>Evaluate register</td>
<td>PSR ← R1?</td>
<td>τ(PSR) ← τ(R1)</td>
</tr>
</tbody>
</table>

* Operations are assumed to have no side effects (e.g. set PSR bits)

Table 4.5: Taint propagation rules for DRM-enabled capability system.
5. Access Control in Capability Systems

5.1 Data Security in a Capability System

5.1.1 Advantages of a Capability Architecture

Previously, DRM techniques have been based on restriction. Barriers are erected between data, in order to withstand access attempts made by agents. Access is granted, unless some restricting rule applies. Because of the variety of agents (both hardware and software), we also need a variety of techniques to hold out their requests. When systems are expanded with new technology, such as MPEG or DSP hardware, specific solutions have to be invented and applied, which is not only time consuming, but leads to difficulty when buying or sharing Intellectual Property (IP).

An opposite approach is used in capability systems. Instead of a policy of restriction, it uses a policy of allowance. In capability systems, resource access is not implicit. In ordinary systems, merely knowing the memory address of some hardware or software object allows you to access it - if it were not for additional rules, designed to deny a request after it has been made. In capability systems, each user, program or procedure only has access to those objects that are explicitly enumerated in its access list. An object cannot be accessed if such an entry (or capability) is not available. Capabilities act as tokens or keys to data, and they are the only way to address data within the system. For example, the possession of one capability may give us read access to some file, whereas another may be used to access a network connection. The propagation of capabilities is well managed, and is enforced by a trusted computing base (TCB). Data access is now a privilege, which can only be exercised by possessors of a suitable capability. This open approach allows copy protection methods to naturally co-exist with sharing and exchange of data.

The capability architecture enables us to contain the spread of information in two ways. First, an object is not accessible – by either software or hardware – if a capability for it is not owned by the agent requesting the data. This is discussed in section 5.2. Second, by constraining the operations that may be carried out on capabilities, the architecture supports data hiding. Such constraints may be imposed by software, and are thereafter enforced by hardware (the TCB). This is discussed in section 5.4.

5.1.2 Memory Allocation and Release

When a new object is made, the program that is used to create this object will make a system call to request memory space to store it. It will then receive a capability from the kernel or Memory Management Unit (MMU). This system call cannot be intercepted. The same method is used to call BIOS routines in regular computer systems. First, the parameters, detailing the specifics of the new capability, are set up in designated registers or on the stack. Then, the request is made by generating an interrupt. This causes the processor to resume execution in trusted memory space, where the interrupt handler is located. The interrupt handler takes care of the memory allocation. It can then stall until the memory allocation request is granted by the MMU, or resume execution, and use flags or further interrupts to signal a task that its requested memory is ready. The latter may be preferred in real-time systems, because it is easier to determine the upper time bound of such an allocation handler.

When memory space for a new allocated object overlaps with a previously de-allocated object, protected data could inadvertently leak. We should therefore clear the memory before granting read access. Unfortunately, the runtime overhead involved with this is quite high; we have to clear each memory location that might contain sensitive information. It is more efficient to clear only that data that is marked as “sensitive”. Thanks to the object-oriented architecture, these locations are well defined, namely, by the parameters in the capabilities. The garbage collection mechanism should thus clear the memory locations of these objects when no references to the object remain.
5.2 Sharing and Containment of Capabilities

5.2.1 Confinement

The subject of delegation of authority has had some attention in the literature. Suppose Alice wants to authorize Bob (and Bob alone) to access a resource R. Of course, Alice needs to possess a capability to R herself. She can then share access by communicating a capability for R to Bob. However, as soon as Bob receives R, he can communicate it further to other objects that he has access to. This is known as the confinement problem. A number of solutions have been suggested. First of all, the receiving program can only distribute the capability to objects that itself has access to. Suppose Alice does not want Eve to get a hold of this capability. Bob cannot share it with Eve, so long as Bob has no capability for Eve. Although this method is a direct result of the principle of minimum authority, central to capability systems, it is often impractical, because receiving programs may have access to a wide variety of other objects, including storage and network.

A different, straightforward solution has been suggested by [Wallach 97]. As we have learned, the creation of capabilities may only be done by the operating system. We can make the delegation of capabilities a similarly privileged operation. However, forcing data sharing operations to go through the kernel undermines one of the fundamental reasons for using a capability architecture in the first place, namely, straightforward sharing of resources. The KeyKOS capability kernel implements a method where the caller can define the output channels of the callee. Quoting [Lampson 73]: “A program to be confined must allow its caller to determine all its inputs into legitimate and covert channels. We say that the channels are masked by the caller.” Although this may allow the owner of a capability complete control over the distribution of the capability and possible derivates, this method is not only difficult to tune to the system (on software and hardware levels), it is also considered too restrictive, because it prohibits fair use and customization (e.g. volume normalization for an audio file, or brightness compensation for a film).

5.2.2 Revocation

We may wish to revoke access to resources, some time after we have delegated access authority. Capabilities are not inherently revocable. However, we can set up a forwarding chain, which can be broken at any time by the agent that delegates the authority. Instead of authorizing the callee for the target resource, we give it access to a forwarding agent. When the caller wants to revoke access to the resource, he simply breaks the chain by disabling or removing the forwarder. This is illustrated in figure 5.1. Alice wishes to provide Bob with revocable access to Carol. To do this, she first sets up the forwarding chain, consisting of F (the forwarding agent) and R (the revoking agent). Alice then sends Bob a capability for F, while retaining R for herself. Any messages that get sent to F are forwarded to Carol through R. Because of the unified addressing and message passing scheme in capability systems, Bob can use F as if it were Carol. When Alice wants to revoke Bob’s access to Carol, she instructs R to stop forwarding. F then becomes useless to Bob. Even if Bob tries to further delegate the authority, he can never provide access to Carol directly, only to F. When R is disabled by Alice, this stops the forwarding from all agents whom Bob may have authorized. Note that no capabilities themselves were revoked in this scheme [Miller 03].
5.2.3 Restriction Inheritance

A capability may be a pointer to a complex data structure, such as a tree. If we want to provide read access to the structure to another agent, we would send a capability to the root, marked as ‘read-only’. The receiving agent can then inspect the child nodes by extracting their respective capabilities from the c-list. However, these capabilities are not necessarily marked as read-only, allowing the receiver to modify the structure, even though the initially shared capability did not permit this.

A solution to this problem is offered in KeyKOS by means of so-called sense keys. They are equivalent to normal capabilities, with the exception that any key delivered to the holder as a result of invoking a sense key is the sensory version of the key being fetched. Sensory keys cannot convey modification rights. Because sensory keys may not be used to obtain non-sensory keys, we can issue read-only access not only to a single object, but to an entire data structure, by issuing a sensory key for it.

5.3 Encapsulation

The goal of copy protection, in terms of capability systems, is to fix the number of object instances available to the user. Copies of any data can only be made by reading the source data, storing it in a buffer, then writing it to some destination. Because of access limits we may impose on a capability, we can allow computer software to execute, while disallowing all read operations on the software object. This can be accomplished simply by not providing a more privileged capability for the software object. Because no higher-privileged capabilities can be forged or derived, no agent is permitted to perform read operations on the object. Note that this does not imply that the (read-only) capability itself cannot be copied.

Thanks to the versatility of the capability scheme, it is easy to provide the program any desired access to its own internal data objects, and external items such as dynamic libraries. By establishing a hierarchy, the only capability that the OS has to deal with is that of the entire program. Within this, there may be a plurality of other capabilities, accessible to the program itself, but hidden to the outside world. Even though the program is operating on a remote computer, knowledge about the system ensures us that the user cannot access the program data.
5.4 Protecting Derived Data

As discussed before, the processing of a file – for example, playing an MPEG encoded film – generates data that is derived from the original, possibly engineered by malicious software so that the original can be duplicated. Unfortunately, this decoding process is inherent to playing modern, digitally encoded media. Therefore, we need to have the option of keeping the initial as well as intermediate capabilities valid and accessible for the entire length of the processing.

The solution is to irrevocably link the original and derived objects. This collection can be seen as a single object. It is treated as a single, protected entity; in fact, the rights for the combined object are the same as for the original object. Derived objects, including intermediate data to a decoding process, can still be individually addressed and discarded explicitly or due to garbage collection after the decoding is complete. Alternatively, in case of intentional modification by the user or other reason for storage, derivates may be saved to hard drive. The user can then opt to discard or overwrite the original. Altogether, the requirements for fair use and user control are satisfied. Invalidation of one capability in an object derivation chain automatically invalidates the rest of the chain.

To allow such grouping of objects into a single collection, each object (rather than capability) contains a collection $\Pi$ (for Parents). When an object $C$ is marked by the taint rules, the reference in the taint flag is used as a pointer. The object $P$ associated with this pointer is added to the $\Pi$-list of $C$:

$$C.\Pi \leftarrow C.\Pi \cup P$$

(equation 5.2)

Furthermore, all objects in the $\Pi$-list of the parent $P$ are also added to the $\Pi$-list of $C$:

$$C.\Pi \leftarrow C.\Pi \cup P.\Pi$$

(equation 5.3)

These operations effectively create unidirectional, referential links from a derived object and to all of its parents. Figure 5.4 shows an example of what such a referential structure might look like at some point during a JPEG decoding process. Downward, black arrows indicate which object was derived from which. Dotted red arrows indicate the references that one object has towards another.

---

**Figure 5.4:** Part of a single group of linked objects.
Now that the complete tainting procedure is known, we can define the taint policy, which shall enforce the copy-protection on flagged objects. When a user wishes to communicate data that he owns, but is under DRM restriction, he can set up a secure network link to another computer. A protected object may then be sent over the secure link, after which the local copy should be invalidated.

The key to enforcement is twofold. First, invalidating one object in a derivation chain invalidates all others that were in its Π-list. Second, for an object to be valid on a system, all of the objects in its Π-list have to be valid as well. Otherwise, the object is itself invalidated. Using this protection scheme, we can manage a dependency link between objects without requiring recursive search algorithms. Dependencies are simply evaluated at time of access, which for a Π-list of $n$ elements can be done trivially in $O(n)$ time and can easily be sped up by hardware extensions.
6 System-wide Security

6.1 Non-volatile Storage

Capabilities provide us with data security within the trusted system (TCB). We have to decide at an early point in development whether the non-volatile storage should be part of this TCB or not. A set-top box might not need a great deal of non-volatile storage, but a portable music player is likely to have a large amount of flash memory. This memory is often highly integrated with the device, so that the integrated circuits are not extractable by the user. However, in many other cases, we would like the memory to be replaceable (in case of malfunction), expandable, or exchangeable. In this case, industry-standard storage devices, such as CompactFlash (CF) or Secure Digital (SD) cards, are preferred, simply because they are established as mature technology, making them cheap and widely available. Although we could use these standard components in conjunction with hardware tamper-detection as an intermediate solution, the downsides of these techniques are known from section 2.2.

When the storage drive of managed data can be extracted from the system without breaching the TCB, we need to take extra steps to safeguard it, regardless of the environment it finds itself in (e.g. systems that are or are not capability-enabled). Primarily, this means that sensitive data may not be stored unencrypted. This would make it trivial for it to be extracted, bypassing all security mechanisms. Further, it is our aim to continue to use industry-standard components.

Fortunately, there has been some initiative in the industry to provide support for this. Sealed storage protects private information by binding the data to the platform that was used to write it to disk. This means that the data can only be read on the same system that was used to write it. Increasingly, hardware support for sealed storage is being embedded inside storage devices. The aforementioned SD card supports a set of technologies called Content Protection for Recordable Media (CPRM). The CPRM method relies on media and playback devices being given unique and secret keys by a central authority. The process goes as follows [4C ENT 07]:

1. A trusted authority provides secret device keys to the device manufacturer for inclusion into each device produced.
2. Media manufacturers place a Media Identifier and Media Key Block generated by the trusted authority on each piece of media.
3. When compliant media is placed within a compliant device, a secret Media Key is generated by the device using its secret keys and the Media Key Block stored on the device itself. The same Media Key is generated regardless which (compliant) device is used to access the media.
4. Content stored on the media is encrypted or decrypted by a Content Key derived from a one-way function of a secret Title Key and the Copy Control Information (CCI) associated with the content. The Title Key is encrypted and stored on the media using a key derived from a one-way function of the Media Key and Media ID.

The CPRM scheme provides content encryption, copy protection for the media itself (i.e. outside a compliant/trusted system), renewability of protected content, and applicability to different types of media. The encryption methods were designed from the ground up to be applicable to DRM-restricted media, so it is especially well suited for inclusion in our capability system.

![Figure 6.1: CPRM protection on an SD-card.](image)
6.2 Network Communication

In order to satisfy our “fair use” requirement, the user should be able to transfer data to and from other devices he owns. Of course, these devices have to be based on a secure platform if we are to extend our trust towards them. We can use the Trusted Platform Modules (TPMs) introduced in section 2.2 to identify trusted platforms beyond our own. This includes systems that are not owned by the user. Indeed, another requirement was that the owner of an object (the end-user) could give it away to a friend, or resell it. This can be done if we can first establish the integrity of the receiving platform, and secondly establish a secure connection to it. After transmitting, we again rely on the hardware in our own system to invalidate the local copy.

A variety of network protocols can be used to establish a secure connection to another system. We can rely on the features built into the TPMs on either side of the connection for this security.

6.3 Other Peripherals

The platform may include a variety of other peripherals, including audio and video processing features. In order to be able to trust these devices, we have to validate their authenticity by means of certification. Although this adds some complexity to these peripherals, it is an inevitable step in order to provide end-to-end encryption. This certification goes hand-in-hand with the earlier proposed bus encryption (see section 3.3). In fact, the requirements are a lot like those of an individual Trusted Platform Module (TPM) chip. It is conceivable that the equivalent of such a chip will be embedded in the silicon of all integrated circuits in a system, to provide distributed trust management.
7. Conclusion

The security of a system is defined by the security of its components. In the previous chapters, we have dealt with the components that constitute a modern, rich multimedia system, such as a personal computer or mobile embedded device. These components include networking, non-volatile data storage, the processor, RAM, and interconnects. Because DRM is a global question that is in dire need of a good answer, many efforts have been and are being undertaken to satisfy the requirements on end-user platforms. In constructing a secure platform, we can make use of the results of past efforts. The main result is a multi-purpose component: the Trusted Platform Module (TPM). This “drop-in security” device can be used to secure storage, networking, and other peripherals. However, DRM solutions still fall short when it comes to securing the processor itself. On standard load/store processors, the kind of which we see in everything from washing machines to personal computers, it is usually trivial to circumvent software protection methods, and inspect or make copies of media that is protected by intellectual property, copyright, or other law. By combining tainting techniques with the intrinsically secure capability architecture, a solution to securing the processor and the software that runs on it can be provided – the last piece of the puzzle.

Capability systems have been built in the past, sometimes for research purposes, other times for commercial reasons (to varying degrees of success). Also, processors supporting data tainting have been built. This demonstrates that the implementation of a combined taint/capability system is feasible. Thanks to current rapid-prototyping technology (Field-Programmable Gate Arrays or FPGAs), it would be possible to build such a RISC processor within reasonable time. Although the performance cannot be expected to equal that of current PC processors, with GHz-range clock speeds and myriad optimizations, the “embedded devices” target market is ever growing, and not in need of dazzling instructions-per-second numbers. Performance per consumed Watt of power is at least as important. Because these embedded platforms include mobile media players, not only music but also video (something which most modern cellphones are capable of), the target market for such a processor is enormous.

The downside of the proposed architecture is its fundamental difference from existing processors. The machine code differs to such a degree that running existing software is not possible. Therefore, it will need to be mostly recompiled from higher-level languages. Emulation is also an option, but causes performance degradation. Binary code transformation may be considered. In this case, the “old” machine code is statically analyzed and transformed into code suitable for the new processor. Because of the limited knowledge we have over the original program, this method may also suffer from performance loss.

A related effort is being undertaken by Microsoft, as part of their research operating system Singularity. This OS is also based on the notion of intrinsically secure software; no program can make a data access attempt outside of its own address space, even if it (maliciously) wanted to. Unfortunately, Singularity has to run on existing hardware, which as we know makes it very simple to violate address space boundaries. Isolation is achieved instead by statically analyzing the high-level (C#) code for type safety. The software may then be given a certificate of validity, after which it can be trusted to run without interfering other programs. The certification infrastructure introduces a lot of unnecessary complications. Once the capability hardware is built, no software verification is required.

The desire is to have a hardware architecture that is secure by nature. No firmware or software should need to be implemented by Original Equipment Manufacturers (OEMs) to support the claimed features. Many of the capability functions, including providing safety with regard to memory allocation and re-use, can be implemented as part of a Memory Management Unit (MMU). Others are of a more algorithmic nature. These can be implemented in an on-die ROM in a traditional manner, akin to microcode: a very restricted set of hardwired programming primitives. Any off-chip firmware is at risk of compromise. Regardless of the implementation details, it has been shown that the required extra functionality is small enough to allow hardware embedding.
8. References

References follow in alphabetical order.

    http://www.4centity.com

[Bunnie 05]: “Hacking the PIC 18F1320”
    Bunnie Studios
    http://www.bunniestudios.com/blog/?page_id=40

[Anderson 96]: “Tamper Resistance - a Cautionary Note “
    Markus Kuhn, Ross Anderson

[Bossuet]: Presentation by Lilian Bossuet, Guy Gogniat, Wayne Burleson

[Brown 99]: “Memory Management on a Massively Parallel Capability Architecture”
    Jeremy Hanford Brown
    1999

[Chander01]: "A State Transition Model of Trust Management and Access Control"
    Ajay Chander, Drew Dean, John Mitchell
    14th IEEE Computer Security Foundations Workshop

[Garbe 07]: Patent “Semiconductor device and method for preventing attacks on the semiconductor device”
    Joachim Garbe, NXP B.V. (Netherlands)

[Hansell 08]: “Europe: Your I.P. Address Is Personal”
    Saul Hansell
    January 22, 2008
    http://bits.blogs.nytimes.com/2008/01/22/europe-your-ip-address-is-personal/

[Holberg]: “Innovative Techniques for Extremely Low Power Consumption with 8-bit Microcontrollers”
    Arne Martin Holberg, Asmund Saetre (Atmel)

[Horton 06]: Kevin Horton
    February 2006
    http://kevtris.org/Projects/votraxml1/index.html

[Lampson 73]: “A note on the confinement problem”
    Butler W. Lampson
    Communications of the ACM, Vol 16, No. 16, 1973

[Miller 03]: “Capability Myths Demolished”
    Mark S. Miller, Ka-Ping Yee, Jonathan Shapiro

[Newsome 05]: “Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software”
    2005
    In Proc. of the 12th Annual Network and Distributed System Security Symposium
    James Newsome and Dawn Song.
[Singer 05]: “Much ado over Apple-Intel developer box”
    Michael Singer (CNet)
    Aug. 4, 2005

[Suh 04]: “Secure Program Execution via Dynamic Information Flow Tracking”
    Proceedings of the 11th international conference on Architectural support for programming languages and operating systems
    G. Edward Suh, Jae W. Lee, David Zhang, Srinivas Devadas

[TXT 03]: “Intel Trusted Execution Technology Architectural Overview”
    Intel Corporation
    2003

[Wallach 97]: “Extensible Security Architectures for Java.”
    D. S. Wallach, D. Balfanz, D. Dean, E. W. Felten

[Witchel 02]: “Mondrian Memory Protection”
    Emmett Witchel, Josh Cates, and Krste Asanović
    2002
    Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems

[Wulf 74]: “HYDRA: The Kernel of a Multiprocessor Operating System”
    W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack
    Communications of the ACM, June 1974, volume 17, number 6

[Xu 06]: “Taint-Enhanced Policy Enforcement”
    2006
    Proceedings of the 15th conference on USENIX Security Symposium - Volume 15
    Wei Xu, Sandeep Bhatkar, R. Sekar